Communication-avoiding Krylov subspace methods

Mark Frederick Hoemmen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-37
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-37.html

April 2, 2010

Communication-Avoiding Krylov Subspace Methods

By

Mark Hoemmen

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor James W. Demmel, chair
Professor Katherine A. Yelick
Professor Ming Gu

Spring 2010

Contents

Acknowledgments

My advisor, Professor James Demmel[1], has been a great help for the entire long journey of my graduate studies. His dedication and hard work have inspired me since the day we first met. Thankfully me spilling tea all over his office carpet that day did not put him off from working with me! I appreciate his leadership along with that of Professor Kathy Yelick[2] of the Berkeley Benchmarking and Optimization ("BeBOP") Group, the research group in which I spent my years at Berkeley. Many thanks also to Professor Ming Gu[3], who served on my thesis and qualifying examination committee, and Professor William Kahan[4], who served on my qualifying examinations committee. Discussions with them have been a great help in shaping my research. They have given generously of their time, despite their busy schedules. I also appreciated the advice of Professors Beresford Parlett and Keith Miller (Mathematics), with whom I had the pleasure of many lunchtime conversations after the weekly Matrix Computation Seminar. Conversations with Professor Armando Fox (Computer Science) gave insights into how to make automatic performance tuning accessible to users, without adding to the complexity of user interfaces. Other University of California Berkeley faculty whom I would like to thank here are Professors John Strain (Mathematics), Per-Olof Persson (Mathematics), and Sara McMains (Mechanical Engineering).

My fellow graduate students at the University of California Berkeley have been a fantastic help during my studies and research. I would first like to thank my collaborator Marghoob Mohiyuddin for his exceptional hard work and assistance. We could not have finished several projects without him. His quiet and selfless dedication deserves fullest respect and admiration. Many other students also deserve thanks, in particular Grey Ballard, Erin Carson, Bryan Catanzaro, Kaushik Datta, Ankit Jain, Brian Kazian, Rajesh Nishtala, Heidi Pan, Vasily Volkov (in the Computer Science Department), and Jianlin Xia (now a Mathematics Department faculty member at Purdue University), and Sam Williams (now a postdoctoral researcher at Lawrence Berkeley National Laboratory). I enjoyed working with all of the students, faculty, and staff in the new Parallel Computing Laboratory. Discussions with them made me more familiar with consumer applications of scientific computing than I would have been otherwise.

I would also like to thank collaborators from other institutions. Dr. Laura Grigori (INRIA, France) and Professor Julien Langou (University of Colorado Denver) contributed immensely to our work with the Tall Skinny QR factorization (see Section 2.3). Drs. Xiaoye

[1]UC Berkeley Computer Science and Mathematics
[2]UC Berkeley Computer Science, and Lawrence Berkeley National Laboratory
[3]UC Berkeley Mathematics
[4]UC Berkeley Computer Science and Mathematics

"Sherry" Li (Lawrence Berkeley National Laboratory) and Panayot Vassilevski (Lawrence Livermore National Laboratory), along with the previously mentioned Jianlin Xia, gave helpful insights into the possible relationship between multigrid and the preconditioned matrix powers kernel (see Section 2.2), during a series of discussions in 2006. Dr. Sam Williams (Lawrence Berkeley National Laboratory) offered many useful suggestions for improving the performance of the computational kernels discussed in Chapter 2. In particular, he suggested using asynchronous task scheduling across multiple kernels to avoid synchronization overhead. Dr. Rajesh Nishtala (then a UC Berkeley graduate student, now at Facebook) provided shared-memory communication primitives. Sparse matrix manipulation code (and advice) from Professor Rich Vuduc (Georgia Institute of Technology) also proved helpful in the performance benchmarks for which we show results in Section 3.6.

Many people at Sandia National Laboratories contributed helpful advice for the presentation of my research results. These include Drs. Michael Heroux, S. Scott Collis, and Erik Boman. Dr. Collis has been especially patient and helpful with many complicated logistical arrangements. Many thanks also to many UC Berkeley staff, including Tamille Johnson, Laura Rebusi, the Parallel Computing Laboratory work-study students, and La Shana Polaris.

My wife Mingjie has been and continues a great support to me for all these years. I cannot express how grateful I am for her love and patience. There are many others whose names I do not have space to list here, who have contributed research insights, encouragement, and moral support.

This research was supported mainly by funding (Award #20080469) from Microsoft and Intel, with matching funding from the University of California Discovery program (Award #DIG07-10227). I also received financial support from a student internship at Sandia National Laboratories,[5] the ACM[6] / IEEE[7]-CS High Performance Computing (HPC) Fellowship Program, a student internship at Lawrence Livermore National Laboratory,[8] and an NSF[9] fellowship (2003 – 2005).

[5]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000. Special thanks to the Department of Energy Office of Advanced Scientific Computing Research (ASCR) for funding through the Extreme-scale Algorithms and Architectures Software Initiative (EASI) Project.

[6]The Association for Computing Machinery: see `http://www.acm.org/`.

[7]The Institute of Electrical and Electronics Engineers: see `http://www.ieee.org/`.

[8]Lawrence Livermore National Laboratory is operated by Lawrence Livermore National Security, LLC, for the Department of Energy's National Nuclear Security Administration.

[9]The National Science Foundation: see `http://www.nsf.gov/`.

Notation

Linear algebra

Throughout this work, we usually follow the "Householder convention"[10] for notating linear algebra:

- Greek letters indicate scalars,
- lowercase Roman letters indicate vectors (or scalar indices, especially if they are i, j, or k), and
- uppercase Roman letters indicate matrices.

Sometimes, especially in Chapters 3 and 4, we extend this, so that uppercase Black letter letters such as $\mathfrak{Q}$, $\mathfrak{R}$, or $\mathfrak{V}$ (those are Q, R, and V, respectively, in the Black letter typeface) indicate matrices which are constructed out of other matrices. For vectors and matrices, we also use subscripts in the following ways.

- v_k denotes a term in a sequence of vectors v_0, v_1, ..., v_k, v_{k+1}, ... Generally all these vectors have the same length. The subscript k does *not* denote element k of the vector; for that we use the notation $v(k)$, imitating the programming language Fortran.
- The notation $v_k(i)$ means element i of the vector v_k, and the notation $v_k(i:j)$ (imitating Fortran 90 and later versions of the programming language) means the vector of length $j-i+1$ containing elements i through j (inclusive) of v_k in sequence.
- V_k denotes a term in a sequence of matrices V_0, V_1, ..., V_k, V_{k+1}, ... Generally these all have the same number of rows. They may or may not have the same number of columns.
- We again use the Fortran-like notation $V_k(i,j)$ to denote the element in row i and column j of the matrix V_k. If the matrix has no subscript, we may write V_{ij}, which means the same thing as $V(i,j)$.
- The notation $V_k(:,j)$ means column j of V_k, and $V_k(i,:)$ means row i of V_k. If V_k is $m \times n$, $1 \le i_1 \le i_2 \le m$, and $1 \le j_1 \le j_2 \le n$, then $V_k(i_1:i_2, j_1:j_2)$ denotes the submatrix of V_k containing rows i_1 to i_2 (inclusive), and columns j_1 to j_2 (inclusive) of V_k.

[10] Thus called because the English-speaking numerical linear algebra community standardized around slight modifications of the notation of Householder [138].

- We write $0_{m,n}$ for an $m \times n$ matrix of zeros, and 0_n for a length n column vector of zeros.
- Imitating Wilkinson [240], we write e_j for a column vector of some context-dependent length n, consisting of zeros except for a one in entry j. Thus, the $n \times n$ identity matrix I can be written as $I = [e_1, e_2, \ldots, e_n]$.

We formulate all of the Krylov methods described in this thesis to be correct either in real or in complex arithmetic, using a single notation for both cases. We support this with the following notation:

- For a scalar α, $\overline{\alpha}$ denotes its complex conjugate if α is complex, else $\overline{\alpha} = \alpha$ if α is real.
- If a scalar α is complex, $\Re(\alpha)$ denotes its real part and $\Im(\alpha)$ its imaginary part. If α is real, $\Re(\alpha) = \alpha$ and $\Im(\alpha) = 0$.
- For a column vector x of length n, x^* denotes its transpose ($x^* = x^T$) if x is real, and its complex conjugate transpose ($x^* = \overline{x}^T$) if x is complex.
- Similarly, for an $m \times n$ matrix A, A^* denotes its transpose ($A^* = A^T$) if A is real, and its complex conjugate transpose ($A^* = \overline{A}^T$) if A is complex.
- For two vectors x and y of the same length n, we define their (Euclidean) inner product as $\langle x, y \rangle = y^* x$, whether the vectors are real or complex.

This notation can obscure the difference between complex Hermitian matrices (for which $A^* = A$ always, but $A^T = A$ only if $\Im(A) = 0$) and real symmetric matrices (for which $A^T = A$). Any real symmetric matrix is also Hermitian, but some people object to calling real matrices Hermitian. We avoid awkward expressions like "the symmetric resp. Hermitian matrix A" by using the term "symmetric" to describe both real symmetric matrices, and complex Hermitian matrices. We do so even though our algorithms and mathematical derivations assume complex arithmetic. Similarly, we say "symmetric positive definite" (SPD) of both real symmetric positive definite matrices, and complex Hermitian positive definite (HPD) matrices.

Given an $n \times n$ SPD (see above discussion about SPD vs. HPD) matrix M, the operation $\langle x, y \rangle_M = y^* M x$ defines an inner product, which we call the M *inner product*. If $M = I$ (the $n \times n$ identity matrix), then the M inner product is the same as the usual Euclidean inner product $\langle x, y \rangle = y^* x$. We say analogously that two length n vectors x and y are M *orthogonal* ($x \perp_M y$), when $\langle x, y \rangle_M = 0$. This reduces to the usual (Euclidean) orthogonality, $x \perp y$, when $M = I$.

Computer performance model

Section 1.2.2 describes our performance model for parallel computers and computer memory hierarchies. In this model, we denote by P the number of processors in a parallel computer. When modeling two levels of a memory hierarchy, "slow" and "fast" memory (e.g., main memory and cache), we denote by W the capacity (in words) of the fast memory.

List of terms

A-conjugate vectors Vectors which are orthogonal ("conjugate") with respect to the A inner product, for a symmetric positive definite (SPD, which see) $n \times n$ matrix A. The length n vectors x and y are A conjugate when $y^*Ax = 0$. This is the origin of the word "conjugate" in the "Method of Conjugate Gradients" (CG, which see).

ADR *Arnoldi with Delayed Reorthogonalization*, a Krylov subspace method by Hernandez et al. [128] for solving nonsymmetric eigenvalue problems. It uses Classical Gram-Schmidt orthogonalization (CGS, which see) to orthogonalize the Krylov basis vectors. This differs from the standard version of Arnoldi iteration, which uses Modified Gram-Schmidt orthogonalization (MGS, which see).

Arnoldi iteration A Krylov subspace method of Arnoldi [8] for solving nonsymmetric eigenvalue problems; see also Bai et al. [16]. Described in Section 3.1.

Arnoldi(s, t) Same as CA-Arnoldi (Section 3.3), a communication-avoiding Krylov subspace method of ours for solving nonsymmetric eigenvalue problems.

AXPY *Alpha X Plus Y*, a computation $y := \alpha x + y$, involving two dense length n vectors x and y, and a scalar value α. The AXPY operation is included in the BLAS interface (which see). Often, AXPY is called by the BLAS interface name specific to the data types of the elements of x and y, such as SAXPY ("single-precision" real floating-point data). Other names in use include DAXPY (double-precision real data), CAXPY (single-precision complex data), or ZAXPY (double-precision complex data). In this thesis, we use the generic term AXPY.

B2GS A Block Gram-Schmidt algorithm with reorthogonalization by Jalby and Philippe [143], discussed in this thesis in Section 2.4.

Balancing A technique for improving the accuracy of solving nonsymmetric eigenvalue problems, by scaling the rows and columns of the matrix. Discussed in Section 7.5.3 of this thesis.

BiCG The *Method of Biconjugate Gradients* of Fletcher [99], a Krylov subspace method for solving nonsymmetric linear systems. Discussed in Section 6.2.

BiCGSTAB (the "Method of Biconjugate Gradients, Stabilized" of van der Vorst [226], a Krylov subspace method for solving nonsymmetric linear systems.

BLAS The *Basic Linear Algebra Subprograms*, an interface for dense linear algebra computations, such as dot products, vector sums, matrix-vector multiply, and matrix-matrix multiply. See e.g., "AXPY," "BLAS 1," "BLAS 2", and "BLAS 3." References include Lawson et al. [162] and Dongarra et al. [88, 87, 86, 85].

BLAS 1 The Basic Linear Algebra Subprograms (BLAS, which see) are organized into three "levels": BLAS 1, BLAS 2, and BLAS 3. The BLAS 1 consist of dense vector-vector operations such as dot products and AXPYs (which see). The BLAS 2 consist of dense matrix-vector operations such as matrix-vector products and triangular solves (with a single right-hand side vector). The BLAS 3 consist of dense matrix-matrix operations such as matrix-matrix products and triangular solves (with multiple right-hand side vectors). BLAS 3 operations have a higher ratio of floating-point arithmetic operations to memory operations than do BLAS 1 or BLAS 2 operations.

BLAS 2 Dense matrix-vector operations. See BLAS 1.

BLAS 3 Dense matrix-matrix operations. See BLAS 1.

BGS *Block Gram-Schmidt*, a class of algorithms related to Gram-Schmidt orthogonalization (see entries "CGS" and "MGS"). BGS algorithms process blocks of consecutive vectors at a time. This results in the algorithm spending more of its time in matrix-matrix products, which have a higher efficiency on modern processors than vector-vector operations (such as dot products and AXPYs (which see)).

Block CGS *Block Classical Gram-Schmidt*, a Block Gram-Schmidt algorithm (which see) patterned from the standard Classical Gram-Schmidt orthogonalization method (which see). Described in Section 2.4.

Block MGS *Block Modified Gram-Schmidt*, a Block Gram-Schmidt algorithm (which see) patterned from the standard Modified Gram-Schmidt orthogonalization method (which see). Described in Section 2.4.

CA-Arnoldi Our communication-avoiding version of Arnoldi iteration, a Krylov subspace method for solving nonsymmetric eigenvalue problems. Described in Section 3.3.

CA-CG Our communication-avoiding version of the Method of Conjugate Gradients (CG, which see), a Krylov subspace method for solving symmetric positive definite systems of linear equations. Described in Sections 5.4 (not preconditioned) and 5.5 (preconditioned).

CA-GMRES Our communication-avoiding version of GMRES (which see), a Krylov subspace method for solving nonsymmetric systems of linear equations. Described in Section 3.4.

CA-Lanczos Our communication-avoiding version of Lanczos iteration, a Krylov subspace method for solving symmetric eigenvalue problems. Described in Section 4.2.

CAQR *Communication-Avoiding QR*, an algorithm (or rather, a class of algorithms) for computing the QR factorization of a dense matrix, described in Demmel et al. [75].

CG The *Method of Conjugate Gradients* of Hestenes and Stiefel [131], a Krylov subspace method for solving symmetric positive definite linear systems. Described in Section 5.1.

CGNE The *Method of Conjugate Gradients on the Normal Equations*, a Krylov subspace method for solving linear least squares problems (see e.g., Saad [208]). It works by using the Method of Conjugate Gradients (CG, which see) on the normal equations.

CG3 The three-term recurrence variant of the Method of Conjugate Gradients (CG, which see), a Krylov subspace method for solving symmetric positive definite systems of linear equations. We use CG3 as an intermediate algorithm in order to develop CA-CG (which see). CG3 is described briefly in Section 5.4 and derived in detail in Appendix C.1.

CGS Depending on the context, this may mean either *Classical Gram-Schmidt*, an algorithm for orthogonalizing a finite collection of linearly independent elements of a vector space equipped with an inner product; or the *Conjugate Gradient Squared* algorithm of Sonneveld [214], a Krylov subspace method for solving nonsymmetric linear systems. Conjugate Gradient Squared is related to the Method of Biconjugate Gradients (BiCG) (which see).

CholeskyQR An algorithm for computing the QR decomposition $A = QR$ of an $m \times n$ matrix A with $m \geq n$. Described in e.g., Demmel et al. [75], where it is observed that CholeskyQR in finite-precision arithmetic, unlike TSQR (which see), may produce a Q whose columns are not at all orthogonal.

Clovertown The code name of a particular shared-memory parallel processor made by Intel.

Computational intensity For a particular algorithm, the ratio of floating-point arithmetic instructions to memory loads and stores.

CR The *Method of Conjugate Residuals*, a variant of the Method of Conjugate Gradients (CG, which see).

GMRES(s, t) Same as CA-GMRES (Section 3.4).

Gram matrix Let $V = [v_1, v_2, \ldots, v_p]$ be a collection of p length-n vectors (usually with $n \gg p$), and let $W = [w_1, w_2, \ldots, w_q]$ be a collection of q length-n vectors (usually with $n \gg q$). The *Gram matrix* of V and W is W^*V. It is used both for orthogonalization (when $W = V$, in which case this algorithm is called "CholeskyQR," which see) and for biorthogonalization. The Gram matrix may be computed using a single matrix-matrix multiply, which means that it requires less communication than (bi)orthogonalization methods that work one column at a time.

CPU *Central Processing Unit*, also called *processor*. The "brain" of a computer, that carries out the instructions in a program. These instructions include arithmetic or bitwise operations, as well as instructions to fetch or store data from resp. to storage units or input / output units.

Doubly stochastic A matrix A is *doubly stochastic* if all the entries of A are nonnegative, and each row and column of A sums to one.

DRAM *Dynamic random access memory*, a random-access memory technology commonly used to implement the main memory of a processor. Often used as a synonym for "main memory."

Equilibration A technique for scaling the rows and columns of a matrix in order to make the diagonal entries one in absolute value, and the off-diagonal entries smaller than one in absolute value. Discussed in Section 7.5.3.

Fast memory for two levels of a memory hierarchy, "fast memory" is the level with lower access latency but smaller capacity than "slow memory."

FOM The *Full Orthogonalization Method* (described in Saad [208]), a Krylov subspace method for solving nonsymmetric linear systems.

GMRES The *Generalized Minimum Residual* method of Saad and Schultz [209], a Krylov subspace method for solving nonsymmetric linear systems.

Golub-Kahan-Lanczos method Also called *Golub-Kahan-Lanczos bidiagonalization*. A Krylov subspace method of Golub and Kahan [113] for solving sparse nonsymmetric singular value problems. For the matrix A, the method is mathematically equivalent to using symmetric Lanczos iteration (which see) to solve the symmetric eigenvalue problem $(A^*A)x = \lambda x$. See also Bai et al. [16].

Householder QR An algorithm for computing the QR factorization of a matrix. It involves processing the matrix column by column, computing an orthogonal transform called a *Householder reflector* for each column. For a description of the algorithm, see e.g., Golub and Van Loan [114].

HPD *Hermitian positive definite*, said of a square matrix A which is both Hermitian ($A = A^*$, i.e., A equals its complex conjugate transpose) and positive definite. See the "SPD" entry.

HSS *Hierarchically semiseparable*, which refers either to a particular class of matrices, or a particular way of storing matrices in this class. HSS matrices have a recursive structure which asymptotically reduces the number of floating-point operations required for various linear algebra operations (such as factorizations, solving linear systems, and matrix-vector multiplies).

ILU *Incomplete LU factorization*, an approximate LU factorization $A \approx LU$ of a sparse nonsymmetric matrix A, that tries to keep the L and U factors as sparse as possible (in order to reduce computational cost and memory usage). ILU is often used as a preconditioner when solving a sparse linear system $Ax = b$ using a Krylov subspace method.

KSM *Krylov subspace method*, a class of iterative algorithms for solving sparse linear systems and eigenvalue problems. KSM is a nonstandard abbreviation which we use to save space.

LAPACK *Linear Algebra PACKage*, a software package written in a subset of Fortran 90, that "provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems" [6].

Laplacian matrix The *Laplacian matrix* A of a graph (without self-loops or multiple edges) is a (possibly sparse) matrix which has a nonzero entry A_{ij} for every edge e_{ij} between two vertices v_i and v_j in the graph. A_{ij} is the degree of v_i if $i = j$, $A_{ij} = -1$ if $i \neq j$ and edge e_{ij} exists, and $A_{ij} = 0$ otherwise.

Leja ordering A particular ordering on a set of complex numbers, which we describe in detail in Section 7.3.2. It is useful for the Newton s-step basis.

LINPACK "a collection of Fortran subroutines that analyze and solve [dense] linear equations and linear least-squares problems" [84]. A predecessor of LAPACK (which see). Also, the origin of the "LINPACK Benchmark," used to assess the performance of computer systems by solving a large square linear system $Ax = b$ using Gaussian elimination with partial pivoting.

LP-CA-CG The left-preconditioned variant of our communication-avoiding version of the Method of Conjugate Gradients (CG, which see), a Krylov subspace method for solving symmetric positive definite systems of linear equations. Described in Section 5.5.

LP-CG3 The left-preconditioned version of CG3 (which see), a Krylov subspace method for solving symmetric positive definite systems of linear equations. Described in Section 5.5.

Lucky breakdown A breakdown condition in Krylov subspace methods, in which the Krylov subspace has reached its maximum dimension given the matrix and starting vector. It is "lucky" both because it is rare, and because it gives useful information about the Krylov subspace.

MGS *Modified Gram-Schmidt*, an algorithm for orthogonalizing a finite collection of linearly independent elements of a vector space equipped with an inner product.

M **inner product** For an $n \times n$ symmetric positive definite (SPD, which see) matrix M, the inner product on vectors of length n defined by $\langle x, y \rangle_M = y^* M x$. (Note that we assume x and y are complex, which is why we reverse their order when writing their inner product. See the SPD entry in this glossary.) If M is the identity matrix I, the M inner product reduces to the usual Euclidean inner product $\langle x, y \rangle = y^* x$.

MINRES The *Minimum Residual* algorithm of Paige and Saunders [187], a Krylov subspace method for solving symmetric indefinite linear systems.

MPI *Message Passing Interface*, a specification of a interface for distributed-memory parallel programming.

Nehalem The code name of a particular shared-memory parallel processor made by Intel.

Nonsymmetric Lanczos iteration A Krylov subspace method of Lanczos [161] for solving nonsymmetric eigenvalue problems; see also Bai et al. [16]. Discussed in Section 6.1.

NUMA *Nonuniform Memory Access*, a design for a shared-memory parallel system, in which different parts of memory have different latencies (and possibly also different bandwidths) to different processors in the system. The opposite of NUMA is "UMA" (Uniform Memory Access), a less frequently used acronym. The advantages of a NUMA design are lower-latency access of a processor to its "local" memory, and usually also higher bandwidth (since a single "local" memory need not serve as many processors).

OOC *Out-of-core*, said of a computer program which can operate on data sets larger than can fit in main memory, by explicitly swapping subsets of the data between main memory and a slower storage device of larger capacity (such as a hard disk). "Core" is an anachronism referring to magnetic solenoid core memory technology, which became popular in the early 1960s as a main memory store.

Orthomin A Krylov subspace method of Vinsome [232] for solving symmetric linear systems. For a discussion, see e.g., Greenbaum [120, pp. 33–38].

OSKI *Optimized Sparse Kernel Interface*, an interface and optimized implementation by Vuduc et al. [237] of sparse matrix computations such as sparse matrix-vector multiply (SpMV, which see) and sparse triangular solve (SpTS, which see), along with optimized implementations of those computations for superscalar cache-based processors.

PDE *Partial differential equation*, a differential equation involving functions of multiple variables. Examples include the heat equation, the wave equation, and Poisson's equation. PDEs are often solved via discretization, which results in a finite system of linear equations. Such linear systems are themselves often solved using Krylov subspace methods, especially if the discretization results in a sparse matrix or a matrix with a structure amenable to performing matrix-vector products inexpensively.

Power method A simple iteration for approximating the principal eigenvector and eigenvalue of a matrix A, by means of repeated sparse matrix-vector multiplications.

PRAM *Parallel Random-Access Machine*, an abstract model of a shared-memory parallel computer with some number P processors. It assumes that synchronization requires no time, that the processors do not contend for resources, and that each processor can access any particular memory location in one cycle.

QMR The *Quasi-Minimal Residual* algorithm of Freund and Nachtigal [103], a Krylov subspace method for solving nonsymmetric linear systems.

RODDEC A QR factorization algorithm for parallel processors on a ring network, described by Erhel [94] and discussed e.g., in Demmel et al. [75]. On a system with P processors, RODDEC requires $\Theta(P)$ messages, which is asymptotically greater than the lower bound of $\Theta(\log P)$ messages. Our parallel QR factorization TSQR ("Tall Skinny QR," see Section 2.3) meets that lower bound.

RR-TSQR Our "Rank-Revealing Tall Skinny QR" factorization, described in Section 2.4.

RR-TSQR-BGS Our "Rank-Revealing Tall Skinny QR - based Block Gram-Schmidt" orthogonalization method, described in Section 2.4.

ScaLAPACK As described by Blackford et al. [33], "ScaLAPACK is a library of high-performance linear algebra routines for distributed-memory message-passing MIMD computers and networks of workstations supporting PVM and / or MPI. It is a continuation of the LAPACK project, which designed and produced analogous software for workstations, vector supercomputers, and shared-memory parallel computers... The name ScaLAPACK is an acronym for Scalable Linear Algebra PACKage, or Scalable LAPACK."

Serious breakdown A particular breakdown condition in nonsymmetric Lanczos iteration (which see) and BiCG (which see), discussed in various places in Chapter 6.

Slow memory For two levels of a memory hierarchy, "slow memory" is the level with higher access latency but larger capacity than "fast memory."

SPAI *Sparse Approximate Inverse*, a class of preconditioners which construct an approximate inverse B of a sparse matrix A, in which the sparsity pattern of B is constrained in some way.

SPD *Symmetric positive definite*, said of a square matrix A. In this thesis, we use SPD to describe both real symmetric positive definite matrices and complex Hermitian positive definite matrices, but we assume complex arithmetic in our algorithms and mathematical derivations. Such notation is nonstandard, but we do so to avoid awkward "SPD resp. HPD" constructions.

SpMV *Sparse matrix-vector multiply*, the product $y := A \cdot x$, where x and y are dense vectors and A is a sparse matrix.

SpMM *Sparse matrix times multiple vectors*, the product $Y := A \cdot X$, where X and Y are dense matrices and A is a sparse matrix. (See the "SpMV" entry.) Usually it is assumed that X and Y have many more rows than columns.

SpTS *Sparse triangular solve*, the computation of the solution y of $Ay = x$, where A is a sparse (upper or lower) triangular matrix, and x and y are dense vectors.

SpTM *Sparse triangular solve with multiple vectors*, the computation of the solution Y of $AY = X$, where A is a sparse (upper or lower) triangular matrix, and X and Y are dense matrices. (See the "SpTS" entry.) Usually it is assumed that X and Y have many more rows than columns.

SVD *Singular Value Decomposition*, a factorization $A = U\Sigma V$ of a general real or complex matrix A. See e.g,. Golub and Van Loan [114] or Demmel [74].

Symmetric Lanczos iteration A Krylov subspace method of Lanczos [161] for solving symmetric eigenvalue problems; see also Bai et al. [16]. Described in Section 4.1.

TFQMR The *Transpose-Free QMR* algorithm of Freund [102], a Krylov subspace method for solving nonsymmetric linear systems.

TSP *Traveling Salesperson Problem*, a problem in combinatorial optimization, namely, to find the shortest path between a set of vertices of a graph, that visits each vertex exactly once. The Traveling Salesperson Problem is known to be NP-complete, which in practice means that solving large TSP problems is computationally expensive (even prohibitively so). We refer to the Traveling Salesperson Problem in Section 2.1.

TSQR *Tall Skinny QR*, an algorithm (or rather, a collection of related algorithms) for computing the QR factorization of an $m \times n$ matrix A with $m \gg n$ (so that the shape of A is "tall and skinny"). See Section 2.3.

Two-sided Arnoldi iteration A Krylov subspace method of Ruhe [205] for solving nonsymmetric eigenvalue problems. This differs from standard Arnoldi iteration (which see), which is a one-sided algorithm.

WATT A collection of real nonsymmetric sparse matrix test problems from petroleum engineering applications. They belong to the Harwell-Boeing sparse matrix test suite (see e.g., Boisvert et al. [35]).

Abstract

Communication-Avoiding Krylov Subspace Methods

by

Mark Hoemmen

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor James W. Demmel, chair

The cost of an algorithm includes both arithmetic and communication. We use "communication" in a general sense to mean data movement, either between levels of a memory hierarchy ("sequential") or between processors ("parallel"). Communication costs include both bandwidth terms, which are proportional to the number of words sent, and latency terms, which are proportional to the number of messages in which the data is sent. Communication costs are much higher than arithmetic costs, and the gap is increasing rapidly for technological reasons. This suggests that for best performance, algorithms should minimize communication, even if that may require some redundant arithmetic operations. We call such algorithms "communication-avoiding."

Krylov subspace methods (KSMs) are iterative algorithms for solving large, sparse linear systems and eigenvalue problems. Current KSMs rely on sparse matrix-vector multiply (SpMV) and vector-vector operations (like dot products and vector sums). All of these operations are communication-bound. Furthermore, data dependencies between them mean that only a small amount of that communication can be hidden. Many important scientific and engineering computations spend much of their time in Krylov methods, so the performance of many codes could be improved by introducing KSMs that communicate less.

Our goal is to take s steps of a KSM for the same communication cost as 1 step, which would be optimal. We call the resulting KSMs "communication-avoiding Krylov methods." We can do this under certain assumptions on the matrix, and and for certain KSMs, both in theory (for many KSMs) and in practice (for some KSMs). Our algorithms are based on the so-called "s-step" Krylov methods, which break up the data dependency between the sparse matrix-vector multiply and the dot products in standard Krylov methods. This idea has been around for a while, but in contrast to prior work (discussed in detail in Section 1.5), we can do the following:

- We have fast kernels replacing SpMV, that can compute the results of s calls to SpMV for the same communication cost as one call (Section 2.1).
- We have fast dense kernels as well, such as Tall Skinny QR (TSQR – Section 2.3) and Block Gram-Schmidt (BGS – Section 2.4), which can do the work of Modified Gram-Schmidt applied to s vectors for a factor of $\Theta(s^2)$ fewer messages in parallel, and a factor of $\Theta(s/W)$ fewer words transferred between levels of the memory hierarchy (where W is the fast memory capacity in words).

- We have new communication-avoiding Block Gram-Schmidt algorithms for orthogonalization in more general inner products (Section 2.5).

- We have new communication-avoiding versions of the following Krylov subspace methods for solving linear systems: the Generalized Minimum Residual method (GMRES – Section 3.4), both unpreconditioned and preconditioned, and the Method of Conjugate Gradients (CG), both unpreconditioned (Section 5.4) and left-preconditioned (Section 5.5).

- We have new communication-avoiding versions of the following Krylov subspace methods for solving eigenvalue problems, both standard ($Ax = \lambda x$, for a nonsingular matrix A) and "generalized" ($Ax = \lambda Mx$, for nonsingular matrices A and M): Arnoldi iteration (Section 3.3), and Lanczos iteration, both for $Ax = \lambda x$ (Section 4.2) and $Ax = \lambda Mx$ (Section 4.3).

- We propose techniques for developing communication-avoiding versions of nonsymmetric Lanczos iteration (for solving nonsymmetric eigenvalue problems $Ax = \lambda x$) and the Method of Biconjugate Gradients (BiCG) for solving linear systems. See Chapter 6 for details.

- We can combine more stable numerical formulations, that use different bases of Krylov subspaces, with our techniques for avoiding communication. For a discussion of different bases, see Chapter 7. To see an example of how the choice of basis affects the formulation of the Krylov method, see Section 3.2.2.

- We have faster numerical formulations. For example, in our communication-avoiding version of GMRES, CA-GMRES (see Section 3.4), we can pick the restart length r independently of the s-step basis length s. Experiments in Section 3.5.5 show that this ability improves numerical stability. We show in Section 3.6.3 that it also improves performance in practice, resulting in a 2.23× speedup in the CA-GMRES implementation described below.

- We combine all of these numerical and performance techniques in a shared-memory parallel implementation of our communication-avoiding version of GMRES, CA-GMRES. Compared to a similarly highly optimized version of standard GMRES, when both are running in parallel on 8 cores of an Intel Clovertown (see Appendix A), CA-GMRES achieves 4.3× speedups over standard GMRES on standard sparse test matrices (described in Appendix B.5). When both are running in parallel on 8 cores of an Intel Nehalem (see Appendix A), CA-GMRES achieves 4.1× speedups. See Section 3.6 for performance results and Section 3.5 for corresponding numerical experiments. We first reported performance results for this implementation on the Intel Clovertown platform in Demmel et al. [79].

- Incorporation of preconditioning. Note that we have not yet developed practical communication-avoiding preconditioners; this is future work. We *have* accomplished the following:

  - We show (in Sections 2.2 and 4.3) what the s-step basis should compute in the preconditioned case for many different types of Krylov methods and s-step bases. We explain why this is hard in Section 4.3.
  - We have identified two different structures that a preconditioner may have, in order to achieve the desired optimal reduction of communication by a factor of s. See Section 2.2 for details.

- We present a detailed survey of related work, including s-step KSMs (Section 1.5, especially Table 1.1) and other techniques for reducing the amount of communication in iterative methods (Section 1.6).

Chapter 1

Introduction

The cost of an algorithm includes both arithmetic and communication. We use "communication" in a general sense to mean data movement, either between levels of a memory hierarchy ("sequential") or between multiple processors ("parallel"). Communication costs are much higher than arithmetic costs, and the gap is increasing rapidly for technological reasons. This suggests that for best performance, algorithms should minimize communication, even if that may require some redundant arithmetic operations. We call such algorithms "communication-avoiding."

Krylov subspace methods (KSMs) are iterative algorithms for solving large, sparse linear systems and eigenvalue problems. Current KSMs spend most of their time in communication-bound computations, like sparse matrix-vector multiply (SpMV) and vector-vector operations (including dot products and vector sums). Furthermore, data dependencies between these operations mean that only a small amount of that communication can be hidden. Many important scientific and engineering computations spend much of their time in Krylov methods, so the performance of many codes could be improved by introducing KSMs that communicate less.

Our goal is to take s steps of a KSM for the same communication cost as 1 step, which would be optimal. We call the resulting KSMs "communication-avoiding Krylov methods." We can do this under certain assumptions on the matrix, and and for certain KSMs, both in theory (for many KSMs) and in practice (for some KSMs). This thesis explains how we avoid communication in these Krylov methods, and how we maintain their accuracy in finite-precision arithmetic.

This first chapter is organized as follows.

- Section 1.1 defines Krylov subspace methods. It shows their basic computational building blocks, called "kernels," and the data dependencies between these kernels in many types of Krylov methods.

- Section 1.2 explains in more detail what we mean by "communication." It gives our basic computer performance models, one for parallel computers and one for the memory hierarchy, which incorporate both the cost of arithmetic and communication. We also justify our assertion that communication is expensive relative to arithmetic, by examining historical and future trends in computer hardare. Finally, we explain what

we mean by "avoiding communication," to differentiate it from similar-sounding expressions such as "hiding communication."

- Section 1.3 lists the computational kernels found in many different Krylov methods in common use today. These kernels include sparse matrix-vector multiply (SpMV), preconditioning, and vector-vector operations such as dot products and "AXPY" operations (weighted vector sums of the form $y := y + \alpha x$, for vectors x and y and a scalar α). We give arguments that all these kernels have communication-bound performance.

- Section 1.4 gives new communication-avoiding kernels, which perform the same work as the above kernels, for a much lower communication cost. These kernels include the matrix powers kernel, the Tall Skinny QR factorization (TSQR), and Block Gram-Schmidt (BGS). We will explain algorithms and implementations of these kernels in Chapter 2.

- Section 1.5 is a comprehensive survey of the so-called "s-step Krylov methods," to which our communication-avoiding Krylov methods are related. As far as we can tell, this is the first such comprehensive survey of related work to be done for the s-step methods.

- Section 1.6 discusses other iterative algorithms for solving linear systems and eigenvalue problems, that have as their aim to communicate less. While these algorithms all have value in the right context, we explain why we chose not to investigate them further in this thesis.

- Finally, Section 1.7 summarizes our contributions, and the contents of the remaining chapters of this thesis.

1.1 Krylov subspace methods

1.1.1 What are Krylov methods?

Krylov subspace methods (KSMs) are a large class of iterative algorithms that work on finite matrices and vectors in real and complex arithmetic.[1] They have many applications, which include solving linear systems, linear least squares problems, eigenvalue problems, and singular value problems. In this work, we discuss the solution of linear systems $Ax = b$ and eigenvalue problems $Ax = \lambda x$. We will present new "communication-avoiding" Krylov methods that solve both these problems more efficiently than existing KSMs, both in performance models and in actual implementations. Later in this Introduction, we will explain what "communication" means in the context of Krylov methods, and what it means for a KSM to be "communication-avoiding."

A number of books give an overview of KSMs. The "Templates for the Solution..." pair of books [23, 16] describe Krylov methods in the form of a decision tree for picking which method to use. One book covers iterations for solving linear systems, and the other book

[1] Alexei Nikolaevich Krylov (1863–1945), a Russian Naval engineer and applied mathematician, developed the class of methods which bear his name in a 1931 paper on solving eigenvalue problems [159].

iterations for solving eigenvalue problems. Saad [208] gives overviews of KSMs for solving $Ax = b$, and summarizes known results on their convergence properties in exact arithmetic. Greenbaum [120] does both of these, and also summarizes known results on their convergence properties in finite-precision arithmetic.

KSMs work by using one or more matrix-vector products in each iteration, to add vector(s) to a basis for one or more so-called "Krylov subspace(s)" with desirable properties. A *Krylov subspace* with respect to a square $n \times n$ matrix A and a length n vector v is the following subspace:

$$\mathcal{K}_k(A, v) = \text{span}\{v, Av, A^2v, \ldots, A^{k-1}v\}, \tag{1.1}$$

where k is a positive integer. KSMs use one or more of these Krylov subspaces. Each iteration of a KSM is supposed to increase the dimension of these subspace(s). They are used as expanding search space(s), from which the approximate solution x_k (if solving $Ax = b$) or the approximate eigenvalues and eigenvectors (if solving $Ax = \lambda x$) are computed via projection. For a classification of Krylov methods by the subspace(s) they construct and how they project onto them, see e.g., Saad [208, Chapter 5].

Krylov methods for solving linear systems $Ax = b$ may also be *preconditioned*. Preconditioned KSMs solve a different linear system, but with the same solution, with the goal of converging in a smaller number of iterations. *Left preconditioning* means solving $M^{-1}Ax = M^{-1}b$ instead of $Ax = b$. Here, M is a nonsingular matrix meant to be an approximation of A in some sense, constructed so that linear systems $My = z$ are relatively inexpensive to solve. *Right preconditioning* means solving $AM^{-1}y = b$ and $Mx = y$, instead of $Ax = b$. *Split preconditioning* means solving $M_L^{-1}AM_R^{-1}y = M_L^{-1}b$ and $M_Rx = y$, instead of $Ax = b$. In this case, M_L and M_R are nonsingular matrices whose product $M_L \cdot M_R$ is meant to be an approximation of A in some sense, such that linear systems $M_Lz = w$ and $M_Rz = w$ are relatively inexpensive to solve. Explaining what makes a good preconditioner is beyond the scope of this thesis. However, preconditioning will affect how we construct our new iterative methods; see e.g., Section 4.3.

Krylov methods and SpMV

Krylov methods have the property that they only access the matrix A in order to perform matrix-vector products with A (and possibly also with the transpose or conjugate transpose of A). They do not change the entries of the matrix, unlike other linear algebra algorithms such as LU or QR factorization. This makes KSMs especially useful in the following cases, which may overlap somewhat:

1. The matrix A is available only as an operator which performs matrix-vector products $v \mapsto A \cdot v$ (and possibly also $v \mapsto A^* \cdot v$).[2]

[2]Here and elsewhere in this work, A^* means the transpose of A if A is real, and the conjugate transpose of A if A is complex. We will use the notation of complex arithmetic; for instance, we will write $A^* = \overline{A}^T$ instead of A^T and $\langle x, y \rangle = y^* \cdot x = \overline{y}^T \cdot x$ instead of $x^T \cdot y$. However, we will use the language of real arithmetic, for example, "the matrix A is symmetric" rather than "the matrix A is Hermitian." We do the latter in order to avoid awkward expressions such as "symmetric resp. Hermitian," and because the conjugate transpose of a real matrix or vector is just the ordinary transpose of that matrix resp. vector.

2. The matrix A is stored using a data structure that admits much more efficient computation of matrix-vector products than if A had been stored as an explicit $n \times n$ matrix.

An example of the "operator only" case is if the $n \times n$ matrix A is not stored at all, rather it is implemented as a routine which maps the discretization f of the forcing term of a partial differential equation, to the solution $A \cdot f$ of the discretization of that equation. The routine implicitly represents a linear, finite operator A, but the entries of A are not stored. As we will explain in Sections 2.1 and 2.2, we cannot avoid communication in the matrix-vector products part of Krylov methods without some access to the internal representation of the matrix A. In the case of preconditioned KSMs, we cannot avoid communication in the preconditioner application in general, without analogous access to the internal representation of the preconditioner. However, some of our new KSMs will still be able to avoid communication in the vector operations, even without access to the representation of the matrix (and preconditioner).

The second case above, where A has a representation that makes matrix-vector products efficient, includes many different data structures. By "efficient" we mean that A requires much less storage than a dense $n \times n$ matrix would, and matrix-vector products with A require many fewer arithmetic operations than ordinary dense matrix-vector products would. One class of data structures in this category exploit the structure of *sparse matrices*, which contain many more zero entries than nonzero entries. These sparse data structures store only the nonzero entries, and their indices in the matrix. For an overview of different sparse matrix storage formats, see e.g., Saad [208].

There are other matrix structures that make matrix-vector products efficient. They do not exploit sparsity in the matrix, but rather other kinds of structure that enable efficient representations. Matrices of these type include *hierarchical matrices* and *hierarchically semiseparable* matrices. We discuss these further in Section 2.2, where we will use them as a model for avoiding communication in preconditioned Krylov methods.

When A is an $n \times n$ sparse matrix and v is a length n vector, we call the operation $v \mapsto A \cdot v$ *sparse matrix-vector multiply* (SpMV). For an overview of how to implement SpMV efficiently with different sparse matrix formats, see e.g., Vuduc [235]. We use the term "SpMV" more generally throughout this work to refer to the $v \mapsto A \cdot v$ operation, when the operation is implemented to save both storage and computation over a dense representation of A.

1.1.2 Kernels in Krylov methods

SpMV is an important, time-consuming part of all Krylov methods, but not the only such part. We call these important, time-consuming parts of Krylov methods "kernels." We will identify kernels in commonly used Krylov methods in Section 1.1.3. The reason we do so is to demonstrate that these kernels' performance is communication bound, and that new Krylov methods constructed out of new kernels can avoid much of this communication.

Before we identify kernels in commonly used Krylov methods, we need to clarify our use of the word "kernel." This word refers to several different concepts in mathematics and computer science. In this thesis, unless we say otherwise, we do *not* use the word in any of

the following senses:

- The kernel of an integral equation or a convolution. For example, κ is the kernel of the integral equation $\int_\Omega \kappa(x,y)\phi(y)\,dy = f(x)$.
- The null space of a linear operator: that is, the set of all inputs for which the output is zero.
- The central component of a computer's operating system: e.g., "the Linux kernel."
- For a star-shaped polygon, the set of all points p inside the polygon, which can "see" all boundary points of the polygon.

Rather, by a *kernel* we mean a building block of a numerical algorithm, that consumes a significant part of the run time of that algorithm. For example, dense matrix-matrix multiply is an important kernel in efficient implementations of dense LU factorization. (The software package LAPACK [5]implements such an LU factorization as `DGETRF`, and it invokes dense matrix-matrix multiply via the DGEMM interface specified by the BLAS 3 [86, 85] set of interfaces.)

What constitutes a "kernel" depends on one's point of view. For example, from the perspective of LAPACK's LU factorization routine `DGETRF`, the standard column-by-column LU factorization routine `DGETF2` is a kernel [5]. From the perspective of `DGETF2`, scaling a dense vector and applying a rank-one update to a dense matrix are kernels. When solving linear optimization problems, a higher-level LU factorization routine such as `DGETRF` may itself be considered a kernel.

A kernel specifies only an interface, not an implementation. One defines a kernel in order to expose particular features of the computation that can be exploited for greater computational efficiency. For example, dense matrix-matrix multiply of two $n \times n$ matrices has the feature that there are $\Theta(n^2)$ data to read, but $\Theta(n^3)$ arithmetic computations.[3] Implementations of the kernel are then free to exploit those features internally to achieve better performance. Different implementations of the same kernel should produce the same results in exact arithmetic, and nearly the same results in floating-point arithmetic. Sometimes in this thesis, we use the same name both for a kernel, and for the algorithm(s) or implementation strategy/ies we use for that kernel. We do so merely to avoid defining terms superfluously. The Tall Skinny QR (TSQR) factorization, which we present in Section 2.3, is one example. "Tall and skinny" refer to the dimensions of the input matrix A of the QR factorization: it should have many more rows than columns (if A is $m \times n$, we require $m \gg n$). Requiring that the input matrix have this property exposes potential optimizations, which we describe in Section 2.3.

Two kernels other than SpMV that will come up often in the following sections are AXPY and dot products. These are both kernels that take two vectors x and y of the same length n as inputs. *AXPY* ("α times x plus y") computes

$$y := y + \alpha \cdot x$$

[3] Not counting Strassen-like implementations.

where α is a scalar. We use AXPY to refer more generally to operations of the form

$$y := \beta \cdot y + \alpha \cdot x,$$

where β is another scalar constant. The *dot product* kernel computes the dot product of two vectors x and y of the same length n. The result is $\sum_{i=1}^{n} \overline{y}_i \cdot x_i$. Computing the two-norm of a vector x, that is $\|x\|_2$, is an operation with similar computational characteristics, which we consider as a variant of the dot product kernel. These two kernels are implemented, for example, in the Level 1 BLAS (see e.g., Lawson et al. [162]).

1.1.3 Structures of Krylov methods

In this section, we categorize standard Krylov methods by their structures. We do so for two reasons:

- to show that standard KSMs are composed of communication-bound kernels, whose communication costs cannot be entirely hidden by overlapping them, and
- to summarize the different approaches we will take in turning standard Krylov methods into communication-avoiding KSMs.

Describing the different categories of KSMs will help us determine both what our new communication-avoiding kernels (see Section 1.4) must compute, and which communication-avoiding kernels we may use in the KSM (see e.g., Section 4.3).

Data dependencies between kernels

In the previous Section 1.1.2, we mentioned four kernels that appear in commonly used Krylov methods:

- SpMVs (sparse matrix-vector products of the form $y := A \cdot x$ or $y := A^T \cdot x$)
- Preconditioning (applying a preconditioner M or M^{-1} to a vector x, resulting in a vector y)
- AXPYs (vector-vector operations of the form $y := \alpha \cdot x + y$)
- Dot products (vector-vector operations of the form $\alpha := y^* \cdot x$)

After we define clearly what we mean by communication in Section 1.2.1, we will show in Section 1.3 that these kernels' performance is almost always bound by the cost of communication. In this section, we will show that typical KSMs have a data dependency between the kernels mentioned above. The data dependency thus hinders hiding the kernels' communication costs by overlapping their execution. Thus, if the kernels are communication-bound, so is the Krylov method, which is composed of the sequential execution of one kernel after another.

Arnoldi iteration [8] for solving nonsymmetric eigenvalue problems is a good example of a Krylov subspace method. It shows the SpMV, dot product, and AXPY kernels "at

Algorithm 1 Arnoldi iteration

Input: $n \times n$ matrix A

Input: Length n starting vector v

Output: $s+1$ length-n unitary vectors $\underline{Q} = [Q, q_{s+1}] = [q_1, q_2, \ldots, q_{s+1}]$ which form a basis for $\mathcal{K}_{s+1}(A, r)$

Output: A nonsingular $s+1$ by s upper Hessenberg matrix $\underline{H}$ such that $AQ = \underline{Q}\underline{H}$

1: $\beta := \|v\|_2$, $q_1 := v/\beta$
2: **for** $j = 1$ to s **do**
3: $\quad w_j := Aq_j$
4: $\quad$ **for** $i = 1$ to j **do** ▷ Use Modified Gram-Schmidt (MGS) to orthogonalize w_j against $q_1, \ldots, q_j$
5: $\quad\quad h_{ij} := \langle w, q_i \rangle$
6: $\quad\quad w_j := w_j - h_{ij} q_i$
7: $\quad$ **end for**
8: $\quad h_{j+1,j} := \|w_j\|_2$
9: $\quad$ **if** $h_{j+1,j} = 0$ **then**
10: $\quad\quad$ Exit, since the Krylov subspace has dimension j ("lucky breakdown")
11: $\quad$ **end if**
12: $\quad q_{j+1} := w_j / h_{j+1,j}$
13: **end for**

work." We explain Arnoldi iteration in more detail in Section 3.1 and give the algorithm there as Algorithm 18. Here, we give an outline of Arnoldi as Algorithm 1. This outline neglects many important features of Krylov methods, such as preconditioning, or how the approximate solution (when solving $Ax = b$) or approximate eigenvalues (when solving $Ax = \lambda x$) are computed at each step. However, as we will show, it illustrates the data dependencies between the SpMV, dot product, and AXPY kernels. Not all KSMs have the same structure as Arnoldi iteration, as we will show in this section. However, almost all standard Krylov methods have analogous data dependencies between kernels.

In Arnoldi iteration (Algorithm 1), the vectors $q_1, q_2, \ldots, q_k$ form a basis for the Krylov subspace

$$\mathcal{K}_k(A, r) = \text{span}\{v, Av, A^2v, \ldots, A^{k-1}v\},$$

as long as the Krylov subspace $\mathcal{K}_k(A, r)$ indeed has dimension at least $k+1$. (The case where it does not results in a breakdown condition, called "lucky" breakdown; see Line 10 of Algorithm 1. This breakdown is "lucky" both because it gives valuable information about the matrix A, and because it is unlikely in practice unless the starting vector v is special. We will discuss lucky breakdown more in the context of each KSM where it matters in the algorithm.) Arnoldi iteration is for solving eigenvalue problems and thus is generally not preconditioned,[4] but other one-sided methods similar to Arnoldi for solving linear systems (such as GMRES, see Saad and Schultz [209]) may be preconditioned. If there is any preconditioning, it happens somewhere inside the loop, depending both on the type

[4]Except perhaps for *balancing*, which involves scaling the rows and columns of A in order to improve accuracy when solving eigenvalues. Balancing need not affect the computations "inside" of the Krylov method. For an explanation of balancing, see e.g., [16].

of preconditioner (left, right, or split), and which iterative method is used.

The part of Arnoldi that best illustrates the data dependencies between kernels is the computation of the "next" basis vector q_{j+1}, from the "current" basis vectors $q_1, \ldots, q_j$. The basis vector q_{j+1} is computed via a vector recurrence

$$q_{j+1} = h_{j+1,j}^{-1} \left(Aw_j - \sum_{i=1}^{j} h_{ij} q_i \right), \tag{1.2}$$

where the h_{ij} coefficients come from dot products involving Aw_j or intermediate vector terms. Different Krylov methods may compute different vector recurrences or compute them in a different order. Algorithm 1 computes them using the order specified by Modified Gram-Schmidt (MGS) orthogonalization. Some iterative methods may use the order specified by Classical Gram-Schmidt (CGS) orthogonalization, or some other order entirely. Nevertheless, each h_{ij} must be computed (via dot products) before its corresponding AXPY operation with the q_i vector can finish. This is true also for the analogous recurrences in other Krylov methods: there are AXPY operation(s) which cannot begin until the dot product operation(s) have finished, and the next SpMV depends on the result of that last AXPY operation. This data dependency limits how much communication cost can be hidden between these kernels via overlapping.

Categories of Krylov methods

We make assertions throughout this section about "standard Krylov methods." There are many different Krylov methods, even too many for us to summarize them all in a reasonable space. However, most of them fall into categories, with the methods in a particular category sharing a common structure that lets us make assertions about the data dependencies between their kernels. Also, this common structure means that we can use the same approach to develop communication-avoiding versions of the KSMs in the same category. This especially affects which kernels we may use in the communication-avoiding KSMs, for example whether we may use a QR factorization to orthogonalize the Krylov basis vectors in the method. It also affects the relative amount of time the KSM may spend in each kernel, and thus the amount of effort we should spend optimizing that kernel.

We construct these categories of Krylov methods by presenting four different criteria by which we classify commonly used KSMs. The criteria are:

1. "One-sided" vs. "two-sided"
2. Long recurrence vs. short recurrence
3. Mathematical structure (general, symmetric, positive definite, ...) of the matrix A
4. Mathematical structure (general, symmetric, positive definite, ...) of the preconditioner

One could imagine constructing all possible combinations of these criteria. We have only investigated the following four combinations. They seem to have the most affect on the structure of our communication-avoiding Krylov methods.

1. One-sided, long recurrence, general A, no or any preconditioner
2. One-sided, short recurrence, symmetric A, no preconditioner or symmetric split preconditioner
3. Two-sided, long or short recurrence, general A, no preconditioner

Note that the above criteria have nothing to do with whether the algorithm solves linear systems or eigenvalue problems. This distinction does not affect the basic performance characteristics of the algorithm. It may affect performance secondarily: for example, when solving eigenvalue problems, Krylov methods often require reorthogonalization, which may be expensive. However, such distinctions generally cut across the criteria: algorithms for solving eigenvalue problems in all three categories may require some form of reorthogonalization, for instance. This is why we let ourselves group together algorithms for solving $Ax = b$ and $Ax = \lambda x$ within the same category.

One-sided vs. two-sided

The Krylov methods we consider in this thesis are either one-sided, or two-sided. *One-sided* KSMs construct a basis of Krylov subspace(s) of the form $\text{span}\{v, Av, A^2v, \dots\}$. The vector is always on the right side of the matrix A in the matrix-vector products used to construct the basis. Often the basis vectors of this subspace are constructed to be orthogonal, with respect to some inner product. Examples of one-sided iterations include

- Arnoldi iteration (for solving nonsymmetric eigenvalue problems; see Section 3.1 and Arnoldi [8]; see also [16])
- GMRES (the "Generalized Minimum Residual" method of Saad and Schultz [209] for solving nonsymmetric linear systems; see Section 3.4)
- Symmetric Lanczos iteration (for solving symmetric eigenvalue problems; see Section 4.1 and Lanczos [161]; see also [16])
- CG (the "Method of Conjugate Gradients" of Hestenes and Stiefel [131] for solving symmetric positive definite linear systems; see Section 5.1)
- MINRES (the "Minimum Residual" method of Paige and Saunders [187] for solving symmetric indefinite linear systems)

Two-sided KSMs construct and use two different Krylov bases. One is for the "right-hand" space $\text{span}\{v, Av, A^2v, \dots\}$, and the other is for the "left-hand" space $\text{span}\{w, A^*w, (A^*)^2w, \dots\}$, where v and w are generally different vectors. "Right hand" means that the vector is multiplied on the right-hand side of the matrix, and "left hand" means that the vector is multiplied on the left-hand side of the matrix (or, the right-hand side of the transposed matrix: $A^*w = (w^*A)^*$). Some two-sided algorithms compute both transposed matrix-vector products $y \mapsto A^* \cdot y$ (or equivalently, $y^* \mapsto y^* \cdot A$) as well as non-transposed matrix-vector products $x \mapsto A \cdot x$. These include

- Nonsymmetric Lanczos (for solving nonsymmetric eigenvalue problems; see Section 6.1 and Lanczos [161]; see also [16])
- BiCG (the "Method of Biconjugate Gradients" of Fletcher [99] for solving nonsymmetric linear systems; see Section 6.2)
- QMR (the "Quasi-Minimal Residual" method of Freund and Nachtigal [103] for solving nonsymmetric linear systems)
- The so-called "Two-sided Arnoldi" iteration of Ruhe [205] (which differs from standard Arnoldi, a one-sided algorithm) for solving nonsymmetric eigenvalue problems

Other two-sided algorithms construct the left-hand space implicitly, that is without computing transposed matrix-vector products. Examples include

- CGS (the "Conjugate Gradient Squared" method of Sonneveld [214], for solving nonsymmetric linear systems)
- BiCGSTAB (the "Method of Biconjugate Gradients, Stabilized" of van der Vorst [226], for solving nonsymmetric linear systems)
- TFQMR (the "Transpose-Free QMR" method of Freund [102], for solving nonsymmetric linear systems)

The one-sided or two-sided distinction matters because the communication-avoiding Krylov methods we will construct in later chapters all work by computing several vectors of a Krylov subspace "all at once," using a kernel we call the "matrix powers kernel." (See Sections 2.1 and 2.2 for details on this kernel.) "Sidedness" changes what the matrix powers kernel must compute. In addition, some two-sided algorithms have breakdown conditions which may interact in not-yet-understood ways with numerical stability issues relating to the basis construction. Their additional complexity is why we elect not to study them in detail in this thesis.

We did not mention preconditioning in the discussion above. Later in this thesis, we will show that left-preconditioned one-sided Krylov methods which assume that the matrix A is symmetric (such as CG), and two-sided Krylov methods, have a similar structure. For a discussion of two-sided methods and of this similarity, see Chapter 6.

Long vs. short recurrence

As we mentioned above, Arnoldi iteration (Algorithm 1) computes the "next" Krylov subspace basis vector q_{j+1} at each iteration using the vector recurrence in Equation (1.2). Nearly all one-sided Krylov methods use a vector recurrence of similar form. Two-sided KSMs also use vector recurrences of analogous form. However, if we use the notation of Equation (1.2), some Krylov methods set all but a few of the coefficients in the recurrence to zero, and do not compute them or compute terms in the above sum that are multiplied by those zero coefficients. We say that such KSMs have a *short recurrence*. For example, in symmetric Lanczos, v_{j+1} depends only on Av_j, v_j, and v_{j+1}. Other Krylov methods assume that any of

the coefficients in Equation (1.2) may be nonzero, and compute them and their corresponding vector terms explicitly. For example, Arnoldi iteration uses MGS orthogonalization to orthogonalize Av_j against v_1, v_2, ..., v_j explicitly. We say that such iterative methods have a *long recurrence*.

This distinction matters for performance tuning, because it affects how much time a KSM spends in each kernel. Long-recurrence methods, such as Arnoldi or GMRES, tend to spend more time in inner products and AXPYs than in SpMV operations. This has influenced previous work in communication-avoiding versions of long-recurrence methods such as GMRES, for example, as we will discuss in Chapter 3. If the algorithm spends most of its time in AXPYs and dot products, rather than in SpMV operations, then the natural place to optimize the algorithm is by improving the AXPY and dot product kernels, or replacing them with faster kernels.

The recurrence used to compute the next basis vector from the previous basis vectors need not have anything to do with orthogonalization, but generally long recurrences are only used when they orthogonalize. Also, both one-sided and two-sided KSMs may use either a short recurrence or a long recurrence. (One example of a two-sided, long-recurrence Krylov method is the "Two-sided Arnoldi" method of Ruhe [205].) For one-sided methods, a short recurrence generally suggests that the method assumes something about the mathematical structure (e.g., symmetry) of the matrix A. We discuss this next.

Mathematical structure of A

The third criterion for categorizing Krylov methods, is whether they assume something about the mathematical structure of the matrix A, such as whether it is symmetric. Many KSMs assume that the matrix A is symmetric. Such methods include CG, symmetric Lanczos, and MINRES. There are specialized Krylov iterations for other mathematical structures, such as complex symmetric ($A = A^T$ but $A \neq \overline{A}^T$) or skew-symmetric ($A = -A^T$). Other Krylov algorithms make no such assumptions about the symmetry of A. Arnoldi iteration, GMRES, and BiCG are three examples.

Symmetry assumptions generally relate to the use of a short recurrence. Assuming symmetry lets many KSMs maintain the orthogonality of their Krylov basis vectors without orthogonalizing each basis vector explicitly against all previous basis vectors. That means they can use a short recurrence. This is the difference between Arnoldi and symmetric Lanczos: the latter assumes that the matrix A is symmetric. Otherwise, the two algorithms compute the same thing in exact arithmetic.

Symmetry assumptions also affect the development of our new communication-avoiding KSMs, especially when preconditioning is included. This is because the preconditioned matrix may or may not be symmetric, even if both the preconditioner and the matrix A are symmetric. This influences both how we replace the sparse matrix-vector products and preconditioner applications with a communication-avoiding kernel, and how we make the resulting basis vectors orthogonal. We discuss this issue further in Section 4.3.

Mathematical structure of the preconditioner

Our fourth and final criterion for categorizing Krylov methods, is the mathematical structure of the preconditioner. *Preconditioning* is a technique for accelerating convergence of a Krylov method when solving $Ax = b$. It amounts to solving a different linear system (or set of linear systems) with the same solution as the original problem $Ax = b$. A *preconditioner* is an approximation $M \approx A$ of the matrix A. Preconditioning may be applied in three different ways:

- *Left preconditioning* means the Krylov method solves $M^{-1}Ax = M^{-1}b$ for x.
- *Right preconditioning* means the Krylov method solves $AM^{-1}y = b$ for y and $Mx = y$ for x.
- *Split preconditioning* means that a factorization $M_L M_R = M$ of the preconditioner M is available, and the Krylov method solves $M_L^{-1}Ay = M_L^{-1}b$ for y and $M_R x = y$ for x.

The most important special case of split preconditioning in this thesis is *symmetric split preconditioning*. This means that the preconditioner M is symmetric with a factorization $M = M_L M_L^*$, for example the Cholesky decomposition.

Preconditioning changes the Krylov subspace(s) which are integral to the structure of a Krylov method. It changes both how the basis vectors of those subspace(s) are computed, and the relationship of the basis vectors to each other. For example, the Method of Conjugate Gradients (CG), either without preconditioning or with symmetric split preconditioning, computes residual vectors that are orthogonal to each other with respect to the usual Euclidean inner product, in exact arithmetic. In left-preconditioned CG with symmetric positive definite preconditioner M, the residual vectors are orthogonal with respect to the M inner product $\langle x, y \rangle_M = y^* M x$. This is not true for left-preconditioned GMRES, for example: the Krylov basis vectors in GMRES are always Euclidean-orthogonal, in exact arithmetic. This distinction is not only theoretical; it affects which kernels we may use in the KSM. For instance, we may only use a QR factorization to orthogonalize a group of basis vectors, if they are to be make Euclidean-orthogonal. A different kernel must be used if the vectors are to be made orthogonal with respect to a different inner product. We will discuss this issue in detail in Section 4.3. Preconditioning also changes how our communication-avoiding methods compute the Krylov basis vectors; see Section 2.2 for details.

Categories of Krylov methods

We now use the above three criteria to identify three categories of Krylov methods, with examples in parentheses after each category.

1. One-sided, long recurrence, general A, no or any preconditioner (Arnoldi, GMRES)
2. One-sided, short recurrence, symmetric A, no preconditioner or symmetric split preconditioner (symmetric Lanczos, CG, MINRES)
3. Two-sided, long or short recurrence, general A, no preconditioner (nonsymmetric Lanczos, BiCG, CGS, BiCGSTAB, QMR)

We will discuss communication-avoiding versions of the first category of KSMs in Chapter 3. Chapters 4 and 5 will concern communication-avoiding versions of the second category of methods. Finally, Chapter 6 will briefly discuss how one might go about developing communication-avoiding versions of algorithms in the third category. We do not go into detail for algorithms in the third category, for reasons mentioned in that chapter.

Exceptions

Some Krylov methods do not fit into one of the three categories mentioned above. Chebyshev iteration [172], which lacks inner products and only uses sparse matrix-vector products and AXPY operations to construct successive basis vectors, is one example. Block Krylov methods are another exception. They use more "exotic" kernels such as sparse matrix times multiple vectors (SpMM) and possibly also rank-revealing decompositions for performing deflation. We discuss both Chebyshev iteration and block Krylov methods, along with some other exceptions, in Section 1.6. In this thesis, we will only investigate methods in one of the above three categories.

1.2 Communication and performance

In this section, we explain what communication is, and why it is expensive relative to floating-point arithmetic. This will justify why we develop communication-avoiding Krylov methods, which communicate less than their corresponding standard Krylov methods, possibly by performing redundant arithmetic.

In Section 1.2.1, we define communication, which includes both data movement between processors working in parallel, and data movement between levels of the memory hierarchy. We also describe the simple "latency-bandwidth" performance model used throughout this work to predict the performance of computational kernels. In Section 1.2.4, we give evidence that communication is expensive relative to arithmetic, and that this will likely continue to be the case for years to come. Finally, in Section 1.2.5, we explain what it means for an algorithm to be "communication-avoiding."

The discussion of communication's effect on performance in this section will help us understand how kernels in standard Krylov methods are slow (Section 1.3). In Section 1.4, we will introduce our new communication-avoiding kernels: the matrix powers kernel, Tall Skinny QR (TSQR), and Block Gram-Schmidt orthogonalization (BGS). By constructing our new Krylov methods out of these kernels, we will make these methods communication-avoiding.

1.2.1 What is communication?

If we want to avoid communication, we first must understand what "communication" means. In this work, we assume that computers do two things which matter for performance: arithmetic, and communication. "Arithmetic" here means floating-point arithmetic operations, like the addition, multiplication, or division of two floating-point numbers. We define *communication* generally as data movement. This includes both data movement between levels

of a memory hierarchy, which we call *sequential* communication, and data movement between processors working in parallel, which we call *parallel* communication. We include in the memory hierarchy all forms of storage connected to a CPU ("Central Processing Unit" or processor), such as a cache, main memory, a solid-state drive, a hard disk, or a tape archive. These may be connected directly to the CPU or over a network interface. The main memory of modern computers is generally made of *DRAM* (Dynamic random access memory), so we use DRAM as a synonym for main memory.

Communication between parallel processors may take the following forms, among others:

- Messages between processors, in a distributed-memory system
- Cache coherency traffic, in a shared-memory system
- Data transfers between coprocessors linked by a bus, such as between a CPU and a GPU ("Graphics Processing Unit")[5]

Data movement between levels of a memory hierarchy may be either

- under automatic control, e.g.,
  - between a (hardware-managed) cache and main memory, or
  - between main memory and swap space on disk, as in a virtual memory system;
- or, under manual control, e.g.,
  - between a local store and main memory, or
  - between main memory and disk (as in an "out-of-core" code that manages disk space explicitly, in order to work with more data than can fit in main memory at once).

If we are considering two levels within a memory hierarchy, we call them *fast memory* and *slow memory*. Fast memory is the level with lower latency to the CPU, and generally smaller capacity, than slow memory. "Fast" and "slow" are always relative to the two levels being examined. For example, when considering cache and DRAM, cache is "fast" and DRAM is "slow." When considering DRAM and disk, however, DRAM is "fast" and disk is "slow."

1.2.2 Performance model

In this work, we use a simple model we call the "latency-bandwidth model" to estimate an algorithm's run time. We will first state this model, and then justify it afterwards. In our model, communication happens in messages. A *message* is a sequence of n words in consecutive memory locations. We say "message" regardless of whether the implementation uses a shared-memory or distributed-memory programming model. The time required to send a message containing n words is

$$\text{Time}_{\text{Message}}(n) = \alpha + \beta \cdot n,$$

[5]In recent years, GPUs have come to be used as coprocessors for tasks other than graphics computations.

where α is the latency (with units of seconds) and β is the inverse bandwidth (seconds per word). The time required to perform n floating-point operations is

$$\text{Time}_{\text{Flops}}(n) = \gamma \cdot n,$$

where γ is the inverse floating-point rate (also called the floating-point *throughput*), with units of seconds per floating-point operation.

For parallel performance, we count run time along the critical path of the algorithm. For the memory hierarchy, each level L has a finite capacity W_L of words, which may be different for each level. Only floating-point words (and their associated indices, if applicable for a sparse matrix computation) are counted towards filling that capacity. Instruction words are not counted. We assume that floating-point words and integer words are the same size, since for the computations of interest in this thesis, they are within a small constant factor of the same size. The latency $\alpha_{\text{L1,L2}}$ and inverse bandwidth $\beta_{\text{L1,L2}}$ between two levels L1 and L2 of the memory hierarchy may be different, depending on the levels being considered. Furthermore, we count floating-point addition, multiplication, and division as having the same throughput γ.

We make the following simplifying assumptions about the memory hierarchy:

- Caches are fully associative: there can only be compulsory misses or capacity misses, no conflict misses.
- We model a cache of capacity W shared among P processors, as P separate fully-associative caches of capacity $\lfloor W/P \rfloor$ each.
- There is no overlap of communication and arithmetic events, nor is there overlap of different communication events.

We make the following simplifying assumptions about parallel computers and algorithms:

- There are P processors, and that number never changes during the computation.
- The P processors are identical: they have the same floating-point throughput γ, and the same memory capacity W.
- Messages are only from one processor to one other processor. Collectives (such as reductions) must be implemented using these "point-to-point" messages.
- The latency and bandwidth of a message does not depend on the processors participating in the message.
- We do not limit the number of messages or number of words which can be "in flight" simultaneously. (There is no "bisection bandwidth" in our model, for example.)

Why latency and bandwidth?

Not all models of communication include both a latency term and a bandwidth term. For example, the paper by Wulf and McKee that coined the term "memory wall" [247] only considers memory bandwidth ("DRAM access time") and cache capacity. We include both because memory latency (in terms of accessing noncontiguous data from slow memory) does matter for communication optimality, even for dense linear algebra algorithms. See e.g., Ballard et al. [22]. Most storage devices, including DRAM to hard disks, achieve their highest data transfer throughput for contiguous data. Also, nearly all communication devices have different costs for sending one very short message, and sending one long message. More sophisticated parallel performance models, such as the LogGP model of Alexandrov et al. [3], account for the higher bandwidth for long messages that some communication networks have.

Why no overlap?

Many forms of communication allow the overlapping of data transfer operations and other computation, or the overlapping of multiple data transfer operations with each other. For example, a program may spawn off an additional thread to wait on a disk read operation, and continue to do useful work in the original thread until the disk read has finished. Some network interfaces have on-board processors that off-load much of the overhead of sending a message from the CPU.

Overlapping is an important technique for improving performance, including kernels in Krylov methods such as SpMV. However, Section 1.1.3 explains that the data dependencies in standard Krylov methods preclude improving performance by more than a factor of $2\times$. This is why we do not consider overlap in our model. Nevertheless, it is usually good for performance. Indeed, it may help improve the performance of our communication-avoiding Krylov methods, especially when combined with dataflow scheduling that cuts across kernels, as we discuss briefly in Section 3.6.5.

What about cache associativity?

Our model of the memory hierarchy assumes that caches are fully associative. We assume only that data must be brought into cache, and count that transfer cost. In cache terminology, we only model compulsory misses (due to the data not yet being in cache), and capacity misses (due to the cache having a finite capacity). We do this for several reasons:

- Standard Krylov methods have little temporal data locality, as we will show later in this work, so compulsory misses are more important for describing their performance.
- Moving less data ("avoiding communication") between slow and fast memory is generally always good for performance, regardless of cache associativity.
- Devising algorithms that exploit cache associativity is much harder than devising algorithms that simply move less data.
- We want our algorithms to work for general memory hierarchies, rather than just for cache and DRAM, so we use a general model rather than a cache-specific model.

What about shared caches?

A further simplification we make is that caches are not shared among processors. We model a cache of capacity W shared among P processors, as P non-shared caches of capacity $\lfloor W/P \rfloor$ each. This is because when we model the memory hierarchy, we are mainly interested in memory traffic issues relating to capacity and compulsory misses. If multiple processors share the same cache, that reduces the effective capacity of that cache.

In shared-memory systems, a shared cache may be used for low-latency communication between processors that share the cache, but we model this as parallel communication rather than as memory hierarchy communication. That is, communication of two processors through a shared cache is just "communication of two processors"; the cache is just the mechanism.

Shared caches are subject to a phenomenon called *false sharing*, in which the cache thinks that its processors are sharing data, but they are not. For example, two processors might be accessing two different words within the same cache line. Alternately, the associativity of the cache may cause two different words in memory, each being accessed by a different processor sharing the same cache, to map to the same location in that cache. This phenomenon is analogous to cache conflict misses in the sequential case, and we do not model it in this thesis for the same reasons.

1.2.3 Simplified parallel and sequential models

Most modern computers include both parallel processors, and a memory hierarchy. Our kernel implementations exploit both in a hierarchical fashion, as we explain for example in Chapter 2. We do not try to model both parallelism and the memory hierarchy, or try to model more than two levels of the memory hierarchy at once. Rather, we only use two models:

- "Parallel": $P \geq 1$ processors, no memory hierarchy.
- "Sequential": One processor, two levels of memory hierarchy (fast and slow memory). Fast memory has capacity W floating-point words, with zero latency and infinite bandwidth, and slow memory has infinite capacity, but positive latency α and finite bandwidth $1/\beta$.

These models may be nested arbitrarily to model different types of computers. Since our communication-avoiding kernels (see Section 1.4 and Chapter 2) have a natural hierarchical structure, we conjecture (but do not prove) that they are communication-optimal for many different kinds of hardware, not just for the two models above.

1.2.4 Communication is expensive

Our reason for developing communication-avoiding algorithms is that communication is much slower than computation, and has been and will likely continue to get slower relative to computation over time. This includes both the latency and bandwidth costs of moving data between levels of the memory hierarchy, and the costs of moving data between processors.

The so-called "memory wall" – the hard upper bound on single-processor performance, due to memory bandwidth being slower than instruction throughput – is the most direct cause. Another wall – the "power wall" – has caused CPU manufacturers to abandon sequential hardware performance improvements, in favor of parallel processing. All of the techniques computer architects used to use to improve sequential performance now either dissipate too much power, or have rapidly diminishing returns. More parallelism means more communication, which exacerbates the memory wall. Even radical developments in hardware technology may not break through either of these walls any time soon, which means that algorithms and software must step in to continue performance improvements.

The memory wall

The *memory wall* refers to the large and exponentially increasing gap between main memory throughput, and single-processor arithmetic throughput. Wulf and McKee [247] popularized this term. They argued that even if all but a tiny fraction of code and data accesses hit in the cache, the decrease of main memory bandwidth relative to instruction bandwidth over time means that eventually the processor will be starved for data and / or instructions. Thus, for any program that performs n arithmetic operations and other non-memory instructions, if it requires within a constant factor of that number of main memory reads and / or writes, it will eventually have memory-bound performance. This is true even if the cache has infinite bandwidth. It is also true even if the entire code and data set fits in cache (i.e., there are no capacity misses), as long as the code and data are not in cache when the program begins.

The memory wall is related to *Amdahl's Law* of parallel performance (see e.g., Amdahl [4]). Amdahl's Law states that the performance benefit of parallelism in a program is limited by the fraction of time the program spends in sequential computations, no matter how many parallel processors are available. In our case, the "sequential computations" are data movement: if a program spends most of its time moving data, its performance will be limited by the cost of moving data, no matter how fast the arithmetic computations are. Amdahl's Law makes it clear that sparse matrix computations are vulnerable to the memory wall, because they perform about as many memory reads and writes from slow memory as they perform floating-point operations (see Section 1.3).

Another phrase sometimes used as a synonym for the memory wall is the *von Neumann bottleneck*. Backus [12] coined this term in his 1977 ACM Turing Award lecture. Here, "von Neumann" refers to von Neumann computer architectures, where code and data must pass through the same (narrow) pipe between main memory and the CPU. Backus uses this bottleneck to argue for functional programming, which has the potential to reduce memory traffic (since fewer memory references have to pass between memory and the CPU). While we do not argue for a style of programming in this thesis, our new Krylov methods have the same goal: to reduce memory traffic to and from main memory. Backus also argues that the von Neumann bottleneck is "intellectual" as well, since the requirements of imperative programming constrain both architectures and software. The conventional wisdom that seeks to minimize floating-point operations, regardless of the cost in data movement, is also a kind of intellectual bottleneck, which we hope the "communication-avoiding" paradigm will overcome.

Latency Wulf and McKee explain the memory wall in terms of bandwidth, but the term is often used in reference to memory latency. This is because DRAM, and many other storage devices, only can be accessed at full bandwidth rate if the data being accessed is contiguous or has a specific regular layout. Many programs do not access data in a sufficiently regular way to ensure the bandwidth rate of access from memory, so they must pay latency costs to access their data.

The most dramatic illustration of the memory wall is to compare DRAM latency with floating-point instruction throughput over time.[6] For example, Graham et al. [119] observe that between 1988 and 2004, the floating-point rate of commodity microprocessors improved by 59% yearly. However, DRAM latency (more specifically, DRAM row access time) decreased by only 5.5%. In 1988, one floating-point operation took six times as long as fetching one word from main memory, but in 2004, one memory fetch took as long as over 100 floating-point operations. These trends continued after 2004, even as typical CPUs changed from a single processing core, to multiple processing cores on a single die. For example, in 2007, according to Kogge et al. [158, Figure 6.12], typical DRAM latency was three orders of magnitude greater than typical CPU cycle time (which is the same order of magnitude as floating-point instruction throughput). In contrast, in 1982, typical CPU cycle time averaged slightly less than typical DRAM latency. (Post-2004 CPU cycle times in Kogge et al. account for the effective CPU performance increase, given multiple cores on the same die.)

The latency of other communication operations, such as hard disk fetches and sending messages over a network, follow a similar trend as DRAM latency. For example, sending a single, short message between processors over a network interface may take several microseconds, a time in which a single node may be able to complete thousands of floating-point operations [158]. The fastest solid-state storage drives have comparable latency, and the latency of a single access to a hard disk is three orders of magnitude slower (1–10 ms, instead of 1–10 μs). Hard disk access time cannot decrease much more than that, since it is mainly a function of how fast the physical magnetic disk can spin. Disks cannot spin much faster than they already do (up to 15,000 revolutions per minute for modern high-performance drives) without physically deforming or otherwise becoming damaged.

Hardware architects often use increasingly deep memory hierarchies to try to evade the memory wall, but this makes the performance of applications ever more sensitive to locality. More specifically, it punishes applications dominated by memory operations, whether the applications are latency-bound or bandwidth-bound. (We say "punishes" because deep memory hierarchies may even increase memory latency if there is no temporal locality, due to cache overhead.) We will show in Section 1.3 that most Krylov methods belong in this category.

[6] Floating-point instruction throughput does differ from the latency of an individual floating-point instruction, since floating-point operations may be pipelined and / or the CPU may have multiple floating-point units that can work in parallel on independent instructions. Nevertheless, we use instruction throughput for the comparison with memory latency. We do this first, because memory latency is so much slower than the latency of an individual floating-point instruction that the difference between instruction latency and throughput does not matter. Second, the CPU executes instructions besides floating-point arithmetic and loads and stores, and these instructions are often pipelined with the floating-point arithmetic operations, so the latency of the floating-point operations may very well be hidden by that of the other instructions.

Bandwidth Memory bandwidth has also grown exponentially slower than arithmetic throughput. For example, while floating-point performance improved by 59% between 1988 and 2004, DRAM memory bandwidth (more specifically, front-side bus bandwidth) increased by only 23% between 1995 and 2004 [119]. The bandwidth of other storage devices, such as disks, as well as the bandwidth of network interfaces, have also not tracked the rapid increase in floating-point throughput. As we will discuss in Section 1.4, the memory hierarchy performance of typical Krylov methods on a single node is generally bound by bandwidth.

There are ways to increase bandwidth of storage devices. Asanovic et al. [9] (see also [10]) observe that DRAM technology has untapped potential for bandwidth improvements, for example, since DRAM storage cells themselves have much more potential parallelism than the circuits by which they are accessed. Typical ways to improve the bandwidth of storage devices include

- Interleaving data across multiple storage devices, which multiplies effective sequential access bandwidth by the number of thusly linked devices. For DRAM these multiple devices are called *banks*, and for hard disks this technique is called *striping*.
- For parallel processors accessing a shared level of the memory hierarchy, partitioning that storage into separate devices or regions, each of which prioritizes service to a disjoint subset of processors. This increases effective bandwidth (and /or decreases effective latency) of a processor to its "local" subset of storage. For DRAM, this technique is called *Non-Uniform Memory Access* (NUMA), and for hard disks or analogous storage devices, this amounts to a distributed file system.

However, these techniques often come at the cost of latency increases (see e.g., Kogge [158]. For example, interleaving data across multiple memory banks increases the overhead of accessing a single word of data. NUMA technology essentially adds another level to the memory hierarchy, since programmers have to reason about locality, and applications pay higher latency and / or bandwidth costs to access nonlocal data. Furthermore, increasing bandwidth of a storage device or network interface usually increases hardware complexity, and therefore cost and energy consumption.

The power wall

The *power wall* refers to power dissipation being the major limiting factor of single-processor CPU performance. See Asanovic et al. [10]. For decades until around 2004, sequential CPU performance has increased at an exponential rate. Part of this was caused by an exponential increase in CPU clock frequency each year. This clock frequency increase was aided by techniques that let hardware designers reduce cycle time, by reducing the amount of work to be done in each cycle. These techniques include

- RISC (the Reduced Instruction Set Computer, with its simplified instruction set), and
- pipelining (which broke up instructions into more, but shorter, steps, increasing instruction throughput at the expense of latency).

See e.g., Click [62]. Pipelining is also a simple example of *instruction-level parallelism* (ILP), which entails finding instructions in sequential code that can be executed in parallel on a single processor, and doing so using parallel hardware resources on that single processor. (ILP is a way to make sequential code go faster on a single processor; (explicitly) parallel code executes on multiple processors.) Pipelining and more advanced ILP techniques, such as multiple instruction issue and out-of-order execution, also improved the performance of sequential code.

Since around 2004, CPU clock frequency increases have slowed, stopped, or even reversed, in some cases. This is because CPUs are using too much power, and therefore running too hot. Power use (and thus, heat generation) of a CPU increases as the square of its clock frequency. CPUs have reached the limits of how much heat their physical package can dissipate, without using unreasonable cooling technologies that consume too much energy. Energy use of the CPU itself is also a major limiting concern. That means sequential performance no longer increases "automatically" every year: the power wall has been reached. See e.g., Asanovic et al. In contrast, heat dissipation and energy use scale linearly in the number of parallel processors, so increasing the number of processors is an attractive path for hardware designers to improve performance.

Furthermore, the ability of CPUs to exploit ILP in sequential programs has reached a plateau. See e.g., Asanovic et al. and Click. The main cause of this "ILP wall" is the memory wall: programs spend too much time waiting for cache misses, for them to get benefits out of ILP. Another cause is that instructions in sequential code often have data dependencies that precludes them executing in parallel. A third cause is that CPU hardware for dynamically detecting these data dependencies between instructions is complex and consumes both power and area on the CPU die. In fact, its use of power and area resources scales as the square of the number of instructions that can be executed in parallel. Some CPUs instead let compilers and / or humans resolve instruction dependencies, but experience shows this to be hard to exploit.

The power wall means that sequential performance will no longer increase exponentially each year, as it had for decades. Instead, CPU manufacturers have shifted to relying on parallel processors for continued performance increases. Specifically, they are exploiting continued Moore's Law exponential improvements in transistor density to fit multiple processors ("cores") on a single CPU die, resulting in "multicore" CPUs. Parallelism is a much more energy- and power-efficient way to improve performance than increasing the clock rate or adding ILP resources. Simpler processing cores, but more of them, are the most energy-efficient approach to improving performance. See e.g., Asanovic et al. The result is that performance can only continue to improve if programmers write explicitly parallel software. Furthermore, the performance of individual processors will not increase rapidly, but the number of processors will increase exponentially. This is true both for embedded platforms and high-performance platforms: cell phones will have 1000-way parallelism, and supercomputers will have $100,000,000$-way parallelism (see Asanovic et al.).

This directly contradicts the conventional wisdom in parallel architecture, expressed for example in Hockney's "Principle of Minimal Parallelism": the best performance comes from using as few, but as fast, processors as possible [136, pp. 14–15]. This is because "No communication is always faster than fast communication, however fast": almost all parallel processing requires communication between processors, and more processors means more

communication. The conventional wisdom would hold, if performance were the only metric of interest, that is, if there were no power wall. The power wall means instead that we must try to improve the performance of Krylov methods, which is already communication-bound, while being forced to communicate more by ever-increasing numbers of slower processors. The only way to do this is to avoid communication whenever possible, which is what our algorithms do.

Physical arguments

One might argue that the aforementioned hardware trends are an artifact of particular choices of technology. Kogge et al. [158] make an effective argument against this, at least for technologies that we can imagine developing over the next five or ten years. In fact, these hardware trends could be said to express the following "law of nature":

> Arithmetic is cheap, bandwidth is money, latency is physics.

The phrase "latency is physics" refers to the fundamental physical limitations preventing latency improvements. One of the most startling observations is that at current CPU clock frequencies, it takes more than one clock cycle for electrical signals just to cross the CPU die (see e.g., Click [62]). Even at the speed of light, information takes nearly a second to cross from the east coast to the west coast of North America. That means, for example, that it is very hard to improve the performance of real applications by connecting geographically distributed supercomputers, even if dedicated communication links exist between them. Some storage media, such as hard disks, have mechanical limitations as well. For example, a rotating storage medium spun too fast will deform beyond the ability of the drive to read it. Current CD-ROMs are within only a small factor of this maximum speed. Latency can be reduced to some extent by reducing or eliminating the software overhead of each communication operation. For example, some high-performance network interfaces have specialized "Direct Memory Access" (DMA) hardware that bypasses hardware bottlenecks between processors and communication channels. Nevertheless, "physics" does pose a current and fundamental lower bound to the latency of many communication operations.

The phrase "bandwidth is money" refers to the costs of improving bandwidth. These costs may include actual money (for additional, more complex hardware), increased energy consumption (which also adds to the electric bill), and increased latency. Increasing bandwidth generally requires techniques like the following:

1. a "wider" data channel that can transfer more bits simultaneously,
2. interleaving accesses across multiple storage units (such as DRAM banks), or
3. techniques for overlapping multiple pending communication events, such as the ability to handle a large number of pending memory accesses.

All of these involve more hardware, whether it be wires, channels, banks, disks, or logic and storage. That means more energy spent, more power to dissipate, more complexity, and therefore more cost (see e.g., [119, pp. 107–8]). It may also mean more latency for accessing noncontiguous data or small amounts of data, for example if multiple interleaved storage units must be accessed in order to find a single datum.

Finally, "arithmetic is cheap" refers to the relative increase in peak floating-point arithmetic performance, relative to how fast data can be moved to those floating-point units. Since processors can perform floating-point operations much faster than they can be supplied with floating-point numbers on which to operate, processors can afford to "waste" arithmetic operations, for example by redundant computation that replaces data movement. This justifies our use of redundant arithmetic operations, for example in the matrix powers kernel (see Sections 2.1 and 2.2), in order to avoid data movement.

1.2.5 Avoiding communication

What does avoiding communication mean?

In Section 1.2.1, we explained that the term "communication" refer to all data movement, either between processors or between levels of the memory hierarchy. We also demonstrated that both in terms of time to complete a single operation and in terms of throughput, communication is much slower than arithmetic and will likely continue to get slower, on almost all computers of interest in the present and for the next few years. This suggests that software must change in order to get some benefit out of that fast arithmetic hardware. "Software" here includes both

- system libraries, such as implementations of the MPI distributed-memory parallel communication library (see e.g., [184]), and
- algorithms, whose implementations may invoke those system libraries in order to perform communication.

Optimizing system libraries is a good way to improve the performance of already deployed software. For example, "smart" MPI implementations should not access the internode network interface when communicating between parallel processes on the same shared-memory node. However, many applications' performance would be communication-bound even if the system libraries were optimally implemented. This suggests that new algorithms are needed, which communicate less than existing algorithms whenever possible, even if achieving this requires more arithmetic operations than alternative formulations. We call algorithms that meet these goals *communication-avoiding algorithms*. Additionally, if we can prove that the algorithms attain lower bounds on communication over a class of such algorithms, then we call them *communication optimal*.

The term "communication avoiding," while seeming awkward, is descriptive. Here is a list of terms we rejected as being inaccurate:

- We do not call these algorithms "low latency" or "high bandwidth," because latency and bandwidth are properties of the hardware or system, not of the algorithm.
- We do not say "latency hiding," because in the literature, that refers to another set of techniques involving overlapping independent operations, in order to accomplish useful work while waiting for one or more operations to complete. This includes overlapping communication with computation (see e.g. [248]). Usually this is not an algorithmic improvement, but an implementation improvement. These techniques are useful, but

as we will show in Section 1.3, overlapping communication with computation can at best improve the performance of Krylov methods by a constant factor.

- We do not say "latency free" or "bandwidth free," because that would imply that no communication takes place. Any nontrivial computation involves some kind of data movement, and data cannot travel faster than the speed of light in a vacuum.

Here is a summary of the characteristics of a communication-avoiding algorithm:

- Relative to other algorithms that perform the same task, it should communicate less.
- Ideally, it should communicate asymptotically less, as a function of the data size and hardware parameters (such as cache size or number of processors).
- The algorithm may achieve this goal by performing extra or redundant computation, if necessary.

Example communication-avoiding algorithm

One example of a communication-avoiding algorithm is *CholeskyQR*, due to Stathopoulos and Wu [216]. While CholeskyQR is not our algorithm, we cite it because it both avoids communication, yet is simple to explain. CholeskyQR computes the "thin" QR factorization $V = QR$ of an $m \times n$ dense matrix V, where $m \geq n$. ("Thin" here and elsewhere means that in the factorization, Q is $m \times n$ and R is $n \times n$.) Generally we assume that $m \gg n$, in which case we call V "tall and skinny." We discuss CholeskyQR along with communication-avoiding QR factorization algorithms that we develop in Demmel et al. [75]. There, we show that if the matrix V is laid out as a column of M row blocks (with $\lfloor m/M \rfloor \geq n$):

$$V = \begin{pmatrix} V_1 \\ V_2 \\ \cdots \\ V_M \end{pmatrix},$$

and if each row block V_j is stored contiguously in memory, then CholeskyQR is communication optimal in terms of the number of words transferred between levels of the memory hierarchy. Obviously the lower bound on communication is that the algorithm read every word of V once, and write every word of Q and R once. CholeskyQR only needs to read each entry of V twice from slow memory into fast memory, and write the corresponding entries of Q and R once from fast memory into slow memory. That makes CholeskyQR asymptotically optimal. (We consider here only the sequential version of CholeskyQR; the parallel version is also communication-optimal.) In contrast, the BLAS 3 version of Householder QR implemented in LAPACK, `DGEQRF`, requires a factor of $\Theta(mn/W)$ more words transferred between fast and slow memory, where W is the number of floating-point words that can fit in fast memory. For details, see Demmel et al. [75, Table 4].

We show an outline of (sequential) CholeskyQR as Algorithm 2. Intuitively, it communicates less than algorithms such as Householder QR or Modified Gram-Schmidt orthogonalization, because it works on blocks that are as "square" as possible, rather than

Algorithm 2 Sequential CholeskyQR

Input: $m \times n$ matrix V, divided into M contiguous row blocks $V_1, \ldots, V_j$, with $\lfloor m/M \rfloor \geq n$
Output: Thin QR factorization $V = QR$, where Q is $m \times n$ with the same row block layout as V, and R is $n \times n$

1: Let C be an $n \times n$ matrix of zeros
2: **for** $j = 1$ to M **do**
3: $\quad C := C + V_j^* \cdot V_j$
4: **end for**
5: Compute the Cholesky factorization $R^*R = C$
6: **for** $j = 1$ to M **do**
7: $\quad Q_j := V_j \cdot R^{-1}$
8: **end for**

column-by-column. See Sections 2.3 and 2.4, as well as Demmel et al. [75] and Ballard et al. [22], for details.

CholeskyQR is not the most accurate communication-optimal version of the "thin" QR factorization. In fact, it can produce a Q factor which is not at all orthogonal, as we discuss in Demmel et al. [75]. Our TSQR factorization presented in that same work always produces a Q factor as orthogonal as that produced by Householder QR, yet it communicates asymptotically no more than CholeskyQR. (We will discuss TSQR further in Section 2.3.) We only use CholeskyQR as an example of a communication-avoiding algorithm.

1.3 Kernels in standard Krylov methods

In Sections 1.1.2 and 1.1.3, we identified the three time-consuming kernels in standard Krylov subspace methods: SpMV, AXPY, and dot products. Preconditioning, if it is part of the method, is a fourth such kernel. In this section, we will briefly summarize existing work, some by us and some by others, showing that the performance of all these kernels is generally dominated by communication costs. This will justify the new kernels we will introduce in Section 1.4 and explain in detail in Chapter 2.

We begin in Section 1.3.1 by showing that the performance of SpMV is communication-bound. In Section 1.3.2, we briefly summarize the performance of various preconditioners, most of which are either communication-bound or spend a significant part of their run time in communication. Finally, in Section 1.3.3, we show that the performance of AXPYs and dot products is communication-bound.

1.3.1 Sparse matrix-vector products

Sequential sparse matrix-vector products

It is well known that memory operations dominate the cost of sequential SpMV. Computing $y := y + A \cdot x$ for sparse $n \times n$ A and dense vectors x and y amounts to a sum

$$y_i := y_i + \sum_{1 \leq j \leq n : A_{ij} \neq 0} A_{ij} x_j.$$

Different sparse matrix data structures implement and order this sum in different ways. However, all of those nonzero entries A_{ij} must be read from main memory. Each A_{ij} is only needed once, so there is no reuse of the nonzero entries of A. Thus, if the nonzero entries of A do not fit in cache, then they can be accessed at no faster rate than main memory bandwidth. More importantly, only two floating-point operations (a multiply and an add) are performed for each A_{ij} entry read from memory. Thus the *computational intensity* – the ratio of floating-point operations to memory operations – is no greater than two. It is in fact less than two, because sparse matrices also store the indices of their nonzero entries, and those indices must be read from memory in order to know which entries of x and y to load and store. Furthermore, typical sparse matrix data structures require indirect loads and / or stores for SpMV. Indirect accesses often interfere with compiler and hardware optimizations, so that the memory operations may not even run at full main memory bandwidth. These issues are discussed e.g., in Vuduc et al. [237] and Williams et al. [241]. Authors such as these have found experimentally that typical sequential SpMV implementations generally achieve no more than 10% of machine peak floating-point rate on commodity microprocessors. In the best case, memory bandwidth usually does not suffice to keep all the floating-point units busy, so performance is generally bounded above by peak memory bandwidth.

SpMV differs markedly from dense matrix-matrix operations like dense matrix-matrix multiplication or dense LU factorization (as in the LINPACK benchmark, for which see Dongarra [83]). For dense matrix-matrix operations, the number of floating-point operations ($\Theta(n^3)$ for $n \times n$ matrices) is asymptotically larger than the amount of data ($\Theta(n^2)$). Irony et al. [142] show that the number of words transferred between fast and slow memory in an $n \times n$ sequential matrix-matrix multiply is $\Omega(n^3/\sqrt{W})$, where W is the fast memory capacity in words. The usual tiled algorithm achieves this lower bound. Ballard et al. [22] extend this lower bound on communication to factorizations of an $n \times n$ matrix, such as Cholesky, LU, QR, LDL^T, dense eigenvalue decompositions, and dense singular value decompositions. They also cite other works giving algorithms that achieve these lower bounds, including our "Communication-Avoiding QR" (CAQR) algorithm [75]. Since these dense matrix-matrix operations involve $\Theta(n^3)$ floating-point arithmetic operations, they may therefore be computed at the full speed of the arithmetic hardware, as long as the fast memory is sufficiently large. This is true no matter how large the matrix is. Such is not true for SpMV.

Some optimizations can improve the computational intensity of SpMV for certain classes of matrices. For example, replacing individual nonzero entries of A with small dense blocks – a technique called *register blocking* – can achieve a small constant factor of reuse of matrix entries, and also reduces the number of indices needing to be read from slow memory. For details, see e.g., Vuduc et al. [235, 237] and Williams et al. [241]. For a small class of nonsquare matrices, a technique called *cache blocking* may increase reuse in the source vector. See Nishtala et al. [179] for details. Nevertheless, none of these optimizations can make the performance of sequential SpMV comparable with the performance of dense matrix-matrix operations.

Parallel sparse matrix-vector products

It is much harder to characterize the performance of parallel SpMV, because it depends on many more factors than sequential SpMV. Typical sequential SpMV performance de-

pends mainly on memory bandwidth, as mentioned above. In the parallel case, performance depends on

- how the matrix and vectors are divided among the processors,
- whether a processor's component of the matrix fits in its cache,
- and the structure of the matrix.

There are many ways to split up the matrix and vectors among P processors. The conventional way for a square matrix A, is to

1. partition the dense destination vector y (in $y := y + A \cdot x$) into blocks of contiguous elements y_1, y_2, ..., y_P,
2. partition the dense source vector x similarly into blocks of contiguous elements x_1, x_2, ..., x_P,
3. partition the sparse matrix A into corresponding sparse submatrices A_{IJ} with $1 \leq I, J \leq P$,
4. associate x_I, y_I, and $A_{I,J}$ (for $1 \leq J \leq P$) with processor I.

("Associate" here conveys a sense of memory locality, whatever that may mean in terms of hardware and software.) This results in a "1-D" block row decomposition of $y := y + A \cdot x$:

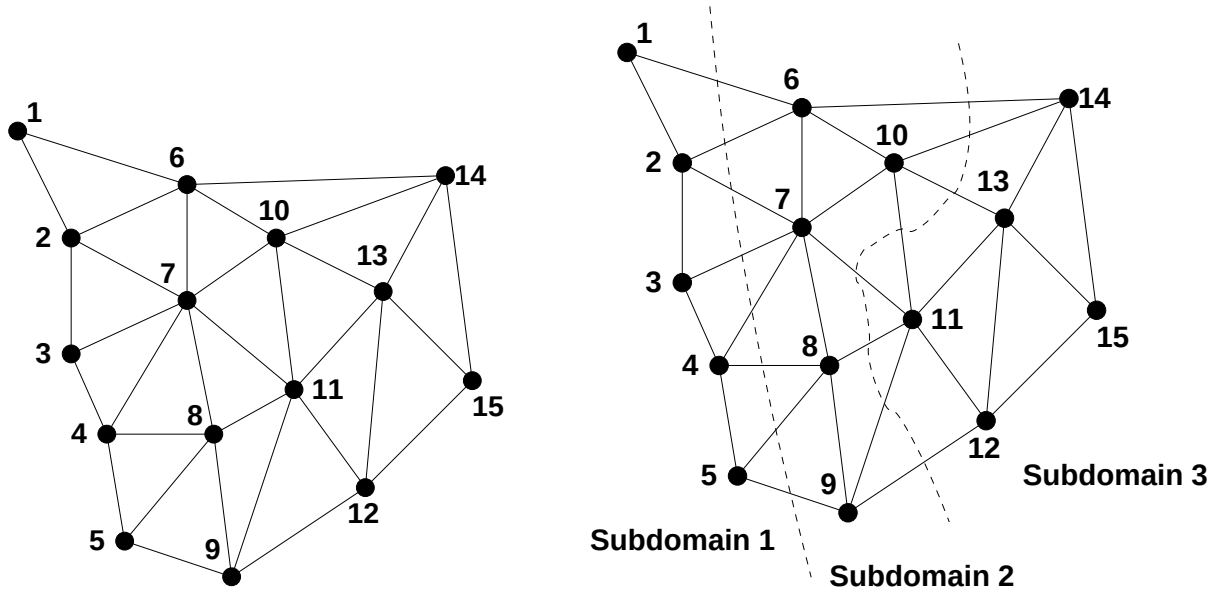
$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_P \end{pmatrix} := \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_P \end{pmatrix} + \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1P} \\ A_{21} & A_{22} & \ddots & A_{2P} \\ \vdots & \ddots & \ddots & \vdots \\ A_{P1} & A_{P2} & \dots & A_{PP} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_P \end{pmatrix}.$$

Processor I computes its block y_I via the sum

$$y_I := y_I + \sum_{J=1}^{P} A_{IJ} x_J.$$

Each A_{IJ} block is a sparse matrix which processor I can access directly. If that block contains no nonzero entries, processor I need not compute that term in the sum. Furthermore, if $I \neq J$, processor I has to fetch entries of x_J from processor J. In fact, one entry of x_J must be fetched from processor J for every column of A_{IJ} containing a nonzero entry. This fetching involves communication: in the distributed-memory programming model, it entails sending and receiving messages, and in the shared-memory programming model, it entails memory traffic from a region of memory belonging to a different processor. In the case of NUMA hardware, that memory region may not have affinity to processor I, so processor I will pay a latency and possibly also a bandwidth penalty in order to access it.

In this simple "1-D" decomposition of the matrix and vectors, the number of messages that processor J must send is related to the number of nonzero A_{IJ} blocks with $I \neq J$ – that is, the number of nonzero off-diagonal blocks A_{IJ} in block row I. Also, the number of



(a) Undirected graph resulting from a triangulation of 15 points in the plane. Each line segment between points represents an edge.

(b) The same undirected graph as in Figure 1.1a, partitioned into three subdomains (1, 2, and 3). The dashed lines indicate the boundaries between subdomains.

Figure 1.1: Illustration of partitioning an undirected graph.

words of x_J that processor J must send is related to the number of nonzero columns in those off-diagonal blocks.

We may explain this relationship in a simplified way by speaking of the *surface-to-volume ratio* between partitions. "Volume" here does not imply three dimensions necessarily, and "surface" does not imply two dimensions; rather, "surface" indicates the boundary between two partitions. Suppose that the graph structure of the sparse matrix comes from an irregular mesh resulting from the triangulation of a 2-D region (see Figure 1.1). Each vertex in the graph is connected to its nearest neighbors. Partitioning the mesh into three sets of vertices (Figure 1.1b) corresponds to partitioning the sparse matrix into three block rows (Figure 1.2). The number of words that must be communicated from Processor 1 (which owns Subdomain 1) to Processor 2 (which owns Subdomain 2) is the number of edges in the graph connecting Processor 1 to Processor 2. What "flows" along those edges are the source vector values x_j at the vertices along the boundary. Clearly, at least one message must pass between Processors 1 and 2. However, Processors 1 and 3 need not communicate, since there are no edges crossing between their subdomains; their subdomains do not "touch" (see Figure 1.1b).

The exact relationship between the nonzero structure of A and the communication volume, when using this 1-D block row decomposition, is explained in detail by Çatalyürek and Aykanat [48]. The authors explain that the model of the previous paragraph is imperfect. In particular, it is only good at describing sparse matrices coming from triangulations of meshes, with nearest-neighbor interactions only; it is not so good at describing more general sparse matrices. However, it illustrates the connection between the sparse matrix structure and the communication volume.

$$A = \begin{pmatrix} A_{11} & A_{12} & 0 \\ A_{21} & A_{22} & A_{23} \\ 0 & A_{32} & A_{33} \end{pmatrix}$$

$$= \left(\begin{array}{ccccc|ccccc|ccccc}
a_{11} & a_{12} & & & & a_{16} & & & & & & & & & \\
a_{21} & a_{22} & a_{23} & & & a_{26} & a_{27} & & & & & & & & \\
 & a_{32} & a_{33} & a_{34} & & & a_{37} & & & & & & & & \\
 & & a_{43} & a_{44} & a_{45} & & a_{47} & a_{48} & & & & & & & \\
 & & & a_{54} & a_{55} & & & a_{58} & a_{59} & & & & & & \\
\hline
a_{61} & a_{62} & & & & a_{66} & a_{67} & & & a_{6,10} & & & & a_{6,14} & \\
 & a_{72} & a_{73} & a_{74} & & a_{76} & a_{77} & a_{78} & & a_{7,10} & a_{7,11} & & & & \\
 & & & a_{84} & a_{85} & & a_{87} & a_{88} & a_{89} & & a_{8,11} & & & & \\
 & & & & a_{95} & & & a_{98} & a_{99} & & a_{9,11} & a_{9,12} & & & \\
 & & & & & a_{10,6} & a_{10,7} & & & a_{10,10} & a_{10,11} & & a_{10,13} & a_{10,14} & \\
\hline
 & & & & & & a_{11,7} & a_{11,8} & a_{11,9} & a_{11,10} & a_{11,11} & a_{11,12} & a_{11,13} & & \\
 & & & & & & & & a_{12,9} & & a_{12,11} & a_{12,12} & a_{12,13} & & a_{12,15} \\
 & & & & & & & & & a_{13,10} & a_{13,11} & a_{13,12} & a_{13,13} & a_{13,14} & a_{13,15} \\
 & & & & & a_{14,6} & & & & a_{14,10} & & & a_{14,13} & a_{14,14} & a_{14,15} \\
 & & & & & & & & & & & a_{15,12} & a_{15,13} & a_{15,14} & a_{15,15}
\end{array}\right)$$

Figure 1.2: The symmetric sparse matrix structure corresponding to the undirected graph in Figure 1.1, partitioned into three subdomains as in Figure 1.1b. Each undirected edge between vertices i and j corresponds to a pair of nonzero entries a_{ij} and a_{ji} in the sparse matrix A, and we also assume that a_{ii} is nonzero for each vertex i (the "self-loops" in the graph are not shown). The horizontal and vertical lines in the matrix mark out the blocks corresponding to the three partitions.

There are many more ways to divide up the matrix and vectors among processors than the simple "1-D" layout presented above. Vastenhouw and Bisseling [231], for example, describe 1-D, 2-D, and even more general block layouts both for square A and nonsquare A. However, this example conveys the following general truths about parallel SpMV. First, only very specific matrix structures require no interprocessor communication. (In the case of the above simple 1-D block layout, only a block diagonal matrix A, with $A_{IJ} = 0$ when $I \neq J$, would result in no communication between processors.) Communication is the common case. Second, the nonzero structure of A and the distribution of A, x, and y among the processors strongly affect

- how many messages one processor must send to different processors,
- the volume of data it must send, and
- how much arithmetic each processor has to do.

Third, both the communication and arithmetic workload may be different for different processors, depending again both on the nonzero structure of A, and on the distribution of A, x, and y among the processors.

Graph or hypergraph partitioners are an essential technology for reducing both the number of messages and the total communication volume between processors in parallel SpMV, as well as for load balancing. This is because optimizing any of these things entails computing a K-way partitioning of a graph or hypergraph related to the structure of the sparse matrix, where K is related to the number of processors (for example, $K = P$ when computing a 1-D block row decomposition). See e.g., Çatalyürek and Aykanat [48], Vastenhouw and Bisseling, Bisseling and Meesen [31], and Wolf et al. [243]. As we will see in Section 2.1, our communication-avoiding sparse matrix kernel (Section 2.1) starts with an analogous partitioning step, but further reduces communication in ways that we will describe later.

The above simple 1-D partitioning shows that an important part of parallel SpMV is sequential SpMV. It is hard to characterize the proportion of time that parallel SpMV spends in sequential SpMV, relative to how much time it spends in communication between processors, because there are many different kinds of parallel processors and sparse matrices. Nevertheless, the common experience is that even if both the computational and communication load are balanced evenly among the processors, the cost of communication is enough, in distributed-memory computations at least, to justify the expense of a complicated reordering scheme that reduces communication volume and / or the number of messages. See e.g., Wolf et al. [243]. This suggests that interprocessor communication is an important part of parallel SpMV, and that avoiding that communication could improve performance significantly.

1.3.2 Preconditioning

Preconditioning is a large field, and there are many different types of preconditioners. We do not try to characterize the performance of all known preconditioners here, or even try to give a survey of previous work doing such characterizations. Rather, in this thesis, we will only address a few classes of preconditioners, which we hope are somewhat characteristic of the performance of common preconditioners. We also focus mainly on the types of preconditioners for which it it possible to avoid communication. In addition, we do not consider

the cost of *constructing* the preconditioner from the original problem; that is generally a one-time cost, whereas *applying* the preconditioner is done at every iteration of a standard Krylov method.[7]

The most obvious preconditioner is itself a sparse matrix, applied using SpMV. For example, computing a sparse approximate inverse preconditioner (SPAI – see e.g., Chow [53, 54]) produces a sparse matrix, and the preconditioner is applied using SpMV. We have already mentioned how the performance of SpMV is dominated by communication. In Section 2.2, we will show how to avoid communication in a Krylov method for both SpMV with the matrix A, and the preconditioner step, if the preconditioner is a sparse matrix with a similar structure as A.

Another popular class of preconditioners involve sparse matrices, but applied via sparse triangular solve (SpTS), rather than SpMV. These include sparse incomplete factorizations, such as Incomplete Cholesky or Incomplete LU. The sparse triangular factors produced by sparse incomplete factorizations are applied via SpTS. Applying a support-graph preconditioner (see e.g., Gremban and Miller [122] and Bern et al. [28]) involves a direct solve of a sparse linear system, whose structure means that the direct solve behaves much like a sparse triangular solve in terms of performance. We do not discuss in this thesis how to avoid communication in SpTS. While we do not have the space here to model the performance of SpTS in detail, it should be clear that both sequential and parallel SpTS have communication-bound performance. In particular, parallel SpTS involves more data dependencies between processors than SpMV, so it is fair to say that it requires even more communication (in general) than SpMV.

One class of preconditioners for which we do show how to avoid communication later in this thesis, are matrix preconditioners with a hierarchical matrix structure (see Börm et al. [37]) or a (hierarchically) semiseparable structure (see e.g., Chandrasekaran et al. [50]). In the ideal case, these structures let one compute an $n \times n$ matrix-vector product with such a matrix in $O(n \log n)$ operations, with $O(n \log n)$ memory requirements. This means also that the sequential performance of these sparse matrix-vector products is at best bandwidth-bound, since there are about as many words to read from slow memory as there are floating-point arithmetic operations to perform. These matrix structures have in common that they can be arranged into block rows, with the off-diagonal part of each row being representable as a low-rank outer product. As we will discuss in Section 2.2, we can exploit this low-rank outer product structure in order to avoid communication, as long as both the matrix and the preconditioner have similar representations.

1.3.3 AXPYs and dot products

AXPYs and dot products are also communication-bound operations. In the sequential case, the bandwidth cost of reading (and writing, for AXPYs) the vectors dominates the operation, if the vectors do not fit in cache and / or do not start in cache. In the parallel case, AXPYs require no data movement between processors, but may require synchronization to ensure that the vector operation is completed before using the results of the AXPY. Parallel dot

[7] For example, for a sparse incomplete Cholesky preconditioner for the matrix A, computing the incomplete factorization $A \approx L \cdot L^*$ is the construction part, whereas the preconditioner is applied by computing sparse triangular solves with L and L^*.

products require a combination of sequential dot products and a scalar reduction over the processors. The scalar reduction is a latency-bound operation, so this latency cost competes with the bandwidth cost of the sequential dot products.

AXPY and dot products in Krylov methods are often used to orthogonalize a set of vectors. We speak more about this topic in Sections 2.3 and 2.4, where we present new communication-avoiding kernels for orthogonalization of a set of vectors.

1.4 New communication-avoiding kernels

In this section, we summarize our new communication-avoiding kernels. The new Krylov methods presented in this thesis will avoid communication, because they are built out of these new kernels, rather than out of the slower kernels (Section 1.3) used in standard Krylov methods.

1.4.1 Matrix powers kernel

Our first communication-avoiding kernel is called the *matrix powers kernel*. We first presented it in Demmel et al. [77], with subsequent performance optimizations in [78] and [79]. The matrix powers kernel takes a sparse matrix A and a dense vector v, and computes s vectors Av, A^2v, ..., A^sv (or vectors similar to those, as we explain in Section 2.1). For a large class of sparse matrices, the matrix powers kernel can compute these vectors for optimally minimal communication. (Specifically, the graph of the sparse matrix A must be partitionable so that the partitions each have a small surface-to-volume ratio. This is true of many types of sparse matrices, including discretizations of partial differential equations on low-dimensional domains.)

In the sequential case, "minimal" means that both the matrix A and the vectors v, Av, ..., A^sv need to be moved between fast and slow memory only $1 + o(1)$ times. A straightforward implementation of this kernel using s invocations of sequential SpMV would require reading the sparse matrix A from slow memory s times. In the parallel case, "minimal" means that the required number of messages per processor can be reduced by a factor of $\Theta(s)$, so that the number of messages for the matrix powers kernel is the same as $1 + o(1)$ invocations of parallel SpMV. Our work cited above show significant speedups of our matrix powers kernel algorithms over the straightforward approach of s invocations of SpMV, both in performance models and in implementations for various hardware platforms. Since many standard Krylov subspace methods spend most of their runtime in SpMV, replacing SpMV operations with the matrix powers kernel has the potential for large speedups. We have observed speedups in practice for our communication-avoiding KSMs [79], due both to the matrix powers kernel and to the two kernels that follow (Sections 1.4.2 and 1.4.3).

This thesis merely summarizes our work on the matrix powers kernel in Demmel et al. [77, 78, 79]. We summarize the matrix powers kernel in more detail in Sections 2.1 (the unpreconditioned case) and 2.2 (the preconditioned case).

1.4.2 Tall Skinny QR

Our second communication-avoiding kernel is called *Tall Skinny QR* (TSQR). We first presented it in Demmel et al. [75], and subsequently in [79] developed an optimized parallel shared-memory implementation and integrated it with our Communication-Avoiding GMRES algorithm (Section 3.4) to achieve significant speedups on practical matrices.

TSQR is a QR factorization which can be used to orthogonalize groups of vectors. "Tall skinny" refers to the input of the algorithm: a dense $m \times n$ matrix A with $m \gg n$. TSQR differs from the Householder QR factorization (the usual method for tall skinny matrices, implemented in the LAPACK software package [5]) and Modified Gram-Schmidt orthogonalization in that the latter algorithms process the matrix column by column, but TSQR processes the matrix by row blocks. In fact, TSQR works by performing a reduction operation over blocks of the matrix, in which the reduction operator is a smaller QR factorization. Parallel TSQR with P processors requires $\Theta(\log P)$ messages, which is a factor $\Theta(n)$ fewer than Householder QR. In the sequential case with a fast memory of size W, TSQR reads and writes the minimum number of floating-point words ($2mn$), a factor $\Theta(mn/W)$ fewer than standard Householder QR. It performs the minimum number $\Theta(mn/W)$ of block transfers, a factor of $\Theta(n)$ fewer than Householder QR.

The CholeskyQR algorithm we described in Section 1.2.5 communicates no more than TSQR, in both the parallel and sequential cases. However, CholeskyQR in finite-precision arithmetic can produce a Q factor that is not at all orthogonal (see Demmel et al. [75]). In contrast, TSQR is just as accurate as Householder QR: it produces a Q factor of A that is orthogonal to machine precision, regardless of the input matrix.

We developed TSQR specifically for our Krylov methods, as a replacement for Gram-Schmidt or Householder QR in the orthogonalization step. The idea of computing a QR factorization via a reduction over blocks is not entirely novel; for details on related work, see Demmel et al. [75]. Nevertheless, our work [75] generalizes previous TSQR-like factorizations by showing how to map the algorithm to different architectures. It includes a careful communication analysis, and proves that both the sequential and parallel versions of TSQR are communication-optimal, and communicate asymptotically less than existing corresponding algorithms in LAPACK and ScaLAPACK. It also includes new sequential and parallel QR factorizations for general dense matrices, which we call *Communication-Avoiding QR* (CAQR), which use TSQR as the panel factorization. We prove that parallel and sequential CAQR are also communication-optimal, and communicate asymptotically less than existing corresponding algorithms in LAPACK and ScaLAPACK. Furthermore, in [79], our parallel shared-memory implementation of TSQR significantly outperforms the existing parallel shared-memory QR factorization in LAPACK.

In this thesis, we only summarize results on TSQR that we have shown in our previous work [75] and [79]. We summarize TSQR in more detail in Section 2.3.

1.4.3 Block Gram-Schmidt

Our third communication-avoiding kernel is called *Block Gram-Schmidt* (BGS). We first presented it in Demmel et al. [79]. The idea of BGS is to improve the performance of standard (Modified or Classical) Gram-Schmidt, by operating on blocks of columns, instead

of column by column. This results in the algorithm spending most of its time in faster dense matrix-matrix operations, rather than slower vector-vector or matrix-vector operations. BGS is not a new idea, and we present related work in Section 2.4. However, combining BGS with the TSQR algorithm above is a new idea, which we do in Demmel et al. [79]. In Section 2.4, we include a new discussion of reorthogonalization not in that work, in which we summarize related work and propose a new reorthogonalization scheme that combines BGS and TSQR.

1.5 Related work: s-step methods

Our communication-avoiding Krylov algorithms are based on the so-called "s-step" Krylov methods. These break up the data dependency between the sparse matrix-vector multiply and the dot products in standard Krylov methods such as CG and GMRES. The methods do so by first computing the next s vectors in the Krylov subspace basis, then using a combination of scalar recurrences, dot products, and AXPY operations to recover the coefficients from s iterations of the standard Krylov algorithm on which the s-step method is based. Our communication-avoiding methods improve on previous s-step methods in the following ways:

- They avoid communication in both the sparse matrix and the dense vector operations, for general sparse matrices. Our methods do so by using the communication-avoiding kernels summarized in Section 1.4 and discussed in more detail in Chapter 2.
- Unlike previous authors' approaches, we improve numerical stability in the s-step basis computation, without spending additional communication or hindering optimizations in that computation.
- We have figured out the required form of the s-step basis for all types of Krylov methods and s-step bases, also with various forms of preconditioning. Previous work only expressed the form of the preconditioned s-step basis for the monomial basis. Our innovation improves stability and allows us to develop the left-preconditioned communication-avoiding CG algorithm (LP-CA-CG, see Section 5.5).
- Our s-step version of GMRES, which we call CA-GMRES, is free to choose s smaller than the restart length $r = s \cdot t$. Previous s-step versions of GMRES were not. This has the potential to improve numerical stability, convergence, and performance. See Section 3.4 for details.
- Our parallel shared-memory CA-GMRES implementation achieves speedups of 4.3× on 8 cores of an Intel Clovertown processor, and speedups of 4.1× on 8 cores of an Intel Nehalem processor. See Section 3.5 for details.

Van Rosendale [228] first developed an s-step version of CG (though he did not call it this) as a theoretical algorithm with much more potential parallelism (in the PRAM model [101]) than standard CG. Chronopoulos and Gear [57] first coined the term "s-step method" to refer to their own variant of CG. These authors understood that breaking the data dependency between SpMV and vector operations exposes potential optimizations when computing the

s-step basis (the next s basis vectors of the Krylov subspace). However, they did not change how the s-step basis was computed; they used only the "straightforward" approach of s standard SpMV operations, and hoped that a vectorizing and parallelizing compiler could extract further optimizations from their code.

The s-step CG of Chronopoulos and Gear had serious numerical stability problems, especially as s increased. In fact, $s > 5$ was enough to cause convergence failures, in the matrices they tested. This was not merely a side effect of details in their algorithm; it was also true for Van Rosendale's algorithm, as Leland [166] discovered in his implementation. Later authors interested in s-step versions of GMRES would discover that the source of instability was the use of the monomial basis v, Av, A^2v, ..., A^sv, which rapidly becomes numerically rank deficient in finite-precision arithmetic. We will discuss this work below.

Meanwhile, Walker [238] developed a new version of GMRES that also uses the s-step technique.[8] However, Walker is motivated by numerical stability concerns, not by performance. The standard version of GMRES [209] uses Modified Gram-Schmidt (MGS) to orthogonalize the Krylov basis vectors. At the time, the gradual loss of orthogonality in MGS was considered a potential source of inaccuracy when using GMRES to solve linear systems. Walker instead used Householder QR to orthogonalize the basis vectors. (Now it is known that despite the loss of orthogonality due to MGS, MGS-based GMRES is no less accurate than Walker's GMRES based on Householder QR. See Paige et al. [186] for a proof.) Using Householder QR to orthogonalize a collection of vectors requires having all the vectors available at once, unlike MGS, which can orthogonalize each successive basis vector as soon as it is computed from the sparse matrix-vector product in GMRES. Thus, Walker adopted the s-step approach entirely out of numerical stability concerns. He computed the s-step basis in the same straightforward way (s invocations of SpMV) that previous authors did in s-step CG. It seems also that Walker did not know of the work of Van Rosendale or Chronopoulos and Gear, as he did not cite it.

The characteristic feature of Walker's "Householder GMRES" is that the restart length equals the s-step basis length. This differs from the s-step CG algorithms of Van Rosendale and Chronopoulos and Gear, which do not need to restart at all. Later s-step GMRES algorithms, such as those by Joubert and Carey [144, 145], Bai et al. [19], and de Sturler and van der Vorst [73], all have this limitation. Our CA-GMRES algorithm does not have this limitation; we are free to choose the restart length arbitrarily larger than the s-step basis length. See Chapter 3 for details.

Walker also later realized that his use of the monomial basis was causing numerical instability. Hindmarsh and Walker [135] corrected this partially by scaling each vector in the monomial basis by its norm. This prevented overflow or underflow if $\|A\|$ is much larger resp. smaller than one, but did not entirely solve the numerical stability problems. Later s-step GMRES authors attacked the stability problem by using a different basis other than the monomial basis. For example, Joubert and Carey [144, 145] used a basis of Chebyshev polynomials of the first kind, and Bai et al. [19] a basis of Newton polynomials. The latter authors avoid the overflow / underflow problems observed by Hindmarsh and Walker, by using the same technique: they scale each s-step basis vector in turn by its norm as it is

[8] Walker's 1985 work cited a 1983 technical report by Saad and Schultz introducing the GMRES algorithm; the latter report later developed into the 1986 paper [209].

computed. Doing so reintroduces the same data dependency between SpMV and vector operations that Van Rosendale and Chronopoulos and Gear had sought to remove, and thus hinders communication-avoiding optimizations. In Section 7.5, we present a novel approach involving equilibration or balancing of the matrix A. It avoids underflow or overflow, helps with numerical stability in practice, and requires only a one-time computation per matrix for the cost of a few SpMV operations. In Section 3.5, we show experiments that demonstrate the value of this approach for improving numerical stability.

The s-step GMRES authors mentioned above differed from Walker, in that improving the performance of GMRES was their main motivation. Bai et al. suggested that performance optimizations in s-step GMRES should focus on the QR factorization, since GMRES often spends most of its time in the vector operations, rather than in SpMV. Erhel [94] takes up this challenge by applying a new parallel QR factorization, RODDEC, that optimizes communication on a ring network of parallel processors. RODDEC requires a number of messages proportional to the number of processors P, so it does not minimize the number of messages for a general network of processors. Joubert and Carey [145] use an algorithm we call CholeskyQR to orthogonalize the basis vectors in their s-step GMRES algorithm. As we discuss in Section 2.3, CholeskyQR may be implemented in a way that minimizes communication requirements both between processors and between levels of the memory hierarchy. Joubert and Carey were only motivated by parallel considerations. Also, the authors developed what we call a "matrix powers kernel" (see Section 2.1) in order to reduce the number of messages when computing the s-step basis. Their version of the kernel only works for 2-D regular meshes; ours works for general sparse matrices, both in theory and in practice. Furthermore, Joubert and Carey's version of the matrix powers kernel only minimizes the number of messages in parallel; ours minimizes the number of messages in parallel, and the number of words transferred between levels of the memory hierarchy. See Section 2.1 for details.

Toledo [222] also optimizes the computation of the s-step basis, in his s-step CG algorithm. Unlike Joubert and Carey, he is concerned not with parallel optimizations, but with reducing the number of words transferred between levels of the memory hierarchy. His s-step basis generation algorithm works for general sparse matrices, even for those with more complicated structures than our matrix powers kernel efficiently supports (see Section 2.1). However, he only implements and benchmarks the algorithm for tridiagonal matrices. Our matrix powers kernel implementation works and achieves speedups over the straightforward approach (of s invocations of SpMV) for general sparse matrices.

Preconditioning was perhaps the Achilles' heel of previous work in s-step Krylov methods. We are the first, as far as we know, to explain why this was so difficult, especially for s-step bases other than the monomial basis. We do so in Section 4.3. Previous work either restricts itself to polynomial preconditioning (see e.g., Leland [166]), to block diagonal preconditioners that require no communication between parallel processors, or to split preconditioning that preserves any symmetry in the matrix A (see e.g., Chronopoulos and Gear [56]). None of these authors show how the s-step basis changes when preconditioning is introduced, nor do they show how to avoid communication in the preconditioner application (other than to use block diagonal preconditioners).

We do show in Section 4.3 how the s-step basis changes when preconditioning is introduced. We also draw an analogy between this approach and two-sided Krylov methods like

nonsymmetric Lanczos and BiCG, which helps us suggest an approach in Chapter 6 for developing the latter. In addition, we present in Section 2.2 two preconditioning approaches that avoid communication in a way either compatible with or analogous to our matrix powers kernel. However, we leave the development of effective communication-avoiding preconditioners for future work.

Table 1.1 gives a concise summary of the prior work we found in which one or more s-step Krylov subspace methods were developed. The table indicates not only the algorithms developed in each work, but also the s-step basis type(s), the forms of preconditioning (if any) used, and whether the authors present or use a form of optimized matrix powers kernel (see Sections 2.1 and 2.2) or TSQR (see Section 2.3). (Some of the iterative methods shown in the table, such as Orthomin and Orthodir, are not so commonly used anymore. Greenbaum [120, Section 2.4] suggests some reasons why; the conventional versions of these algorithms converge more slowly than GMRES, require more storage, or both.) The table includes footnotes, which we explain here:

1. "Local" preconditioning refers to a block diagonal preconditioner, that is, which is local to each subdomain (and thus requires no parallel communication).

2. Actually the paper includes s-step versions of MINRES, CR (conjugate residual), GCR (generalized conjugate residual), and Orthomin(k).

3. Matrix powers kernel only for 2-D regular meshes, not for general sparse matrices.

4. The paper does include a more efficient parallel QR decomposition (called RODDEC) for ring communication networks. However, unlike TSQR, it does not minimize latency in the fully general parallel case.

5. Toledo gives a "matrix powers kernel"-like sequential algorithm, based on "blocking covers," for minimizing memory traffic between fast and slow memory. The algorithm covers general sparse matrices and even more general constructs. However, Toledo does not give a general implementation; the implementation for which he presents benchmarks is for tridiagonal matrices.

1.6 Related work on avoiding communication

In this section, we discuss related work which avoids communication in Krylov methods, in different ways than the methods discussed in this thesis. We summarize the advantages and disadvantages of each technique, and explain why we chose s-step methods over that approach as the basis for our algorithms. We begin with the most conservative of the methods, and conclude with the most aggressive. "Conservative" here suggests one or both of the following:

- Rearrangements of existing methods, with the same convergence behavior in exact arithmetic

- Little algorithmic or implementation effort, beyond that of existing implementations

Citation(s)	Algorithms	Basis	Precond	Akx?	TSQR?
[100]	Steepest descent	monomial	none	N	N
[228]	CG	monomial	polynomial	N	N
[166]	CG	monomial	polynomial	N	N
[239]	GMRES	monomial	none	N	N
[57]	CG	monomial	none	N	N
[56]	CG	monomial	polynomial, local (1)	N	N
[58]	Orthomin,GMRES	monomial	none	N	N
[55]	MINRES (2)	monomial	none	N	N
[151]	Symm. Lanczos	monomial	none	N	N
[152]	Arnoldi	monomial	none	N	N
[71]	GMRES	Chebyshev	none	N	N
[144, 145]	GMRES	Chebyshev	none	Y (3)	N
[153]	Nonsymm. Lanczos	monomial	none	N	N
[19]	GMRES	Newton	none	N	N
[94]	GMRES	Newton	none	N	N (4)
[73]	GMRES	Chebyshev	general	N	N
[222]	CG	monomial, recycled	polynomial	Y (5)	N
[60]	GCR,Orthomin	monomial	none	N	N
[59]	Orthodir	monomial	none	N	N

Table 1.1: "Citation(s)": citations of prior work on s-step Krylov subspace methods. "Algorithms": the type(s) of Krylov subspace methods which those author(s) developed. For example, van Rosendale [228] developed an s-step variant of CG, so we list "CG" in its "Algorithms" field. "Basis": the type(s) of s-step bases used (not just suggested) in the paper: either monomial, Newton, Chebyshev, or "recycled" (reusing the coefficients that orthogonalized the last set of s-step basis vectors). "Precond": the type(s) of preconditioners used in the paper. If "Akx?" is Y, it means that the paper includes an communication-avoiding matrix powers kernel (see Section 2.1). This does not include just exposing the kernel and hoping an optimizing compiler will make it go faster. If "TSQR?" is Y, it means that the paper includes an optimized Tall Skinny QR factorization (see Section 2.3). We mean by this a QR factorization as stable as Householder QR, which uses a single reduction to factor the matrix. For footnotes (e.g., "(1)"), see text.

"Aggressive" here suggests one or both of the following:

- Significant algorithmic or implementation effort
- Based on existing methods, but converge differently than they do, even in exact arithmetic

Section 1.6.1 begins with algorithms that replace the Modified Gram-Schmidt (MGS) orthogonalization method in the standard version of Arnoldi iteration with other orthogonalization methods that require fewer synchronization points. Section 1.6.2 describes the block Krylov methods, which use a different approach than our Krylov methods to avoid communication in the sparse matrix-vector products. In Section 1.6.3, we discuss Chebyshev iteration, which communicate less per iteration than other Krylov methods because they require no inner products. In Section 1.6.4, we present some existing approaches to avoiding communication in multigrid. Finally, Section 1.6.5 summarizes asynchronous iterations. These relax the synchronization constraints of existing block relaxation iterations, resulting in nondeterministic intermediate results in exact arithmetic, but eventual convergence under certain conditions.

1.6.1 Arnoldi with Delayed Reorthogonalization

Hernandez et al. [128] propose new rearrangements of Arnoldi iteration that use Classical Gram-Schmidt (CGS) instead of Modified Gram-Schmidt (MGS) to orthogonalize the current basis vector against all the previous basis vectors. In standard Arnoldi (Algorithm 18 in Section 3.1), iteration j of the algorithm first computes a vector $w_j := Aq_j$, and then uses MGS orthogonalization to orthogonalize w_j against all the previous basis vectors q_1, q_2, ..., q_k. The MGS orthogonalization step requires j dot products and AXPY operations in iteration j. In parallel, each of those dot products requires a global synchronization between processors. In contrast, replacing MGS orthogonalization with CGS orthogonalization can be done with two dense matrix-vector products. In parallel, this requires only two global synchronizations. (The number of floating-point operations is no different and the amount of data movement between levels of the memory hierarchy is not asymptotically different.) Since CGS is less accurate than MGS in finite-precision arithmetic, the authors do reorthogonalization at every iteration, even though this entails redundant computation and may also involve extra communication (as compared with standard Arnoldi without reorthogonalization). The most aggressive algorithm Hernandez et al. present, "Arnoldi with Delayed Reorthogonalization" (ADR), avoids any extra synchronization points due to reorthogonalization. It does so by mixing together the work of the current, previous, and next iterations, in a way analogous to software pipelining.

ADR requires fewer synchronization points in parallel for the vector operations than standard Arnoldi iteration. Some number s iterations of either algorithm add s basis vectors to the Krylov subspace, but s iterations of ADR require only $2s$ synchronization points for the vector operations, whereas s iterations of standard Arnoldi require $s^2/2 + \Theta(s)$ synchronization points for the vector operations. However, our communication-avoiding version of Arnoldi, CA-Arnoldi (see Chapter 3), requires only a constant number of synchronization points in parallel for s iterations. ADR only uses BLAS 2 - type operations (dense matrix-vector products) when orthogonalizing the basis vectors, but CA-Arnoldi uses BLAS 3 -

type operations (dense matrix-matrix products) when orthogonalizing the basis vectors. As we show in Sections 2.3 and 2.4, this means that CA-Arnoldi requires less data movement between levels of the memory hierarchy than ADR (or standard Arnoldi). Furthermore, CA-Arnoldi avoids communication in the sparse matrix operations as well, which ADR does not. In fact, CA-Arnoldi performs the same work as s iterations of ADR (or standard Arnoldi), but the sparse matrix operations communicate only as much as a single sparse matrix-vector product.

ADR may also be less accurate than either standard Arnoldi or our CA-Arnoldi. The ADR update formula sacrifices some of the value of the reorthogonalization, because it imposes assumptions about the orthogonality of the Arnoldi basis vectors that standard Arnoldi does not impose. While some number s iterations of ADR is equivalent to s iterations of standard Arnoldi in exact arithmetic (not considering the software pipelining), the update formula can affect the quality of the algorithm in finite-precision arithmetic. This is because iterative eigensolvers are sensitive to the quality of the orthogonalization. CA-Arnoldi uses a QR factorization as accurate as Householder QR (see Section 2.3) to orthogonalize the Krylov basis vectors, so it should produce basis vectors at least as orthogonal as standard Arnoldi and possibly better.

Hernandez et al. implemented several Arnoldi variants, including ADR, in parallel on three different platforms:

- A cluster of 55 dual-processor 2.8 GHz Pentium Xeon nodes, using only one processor per node, with an SCI interconnect configured as a 2-D torus
- An IBM SP RS/6000 system with 380 nodes and 16 processors per node
- An IBM JS20 cluster of 2406 dual processor nodes with 2.2 GHz PowerPC 970FX processors at 2.2 GHz with Myrinet interconnect, using only one processor per node

In the resulting implementation, for many (but not all) of the matrices tested by the authors, each iteration of ADR usually takes less time than each iteration of standard Arnoldi. Thus, if one counts only the time to execute a certain number of iterations, ADR often runs faster than standard Arnoldi. However, the approximate eigenvalues computed by ADR fail to converge in some cases where standard Arnoldi's approximate eigenvalues converge. The authors test their algorithms by computing the 10 largest eigenvalues (at a tolerance of 10^{-7}, restarting after every 50 iterations) of every matrix from the Harwell-Boeing and NEP collections (a total of 210 matrices). ADR fails for 40 of those matrices. Unfortunately the authors show convergence as a histogram over the set of matrices. This makes no sense statistically, since the order of the matrices is arbitrary and the set of matrices is also more or less arbitrary. It also obscures valuable information about the types of matrices on which ADR fails.

Our experiments with communication-avoiding Arnoldi confirm the value of focusing on orthogonalization as Arnoldi's most expensive kernel, which is what Hernandez et al. do. Our CA-Arnoldi algorithm requires less communication than ADR in both in the sparse matrix operations and the vector operations, however.

1.6.2 Block Krylov methods

Block Krylov methods are related to standard Krylov subspace methods. The latter work with one or more Krylov subspaces, such as

$$\mathcal{K}_k(A, v) = \text{span}\{v, Av, \ldots, A^{k-1}v\},$$

where v is a single vector. In contrast, block methods work with one or more "block Krylov subspaces," such as

$$\text{BlockKrylov}(A, B, k) = \text{span}\{B, AB, \ldots, A^{k-1}B\},$$

where B is a dense matrix whose columns are a collection of (linearly independent) vectors. (The number of columns of B is generally much smaller than the number of rows.) Computing such subspaces involves a different kernel, *SpMM*, which multiplies a sparse matrix A, with multiple dense vectors arranged in a dense matrix B. The straightforward way of implementing SpMM would invoke SpMV once for each column of B, so that the matrix A must be read s times if B has s columns. However, the SpMM kernel can be implemented in a communication-avoiding way, so that the matrix A is read only once, for all columns of B. This so-called *multivector optimization* is described by Vuduc [235], among others. Block Krylov methods differ in exact arithmetic from their analogous non-block methods.

Block methods originated as modifications of Lanczos iteration for finding eigenvalues of large sparse symmetric matrices [64, 223, 116]. There, they reduce the number of iterations required to resolve a cluster of nearby eigenvalues. One typically must choose the number of dense vectors in the resulting SpMM kernel according to the expected number of eigenvalues in the cluster (see e.g., [16] for details), rather than according to performance considerations. For some applications, block methods are necessary for computing all the desired eigenvalues to the desired accuracy; they are not (just) a performance optimization.

There are also block Krylov algorithms for solving systems of linear equations with multiple right-hand sides. O'Leary first extended the block sparse eigenvalue solvers above to a CG-like iteration for this case [182], and others later extended and popularized the technique (see e.g. [175, 20]). O'Leary's motivation was to solve sparse linear systems with multiple right-hand sides efficiently. Ordinary Krylov methods can only solve for one right-hand side at a time; the iterations must be repeated for each right-hand side. However, block methods can solve for all the right-hand sides "at once." It is not guaranteed that this will require fewer iterations than if one were to solve for each right-hand side in sequence. However, if the block method does require fewer iterations, then it should perform better. This is because the cost of reading the sparse matrix in the sparse matrix-vector product often dominates the cost of an iterative method (especially for short-recurrence methods like CG), so the additional dense vectors in the SpMM kernel may come "for free."

Block Krylov methods were originally formulated for multiple right-hand sides. This is an important case but by no means all-inclusive. For example, a structural engineering problem may involve only one right-hand side: the load, which is fixed and part of the design requirements of the structure. Block Krylov methods have been extended to handle the single right-hand side case; see e.g., Baker [20]. Baker does this by fixing a restart length (e.g. 30 iterations before restart), and using an error term from one restart sequence to augment the

right-hand side for the next restart sequence. Each restart sequence augments the right-hand side by one additional vector. This technique prevents stagnation of the convergence rate by preserving basis information from previous restart sequences. This means that the number of right-hand sides requires $s-1$ restart cycles to reach s vectors. Baker ensures that there are always s right-hand sides by filling in those which have not yet been computed with additional random (but linearly independent) vectors. The relationship between these random vectors and the convergence rate is unclear. Alternately, one could omit the random vectors and just increase the number of dense vectors in SpMM by one with each restart cycle, but this would entail a high start-up cost before the multivector optimization can take full effect. If the preconditioner is effective enough, the multivector optimization may never take full effect. Baker also omits the deflation step, which the algorithm requires for correctness in certain corner cases but did not seem to affect convergence for the problems tested. Deflation may require an expensive rank-revealing decomposition. In all cases, it is hard to estimate *a priori* how the convergence rate will improve with the number of artificially-added right-hand sides, so the extra computation may not pay off. Wasted redundant computation consumes execution time, and also space in memory (to store the additional vectors in each block that are not contributing sufficiently to improving the convergence rate). It may also consume more energy, though determining whether this is the case is future work.

In contrast, our communication-avoiding iterative methods are mathematically equivalent to the ordinary Krylov iterations from which they were derived, and have similar convergence behavior in practice. That makes it easier to predict the performance benefit of the redundant computation which they perform. Our algorithms also run at full optimization after at most s or $2s$ steps of a standard iteration (to get eigenvalue estimates for basis construction – see Chapter 7 for details).

The multivector optimization for SpMM does have significant advantages over our matrix powers optimizations described in Section 2.1 and 2.2.

- The multivector optimization improves performance for any sparse matrix structure, by amortizing the cost of reading each entry of the sparse matrix. The matrix powers optimizations work best when the sparse matrix can be partitioned effectively into partitions with a low surface-to-volume ratio. However, such matrices include (but are not restricted to) finite-element and finite-difference discretizations of low-dimensional domains, which is an important class of problems.

- The multivector optimization *in itself* (not counting the convergence of the block Krylov method) involves no redundant computation over the straightforward way of implementing SpMM. The matrix powers kernel generally does require redundant computation over the straightforward approach (of invoking SpMV s times on a single vector v). In contrast, the matrix powers kernel may require redundant computation.

- The multivector optimization generally requires less data reorganization than the matrix powers kernel. The latter requires significant preprocessing to extract and possibly replicate boundary layers between subdomains. In contrast, SpMM performs better when the source and destination blocks are stored in row order (see e.g., [235, 20]), which requires little preprocessing of the data.

We consider block Krylov methods an independent approach to that of the algorithms in our thesis. In fact, one could consider combining s-step methods with block Krylov methods. We have not done so in this thesis, because it would require developing new algorithms and demonstrating their numerical stability and efficiency. We do not yet understand how the s-step basis construction would interact with block methods and the possible need for deflation. Also, the matrix powers kernel would need to be modified to handle multiple source and destination vectors, and performance tuning would need to be done on the resulting implementation. We leave all of this for future work.

Another area in which our communication-avoiding kernels could be useful is in the deflation part of block Krylov methods. Many block methods require a rank-revealing decomposition to ensure that all the vectors in a block are linearly independent. One such decomposition used in practice is QR with column pivoting, which requires a lot of communication and data movement. The TSQR kernel could be used as the first step in a rank-revealing QR factorization, as suggested by da Cunha et al. [67] and as we explain in Section 2.4.8. Investigating this is future work, but probably not as hard a task as that described in the previous paragraph.

1.6.3 Chebyshev iteration

Chebyshev iteration (see e.g., [23]) is a Krylov subspace method widely considered to require less communication, as it computes no inner products. It only performs one sparse matrix-vector product, some vector AXPY operations, and (possibly also) one preconditioner application per iteration. Chebyshev iteration is an iterative method but not an s-step method. It is not to be confused with the Chebyshev s-step basis presented in Section 7.3.3, even though both exploit the properties of Chebyshev polynomials to achieve their aims.

Chebyshev iteration does not compute inner products because it computes the coefficients it needs from a bounding ellipse of the spectrum of the matrix A (or the preconditioned matrix, if a preconditioner is used). The bounding ellipse must be known in advance and the bounds must be fairly tight, otherwise the iteration may fail to converge. This creates an apparent chicken-and-egg problem, as the eigenvalues of a sparse matrix are generally not known in advance, and the typical way to compute them is via a Krylov subspace method. Manteuffel [172] developed an adaptive algorithm which estimates a bounding ellipse from eigenvalue approximations computed via a few steps of the power method, and Ashby [11] implemented this algorithm. Elman, Saad, and Saylor [92] use Arnoldi (in its GMRES form) to compute those eigenvalue approximations, in a hybrid algorithm that alternates between slower but more effective GMRES, and faster but less effective Chebyshev iteration. (Chebyshev iteration's residuals cannot converge more rapidly than the GMRES residuals, due to the latter's minimum residual property.) Calvetti, Golub, and Reichel [45] instead compute modified moments from inner products of residual vectors coming from the Chebyshev iteration itself. In all cases, just a few steps often suffice to compute a sufficiently accurate bounding ellipse so that Chebyshev iteration converges well.

Chebyshev iteration's performance advantage is its lack of inner products, but in terms of convergence this is its disadvantage. Inner products enable orthogonalization, whether implicit (as in symmetric Lanczos) or explicit (as in GMRES). Orthogonalization in Krylov

methods often results in favorable minimization properties of some measure of the error at each iteration. The inner products in a Krylov method also help approximate the eigenvalues of the matrix, and convergence of an eigenvalue approximation in Krylov methods is often associated with superlinear convergence when solving linear systems (see e.g., [227] in the case of GMRES, and in general [23]). Furthermore, Chebyshev iteration may diverge if the elliptical bound of the eigenvalues is not accurate or if an ellipse is not a good fit to the spectrum of the matrix. The hybrid approaches mentioned in the previous paragraph cannot help with the latter problem.

One option for Chebyshev iteration, which as far as we can tell previous authors have not considered, is to use the matrix powers kernel to implement the algorithm. This requires no modification to the matrix powers kernel, as it is already designed to work with polynomials that satisfy a three-term recurrence (see Section 2.1). This "s-step Chebyshev iteration" could be combined with the hybrid approaches of previous authors, in particular with the hybrid Chebyshev / GMRES algorithm of Elman, Saad, and Saylor [92]. Replacing GMRES with our CA-GMRES algorithm (Section 3.4) would allow the resulting implementation to exploit the same matrix powers kernel for accelerating both the Chebyshev iteration and GMRES phases of the computation. We leave this "hybrid s-step Chebyshev iteration" suggestion for future work.

1.6.4 Avoiding communication in multigrid

Multigrid (see e.g., Briggs et al. [38]) is a popular approach for solving or preconditioning linear systems. Multigrid methods are most commonly applied to the discretizations of certain partial differential equations, but can be applied to some other problems as well. These algorithms can solve certain $n \times n$ linear systems in $O(n)$ time, which is theoretically optimal. Nevertheless, multigrid performance is usually dominated by communication: both memory traffic and messages in parallel. Some authors have developed new techniques to reduce memory traffic in multigrid. In this section, we will present two such techniques: one that avoids memory traffic for repeated smoother applications, and another that avoids memory traffic for multiple applications of an entire multigrid cycle.

Techniques like the matrix powers kernel we summarize in Section 2.1 can be used to avoid data movement for many operations that "look like" sparse matrix-vector multiplication. The smoother application phase of multigrid can be optimized in this way, depending on the properties of the matrix and the particular smoother. Douglas et al. [89] propose this for geometric multigrid, where the smoother is a relaxation operation on a regular grid. Strout [219] extends this to unstructured grids. However, the central idea of multigrid is that smoothing quickly dampens only the high-frequency components of the error, beyond which coarsening is needed to address lower-frequency components. That means we can expect diminishing returns for repeated smoother applications. In practice, more than two or three repetitions may not improve convergence. This limits the possible speedups when using the strategy of Douglas et al. and Strout, even though significant speedups are still possible.

A more aggressive approach involves reducing the memory traffic required by an entire multigrid cycle, when applying it multiple times to the same vector. Toledo, in his PhD thesis [222], developed a rearrangement of multigrid that avoids communication in this way.

However, the benefit scales only as the fifth root of the cache size. Furthermore, the implementation that would be required would be very complicated and might have significant overhead. (Toledo only implemented these techniques for a regular 1-D mesh.)

1.6.5 Asynchronous iterations

Asynchronous iterations (also called *chaotic iterations*), are a class of iterative methods that relax parallel synchronization constraints that follow from data dependencies in existing iterative methods. This results in new iterative methods that may converge differently in exact arithmetic, but it avoids the global communication required by synchronization. It also may allow the iteration to make progress even if some messages between processors arrive irregularly or not at all, under certain conditions. Chazan and Miranker [51] first developed asynchronous versions of relaxation methods, and Baudet [24] continued the development of theory and implemented asynchronous versions of Jacobi and Gauss-Seidel on a new shared-memory parallel computer. Bru et al. [40] combine relaxation and matrix splitting techniques to form new asynchronous methods. Later work (see e.g., [80, 249]) focuses mainly on convergence studies and proofs. For a survey, including algorithms for solving both linear (nonsingular and singular) and nonlinear equations, see e.g., Frommer and Szyld [106]. Bahi et al. [14] have an extensive bibliography, and promote the technique for grid computing, where message latencies may be irregular.

The simplest example of an asynchronous method is performing Gauss-Seidel iteration in which each processor is responsible for updating its own block of solution vector entries, but relaxing the requirement in ordinary Gauss-Seidel that all processors have finished updating their components of the solution before continuing to the next iteration. (For shared-memory programming, this means that the asynchronous version of Gauss-Seidel deliberately introduces a race condition.) The effect is that different processors may have different "views" of neighboring processors' components of the solution. Different processors see those components' values as they would be at different iterations. Under some conditions (for example, that at least some messages "get through" – see [249]), it is possible to prove that the algorithms will nevertheless always converge for a certain class of matrices. These proofs work by analyzing the method as a fixed-point iteration. As long as the matrix satisfies certain properties and as long as the processors do not lag one another by too many iterations, the iteration is a contraction and thus will eventually converge to the correct solution. More complicated asynchronous iterations involve relaxations like block Jacobi, domain decompositions like additive Schwarz, or combinations of relaxation and matrix splitting techniques.

Asynchronous iterations suffer perhaps from the lack of an accurate name. Authors developing such algorithms use either "asynchronous" or "chaotic" to describe them. One might also use the term "nondeterministic," as the results of each "iteration" are nondeterministic in exact arithmetic. However, all three of these words have overloaded meanings.

- "Asynchronous" may also be used to describe algorithms or implementations that schedule computations dynamically and without global synchronization points, yet produce deterministic results in exact arithmetic. In many cases, dynamic scheduling can help keep processors busy when data dependencies in a statically scheduled implementation would leave them idle. Under this definition, an implementation of

CG or GMRES with dynamically scheduled parallel SpMV and dot products would be called "asynchronous." In contrast, asynchronous iterations (as described above) may not produce deterministic results in exact arithmetic. Many authors, such as Frommer and Szyld [106], seem to confuse dynamic scheduling (where the output is deterministic in exact arithmetic, even if the task graph is not), with nondeterminism.

- "Chaotic" masks the intent of asynchronous iterations: they are intentionally *not* chaotic, because their steady-state behavior is predictable (they eventually converge). In mathematics, a chaotic process is deterministic, but highly sensitive to initial conditions, so that its future state is impossible or very difficult to predict. That is the opposite of asynchronous iterations, which are implemented assuming nondeterminism, but converge to a predictable result (in exact arithmetic) nevertheless.

- "Nondeterministic" does describe the behavior of asynchronous iterations in exact arithmetic. However, in floating-point arithmetic, even the result of summing a list of n numbers may be nondeterministic, if the order of summands is not guaranteed. This is usually the case in parallel implementations. For example, summing a list of n floating-point numbers may produce different bitwise results on different numbers of processors, because the shape of the tree used for the reduction affects the order of summands. Even if the number of processors is fixed, many libraries of parallel primitives, or parallel compilers, do not guarantee bitwise determinism. Guaranteeing it generally requires an extreme performance penalty.

We have chosen to call this class of algorithms "asynchronous" as the least confusing word used by authors studying this area.

Asynchronous iterations may be the only workable approach to parallelism under certain conditions of unreliable or irregular-latency communication. However, we chose not to pursue this approach for the following reasons.

1. Dynamic or even just more intelligent static task scheduling may be able to avoid many global synchronizations without needing to resort to nondeterminism in the iteration itself.

2. Making progress in an iterative method usually requires some communication (unless the problem decouples, which is a structure that is easy to recognize). Relaxing the order of updates does not relax the requirement that information passes between subdomains.[9] Our communication-avoiding approach passes that information in a more efficient way, without changing the algorithm's output in exact arithmetic.

3. Asynchronous methods tend to be based on slow relaxation-type iterations, like Jacobi or Gauss-Seidel or simple domain decomposition. This could be because proving that

[9] One can illustrate this dramatically by solving the 1-D Poisson equation with homogeneous Dirichlet boundary conditions and with a forcing term that is zero everywhere but of large magnitude at a point in the center of the domain. If the equation is solved with second-order central differences on a domain of n equally spaced points, the resulting matrix A will be tridiagonal. Thus, a nonpreconditioned iterative method with a zero initial guess that performs sparse matrix-vector products with A will require at least $n/2 + o(n)$ of those sparse matrix-vector products to produce an approximate solution with (correctly) nonzero elements near the boundary.

they converge with asynchronous updates requires resorting to a fixed-point iteration model, and this is too general of a model for describing good ways to solve linear systems. Finite linear algebra has a rich structure of orthogonality and subspaces, which most Krylov subspace algorithms exploit to their benefit. This could be why few recent authors in this area risk comparing the convergence of asynchronous methods with their synchronous counterparts.

4. Nondeterminism is not the same as randomness. Biases in the way the computer hardware or software resolve race conditions could affect convergence of the asynchronous iterations. Introducing randomness to defeat these biases would add complexity to the algorithm. It may also entail slowing down some messages, which defeats the goal of improving performance under variable message arrival times.

The most important reason why we chose not to pursue asynchronous methods further is that they are perhaps ahead of their time. Computer scientists develop abstractions to help them reason about performance and correctness, and build algorithms atop those abstractions. If the abstractions deviate slightly from reality, they deploy hardware (such as error-correcting memory) or software (such as message-passing libraries that resend lost messages) to make reality conform to the abstractions. However, when the abstractions deviate severely from reality, computer scientists must form new abstractions, and develop new classes of algorithms to conform. In the past, one such abstraction was that floating-point operation counts were good indicators of the performance of linear algebra codes, and communication was free.[10] Communication-avoiding algorithms came about because this abstraction deviated too severely from reality, in the context of Krylov subspace methods, to be useful for predicting performance. Another such abstraction, in the context of parallel computing, is that messages always arrive and that message latencies do not vary much. Today's typical parallel machines have hardware and software support for this abstraction. Maintaining this abstraction for future parallel machines may be impossible. First, the large number of processors for the desired exascale machines will increase mean time to failure rates enough to make it likely that individual messages may not get through, or may be delayed or not sent at all (for example if a node crashes during a computational phase and must be restarted). Second, the shrinking feature sizes on a single CPU die may introduce the same problem on the smaller scale of a single node of a cluster. Thus, unreliability at multiple scales of parallelism may ultimately force the abandonment of the abstraction of determinism. However, parallel computers have not reached this point yet, and it is not clear when they will. We have chosen for now to focus on reducing communication in a non-chaotic way.

1.6.6 Summary

In Section 1.6.1, we described "Arnoldi with Delayed Reorthogonalization" (ADR), which is a rearrangement of standard Arnoldi iteration for solving sparse nonsymmetric eigenvalue problems. ADR has fewer synchronization points, but the resulting Arnoldi basis vectors

[10] The PRAM model of parallel performance [101] is one such abstraction, which assumes that all processors share a large pool of memory and can access any entry in it infinitely fast.

appear to lose orthogonality more quickly in finite-precision arithmetic than in standard Arnoldi. Our CA-Arnoldi algorithm (Section 3.3) achieves the same performance goal of reducing the number of synchronization points in Arnoldi, yet it produces Arnoldi basis vectors that should lose their orthogonality in finite-precision arithmetic much more slowly than the basis vectors in ADR. More experimentation is required, however, to see what the performance differences are.

In Section 1.6.2, we discussed block Krylov methods for solving linear systems and eigenvalue problems. These methods avoid communication using the SpMM (sparse matrix times multiple dense vectors) kernel. However, they converge differently than their corresponding non-block methods in exact arithmetic. We have chosen the s-step approach to avoiding communication, but consider block Krylov methods an equally valuable direction of research. Future work may involve combining the two techniques.

In Section 1.6.3, we mentioned Chebyshev iteration, which communicates less per iteration than other Krylov methods because it requires no inner products or other global reductions. The convergence of Chebyshev iteration is generally not competitive with that of other Krylov methods. However, as a smoothing step in multigrid, optimizing Chebyshev iteration using the matrix powers kernel (Section 2.1) may be useful.

Next, in Section 1.6.4, we considered techniques for avoiding communication in multigrid. The theoretical benefits of these techniques appear small; for example, they can reduce the number of words transferred between levels of the memory hierarchy by at best a factor of the fifth root of the fast memory size. We have not found parallel versions of these algorithms. Also, they may be challenging to implement.

Finally, in Section 1.6.5, we described asynchronous iterations, which relax synchronization constraints of existing block relaxation methods (like block Jacobi) in order to avoid synchronization overhead and permit continuing the method if messages from remote processors are lost or delayed. We consider these methods more valuable for the latter reason of improving robustness, than for improving performance. As hardware evolves and likely becomes less reliable, such algorithmic approaches to robustness may become more important. We do not investigate these methods further in this thesis.

1.7 Summary and contributions

The rest of this thesis is organized as follows.

- Chapter 2 gives algorithms and implementations for our new communication-avoiding kernels: the matrix powers kernel, TSQR, and BGS.

- Chapter 3 explains our communication-avoiding versions of Arnoldi iteration and GMRES for solving nonsymmetric eigenvalue problems resp. linear systems (with and without preconditioning). It also shows the results of numerical and performance experiments for our Communication-Avoiding GMRES implementation.

- Chapter 4 gives our communication-avoiding versions of (symmetric) Lanczos iteration, both for standard symmetric eigenvalue problems $Ax = \lambda x$ and for generalized symmetric eigenvalue problems $Ax = \lambda Mx$ (where A and M are both symmetric).

- Chapter 5 shows our communication-avoiding versions of the Method of Conjugate Gradients (CG) for solving symmetric positive definite linear systems, with and without preconditioning.
- Chapter 6 suggests an approach for developing communication-avoiding versions of nonsymmetric Lanczos iteration (for solving nonsymmetric eigenvalue problems $Ax = \lambda x$) and the Method of Biconjugate Gradients (BiCG) for solving nonsymmetric linear systems.
- Chapter 7 explains in detail the numerical stability aspects of the matrix powers kernel, such as the choice of s-step basis, what it means for that basis to be "well conditioned," and how scale the matrix A in order to avoid underflow or overflow in the basis vectors.
- Appendix A shows the experimental platforms on which we ran performance experiments.
- Appendix B gives details on the test problems used in our numerical and performance experiments.
- Appendix C shows mathematical derivations that are too long and / or tedious to belong in the main text.
- Finally, Appendix D explains some mathematical terms in detail, that are used briefly in this thesis but are tangential to the main argument where they appear.

This thesis makes the following contributions. Our algorithms are based on the so-called "s-step" Krylov methods, which break up the data dependency between the sparse matrix-vector multiply and the dot products in standard Krylov methods. This idea has been around for a while, but in contrast to prior work (discussed in detail in Section 1.5), we can do the following:

- We have fast kernels replacing SpMV, that can compute the results of s calls to SpMV for the same communication cost as one call (Section 2.1).
- We have fast dense kernels as well, such as Tall Skinny QR (TSQR – Section 2.3) and Block Gram-Schmidt (BGS – Section 2.4), which can do the work of Modified Gram-Schmidt applied to s vectors for a factor of $\Theta(s^2)$ fewer messages in parallel, and a factor of $\Theta(s/W)$ fewer words transferred between levels of the memory hierarchy (where W is the fast memory capacity in words).
- We have new communication-avoiding Block Gram-Schmidt algorithms for orthogonalization in more general inner products (Section 2.5).
- We have new communication-avoiding versions of the following Krylov subspace methods for solving linear systems: the Generalized Minimum Residual method (GMRES – Section 3.4), both unpreconditioned and preconditioned, and the Method of Conjugate Gradients (CG), both unpreconditioned (Section 5.4) and left-preconditioned (Section 5.5).

- We have new communication-avoiding versions of the following Krylov subspace methods for solving eigenvalue problems, both standard ($Ax = \lambda x$, for a nonsingular matrix A) and "generalized" ($Ax = \lambda Mx$, for nonsingular matrices A and M): Arnoldi iteration (Section 3.3), and Lanczos iteration, both for $Ax = \lambda x$ (Section 4.2) and $Ax = \lambda Mx$ (Section 4.3).

- We propose techniques for developing communication-avoiding versions of nonsymmetric Lanczos iteration (for solving nonsymmetric eigenvalue problems $Ax = \lambda x$) and the Method of Biconjugate Gradients (BiCG) for solving linear systems. See Chapter 6 for details.

- We can combine more stable numerical formulations, that use different bases of Krylov subspaces, with our techniques for avoiding communication. For a discussion of different bases, see Chapter 7. To see an example of how the choice of basis affects the formulation of the Krylov method, see Section 3.2.2.

- We have faster numerical formulations. For example, in our communication-avoiding version of GMRES, CA-GMRES (see Section 3.4), we can pick the restart length r independently of the s-step basis length s. Experiments in Section 3.5.5 show that this ability improves numerical stability. We show in Section 3.6.3 that it also improves performance in practice, resulting in a 2.23× speedup in the CA-GMRES implementation described below.

- We combine all of these numerical and performance techniques in a shared-memory parallel implementation of our communication-avoiding version of GMRES, CA-GMRES. Compared to a similarly highly optimized version of standard GMRES, when both are running in parallel on 8 cores of an Intel Clovertown (see Appendix A), CA-GMRES achieves 4.3× speedups over standard GMRES on standard sparse test matrices (described in Appendix B.5). When both are running in parallel on 8 cores of an Intel Nehalem (see Appendix A), CA-GMRES achieves 4.1× speedups. See Section 3.6 for performance results and Section 3.5 for corresponding numerical experiments. We first reported performance results for this implementation on the Intel Clovertown platform in Demmel et al. [79].

- Incorporation of preconditioning. Note that we have not yet developed practical communication-avoiding preconditioners; this is future work. We *have* accomplished the following:

  - We show (in Sections 2.2 and 4.3) what the s-step basis should compute in the preconditioned case for many different types of Krylov methods and s-step bases. We explain why this is hard in Section 4.3.
  - We have identified two different structures that a preconditioner may have, in order to achieve the desired optimal reduction of communication by a factor of s. See Section 2.2 for details.

- We present a detailed survey of related work, including s-step KSMs (Section 1.5, especially Table 1.1) and other techniques for reducing the amount of communication in iterative methods (Section 1.6).

Chapter 2

Computational kernels

In this chapter, we present several communication-avoiding computational kernels. These will be used to construct the communication-avoiding Krylov subspace methods described in later chapters of this thesis. Just as standard Krylov subspace methods inherit their performance characteristics from the SpMV, preconditioning, and vector-vector operation kernels out of which they are built, our Krylov methods inherit their performance characteristics from the kernels described in this chapter. Not every Krylov algorithm we develop in this thesis will use all the kernels in this chapter, but all of the kernels will be used by some algorithm developed here.

We begin with Section 2.1, in which we describe the matrix powers kernel. This kernel does the same work as several SpMV operations, for about the same communication cost as a single SpMV. It achieves significant modeled and measured performance improvements over SpMV, for a large class of sparse matrices and computer hardware. Section 2.2 shows how the matrix powers kernel changes when we add preconditioning. The lack of preconditioning was a major deficiency in the "s-step" Krylov methods on which our communication-avoiding Krylov methods are based. In Section 2.2, we show the form that the matrix powers kernel must take, for left preconditioning and split preconditioning, and for general s-step Krylov bases. We also show two different structures which a preconditioner can have, in order to avoid communication in the matrix powers kernel.

In Sections 2.3 and 2.4, we discuss the two kernels which we use to orthogonalize the basis vectors in our communication-avoiding versions of Arnoldi (Section 3.3) and GMRES (Section 3.4). The first kernel (Section 2.3) is Tall Skinny QR (TSQR). TSQR computes the QR factorization of a tall skinny matrix as accurately as Householder QR, but with asymptotically less communication. In fact, it attains the lower bound of communication in parallel and between levels of the memory hierarchy, for all QR factorizations. Our sequential and parallel implementations of TSQR attain significant speedups over competing QR factorizations. The second kernel is Block Gram-Schmidt (BGS) (Section 2.4), which is a class of algorithms that are blocked versions of standard Gram-Schmidt orthogonalization. BGS itself is not new, but our novel combination of BGS and TSQR helps us get good performance in our implementation of Communication-Avoiding GMRES (see Section 3.4), without sacrificing accuracy. We also develop a new BGS with reorthogonalization algorithm that communicates less than previous Block Gram-Schmidt reorthogonalization schemes. Finally, Section 2.5 presents a version of Block Gram-Schmidt for more general

inner products.

2.1 Matrix powers kernel

In this section, we describe the matrix powers kernel. This kernel replaces the sparse matrix-vector products in standard iterative methods. A single invocation of the matrix powers kernel computes the same vectors as s invocations of sparse matrix-vector multiply (SpMV). However, it requires a factor of $\Theta(s)$ fewer messages in parallel and a factor of $\Theta(s)$ fewer reads of the sparse matrix from slow memory. In doing so, it requires no more than a constant factor more arithmetic operations than s invocations of SpMV.

We define the matrix powers kernel in Section 2.1.1. In Section 2.1.2, we present model problems to evaluate the performance of various algorithms for the kernel. Section 2.1.3 informally describes three parallel matrix powers kernel algorithms – PA0, PA1, and PA2 – and gives performance models for these on the model problems. Section 2.1.4 discusses the asymptotic behavior of these models, in the large-latency and small-latency cases. Next, Section 2.1.5 shows how PA0, PA1, and PA2 work for general sparse matrices. Section 2.1.6 shows four sequential algorithms for the matrix powers kernel – SA0, SA1, SA2, and SA3. Next, Section 2.1.7 illustrates two hybrid algorithms that combine a parallel and a sequential algorithm hierarchically. Section 2.1.8 describes low-level optimizations for our shared-memory parallel implementation of the matrix powers kernel. In Section 2.1.9, we summarize both performance models and benchmark results for those optimized implementations.

The remaining sections discuss related work. Section 2.1.10 shows some variations on the matrix powers kernel. These variations differ from the kernel as we define it, either in that they require the sparse matrix to have a specific structure, or that they compute different vector outputs. In Section 2.1.11, we give an overview of other sparse matrix kernels that avoid communication. Finally, in Section 2.1.12, we summarize related work for the matrix powers kernel itself.

Most of the content in this section was originally presented in three joint papers by Demmel et al. [77, 78, 79]. Much of the work, including all the implementations of the matrix powers kernel, was done by our coauthor Marghoob Mohiyuddin. We have changed the notation slightly to coordinate better with the algorithms elsewhere in this thesis.

2.1.1 Introduction

The *matrix powers kernel* takes an $n \times n$ sparse matrix A, a dense vector v of length n, and a three-term recurrence

$$p_{k+1}(z) = a_k z p_k(z) - b_k p_k(z) - c_k p_{k-1}(z), \text{ for } k = 1, \ldots, s, \text{ with } a_k \neq 0 \tag{2.1}$$

for a set of polynomials $p_0(z), p_1(z), \ldots, p_s(z)$ where $p_j(z)$ has degree j. The kernel computes vectors $v_1, v_2, \ldots, v_{s+1}$ given by $v_j = p_{j-1}(A)v$. The resulting vectors

$$\underline{V} = [v_1, v_2, \ldots, v_{s+1}] = [p_0(A)v, p_1(A)v, \ldots, p_s(A)v] \tag{2.2}$$

form a basis for the Krylov subspace

$$\mathcal{K}_s(A, v) = \text{span}\{v, Av, A^2v, \ldots, A^sv\}.$$

We usually assume $p_0(z) = 1$, so that $v_1 = v$. In that case, we write

$$\underline{\acute{V}} = [\acute{V}, v_{s+1}] = [v_2, v_3, \ldots, v_s, v_{s+1}].$$

(The acute accent here, as elsewhere in this thesis, suggests "shift one to the right.") The simplest polynomials used in the recurrence (2.1) are *monomials* $p_j(z) = z^j$, in which case $a_k = 1$ and $b_k = c_k = 0$ for all k. See Chapter 7 for other choices of polynomials. We require no more than three terms in the recurrence in order to keep the amount of data in fast memory small, and also for simplicity. Chapter 7 discusses the possibility of longer recurrences.

Note that the above is a slight modification to the kernel defined in Demmel et al. [77, 78], which computed only the monomial basis $\underline{\acute{V}} = [Av, A^2v, \ldots, A^sv]$. It also differs from the kernel defined in [79], namely the *Newton basis*

$$\underline{\acute{V}} = [(A - \theta_1 I)v, (A - \theta_2 I)(A - \theta_1 I)v, \ldots, (A - \theta_s I) \cdots (A - \theta_1 I)v].$$

From an implementation perspective, the three kernels differ only slightly from each other. We refer to all of them by the same name.

Algorithm 3 Straightforward matrix powers kernel implementation

Input: $n \times n$ sparse matrix A and length n vector v_1
Input: Recurrence coefficients a_k, b_k, c_k $(k = 1, 2, \ldots, s)$
Output: Vectors v_2, $\ldots$, v_{s+1} satisfying $v_{k+1} = a_k A v_k - b_k v_k - c_k v_{k-1}$ for $k = 1, 2, \ldots, s$, where $v_0 = 0$

1: $v_0 = 0$
2: **for** $k = 1$ to s **do**
3: $\quad v_{k+1} = a_k(Av_k) - b_k v_k - c_k v_{k-1}$ ▷ One SpMV and two AXPYs
4: **end for**

Algorithm 3 depicts the straightforward way to compute the vectors in Equation (2.2). It requires s sparse matrix-vector multiplications (SpMVs), one after another. SpMV is a memory-bound kernel, and the approach of Algorithm 3 fails to exploit potential reuse of the matrix A. Usually the SpMV operations dominate the performance of the kernel. In a sequential computation, s SpMV invocations require reading the entire matrix s times from slow memory. In parallel, they require $\Omega(s)$ messages (if the matrix is not block diagonal). Our algorithms can compute the same vectors sequentially with only $1 + o(1)$ read of the sparse matrix A, and in parallel with only $O(1)$ messages. This is optimal in both cases. Our algorithms can do this if the matrix A has a reasonable and common sparsity structure, to be discussed in more detail in the following sections.

2.1.2 Model problems

The algorithms of this section work for general sparse matrices, but the case of regular d-dimensional meshes with $(2b + 1)^d$-point stencils illustrates potential performance gains for

a representative class of matrices. By a *regular mesh*, we mean a simply connected domain in $\mathbb{R}^d$ (for $d = 1, 2, 3, \ldots$), with evenly spaced vertices at the grid points. Their connectivity is described by the *stencil*, which shows for each point in the mesh, how it is connected to its neighbors. For example, on a 2-D mesh, a *5-point stencil* means that each point is connected to its east, south, west, and north immediate neighbors. On a 3-D mesh, a *7-point stencil* means that each point is connected to its east, south, west, north, top, and bottom immediate neighbors. We assume here that grid points on the boundary of the domain are connected only to neighbors that are in the domain, but still following the pattern of the stencil. In the cases of the 5-point and 7-point stencils, we say that the *stencil radius* is $b = 1$, since each point is only connected to its nearest neighbors. We generalize this to $b > 1$ in the model problems that follow, so that for example, a $(2b + 1)^d$-point stencil reaches out b grid points in each of the cardinal directions (which includes "up" and "down" in 3-D).

Stencils on regular d-dimensional meshes often arise in the finite-difference discretization of partial differential equations. Here, they serve as a model problem with similar connectivity as more complex discretizations (e.g., from the method of finite elements) on possibly unstructured meshes (where the points are not regularly placed and spaced in the domain). While stencils on regular meshes are often stored in a special format that exploits their regular structure to save storage, we assume here that they are stored in a general sparse matrix format. This allows our model problems to represent the performance of our matrix powers kernel algorithms on general sparse matrices. See Section 2.1.10 for discussion of special cases of the matrix powers kernel that can exploit the storage-saving format of stencils on regular meshes.

We call the *surface* of a mesh the number of points on the partition boundary. For a 2-D square mesh with n points on each side corresponding to a 5-point stencil, partitioned into P squares of equal dimensions, the surface is $4n/\sqrt{P}$. For a 3-D cubic mesh with n points on each side corresponding to a 7-point stencil, partitioned into P cubes of equal dimensions, the surface is $6n^2/P^{2/3}$. The *volume* of a mesh is the total number of points in each processor's partition, namely n^2/P and n^3/P in the 2-D resp. 3-D cases. The *surface-to-volume ratio* of a mesh is therefore $4\sqrt{P}/n$ in the 2-D case and $6P^{1/3}/n$ in the 3-D case. (We assume all the above roots (like $P^{1/3}$) and fractions (like $n/P^{1/2}$) are integers, for simplicity.) The surface-to-volume ratio of a partition of a general sparse matrix is defined analogously. All our algorithms work best when the surface-to-volume ratio is small, as is the case for meshes with large n and sufficiently smaller P.

2.1.3 Parallel Algorithms

In this section, we describe three parallel algorithms for computing the matrix powers kernel: PA0, PA1, and PA2. PA0 is the straightforward approach using s SpMVs, whereas PA1 and PA2 avoid communication by rearranging the computation and performing redundant floating-point operations. We present the algorithms informally here, and only for the model problems described in Section 2.1.2. Section 2.1.5 will show PA0, PA1, and PA2 formally for general sparse matrices.

The straightforward parallel matrix powers kernel algorithm, which we call "PA0," consists of s applications of the usual parallel SpMV algorithm. Step k of PA0 computes $v_k = p_{k-1}(A)v$ using the three-term recurrence (Equation (2.1)). This requires (except for

$k = 1$) one sparse matrix-vector multiply and anywhere from 0 to 2 AXPY operations (depending on whether or not the coefficients b_{k+1} and c_{k+1} are zero). This works by each processor receiving messages with the necessary remotely stored entries of v_k (and possibly also v_{k-1}), and then computing its local components of v_{k+1}. PA0 does not attempt to avoid communication; we present it as a point of comparison.

In our first parallel algorithm, PA1, each processor first computes all elements of $\acute{V} = [v_2, \ldots, v_{s+1}]$ that can be computed without communication. Simultaneously, it begins sending all the components of v_1 needed by the neighboring processors to compute their remaining components of $[v_2, \ldots, v_{s+1}]$. When all local computations have finished, each processor blocks until the remote components of v_1 arrive, and then finishes computing its portion of $[v_2, \ldots, v_{s+1}]$. This algorithm maximizes the potential overlap of computation and communication, but performs more redundant work than necessary because some entries of $\acute{V}$ near processor boundaries are computed by both processors. As a result, we developed PA2, which uses a similar communication pattern but minimizes redundant work.

In our second parallel approach, PA2, each processor computes the "least redundant" set of local values of $\acute{V}$ needed by the neighboring processors. This saves the neighbors some redundant computation. Then the processor sends these values to its neighbors, and simultaneously computes the remaining locally computable values. When all the locally computable values are complete, each processor blocks until the remote entries of $\acute{V}$ arrive, and completes the work. This minimizes redundant work, but permits slightly less overlap of computation and communication.

We estimate the cost of our parallel algorithms by measuring five quantities:

1. Number of arithmetic operations per processor
2. Number of floating-point numbers communicated per processor (a bandwidth term)
3. Number of messages sent per processor (a latency term)
4. Total memory required per processor for the matrix
5. Total memory required per processor for the vectors

Informally, it is clear that both PA1 and P2 minimize communication to within a constant factor. We ignore cancellation in the powers of the sparse matrix or in the three-term recurrence (Equation (2.1)), so that the complexity only depends on the sparsity pattern of A. For simplicity of notation we assume all the nonzero entries of A and v_1 are positive. Then, the set $\mathcal{D}$ of processors owning entries of v on which block row i of $\acute{V}$ depends is just the set of processors owning those v_{j+1} for which block row i of $A+A^2+\cdots+A^k$ has a nonzero j-th column. In both algorithms PA1 and PA2, the processor owning row block i receives exactly one message from each processor in $\mathcal{D}$, which minimizes latency. Furthermore, PA1 only sends those entries of v in each message on which the answer depends, which minimizes the bandwidth cost. PA2 sends the same amount of data, although different values, so as to minimize redundant computation.

Example: 1-D mesh

We now illustrate the difference between PA0, PA1, and PA2, using the example of a 1-D mesh with stencil radius $b = 1$, that is, a tridiagonal sparse matrix. In PA0, the computational cost is $2s$ messages, $2s$ words sent, and $5sn/P$ flops (3 multiplies and 2 additions per vector component computed). The memory required per processor is $3n/P$ matrix entries and $(s + 1)n/P + 2$ vector entries (for the local components of $\underline{V}$ and for the values on neighboring processors).

Figure 2.1a shows the operation of PA1 with $s = 8$. Each row of circles represents the entries of $A^j v$, for $j = 0$ to $j = 8$. (It suffices to consider only the monomial basis.) A subset of 30 components of each vector is shown, owned by 2 processors, one to the left of the vertical green line, and one to the right. (There are further components and processors not shown.) The diagonal and vertical lines show the dependencies: the three lines below each circle (component i of $A^j v$) connect to the circles on which its value depends (components $i - 1$, i and $i + 1$ of $A^{j-1} v$). In the figure, the local dependencies of the left processor are all the circles that can be computed without communicating with the right processor. The remaining circles without attached lines to the left of the vertical green line require information from the right processor before they can to be computed.

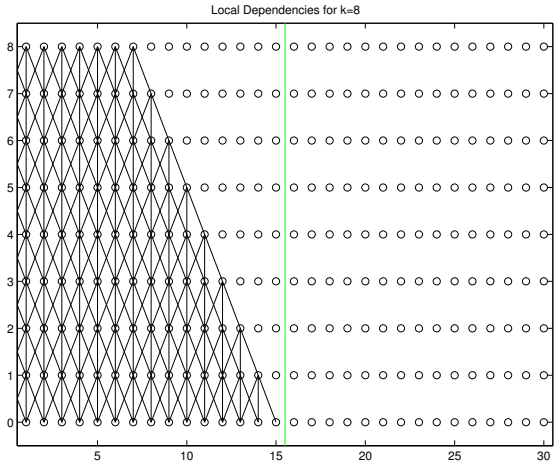
Figure 2.1b shows how to compute these remaining circles using PA1. The dependencies are again shown by diagonal and vertical lines below each circle, but now dependencies on data formally owned by the right processor are shown in red. All these values in turn depend on the $s = 8$ leftmost value of v owned by the right processor, shown as black circles containing red asterisks in the bottom row. By sending these values from the right processor to the left processor, the left processor can compute all the circles whose dependencies are shown in Figure 2.1b. The black circles indicate computations ideally done only by the left processor, and the red circles show redundant computations, i.e., those also performed by the right processor.

We assume that $s < n/P$, so that only data from neighboring processors is needed, rather than more distant processors. Indeed, we expect that $s \ll n/P$ in practice, which will mean that the number of extra flops (not to mention extra memory) will be negligible. We continue to make this assumption later without repeating it, and use it to simplify some expressions in Table 2.1.

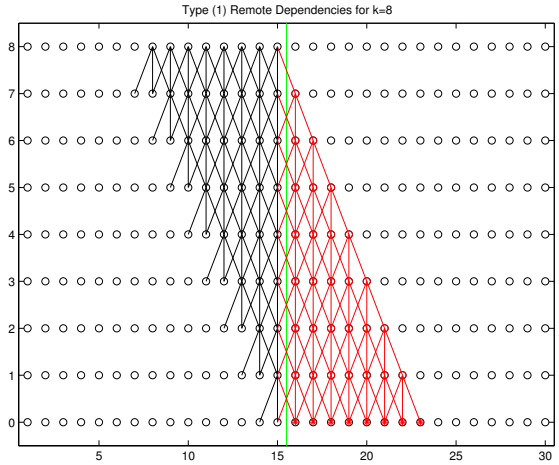
Figure 2.1c illustrates PA2. We note that the blue circles owned by the right processor and attached to blue lines can be computed locally by the right processor. The 8 circles containing red asterisks can then be sent to the left processor to compute the remaining circles connected to black and/or red lines. This saves the redundant work represented by the blue circles, but leaves the redundant work to compute the red circles, about half the redundant work of PA1. PA2 takes roughly $2.5s^2$ more flops than PA0, which is half as many extra flops as PA1.

Parallel complexity of PA0, PA1, and PA2 on regular meshes

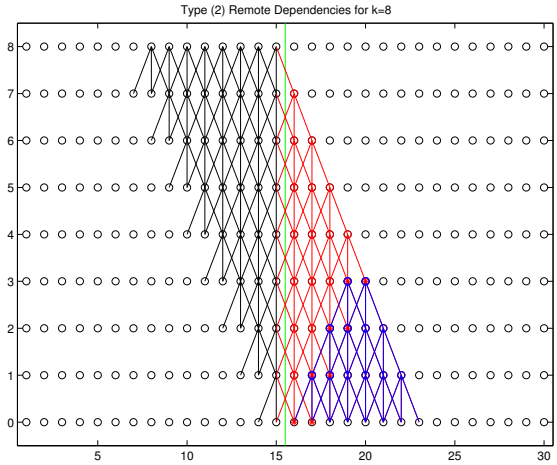
PA1 and PA2 can be extended to higher dimensions and different mesh bandwidths (and sparse matrices in general). There, the pictures of which regions are communicated and which are computed redundantly become more complicated, higher-dimensional polyhedra,



(a) Locally computable components.



(b) Remote dependencies in PA1.



(c) Remote dependencies in PA2.

Figure 2.1: Dependencies in the PA1 and PA2 algorithms for computing the matrix powers kernel $\underline{\acute{V}} = [v_2, \ldots, v_{s+1}]$ for a tridiagonal sparse matrix A.

but the essential algorithms remain the same. Table 2.1 summarizes all of the resulting costs for 1-D, 2-D, and 3-D meshes. In that table, "Mess" is the number of messages sent per processor, "Words" is the total size of these messages, "Flops" is the number of floating point operations, "MMem" is the amount of memory needed per processor for the matrix entries, and "VMem" is the amount of memory needed per processor for the vector entries. Lower order terms are sometimes omitted for clarity. For more detailed parallel performance models, and evaluations of those models on existing and projected future machines, see Demmel et al. [77, 78].

2.1.4 Asymptotic performance

An asymptotic performance model for the parallel algorithms suggests that when the latency α is large, the speedup is close to s, as we expect (see e.g., Demmel et al. [77]). When the latency α is not so large, the best we could hope for is that s can be chosen so that the new running time is fast independent of α. The model shows that this is the case under two reasonable conditions:

1. The time it takes to send the entire local contents of a processor is dominated by bandwidth, not latency, and
2. The time to do $O(N^{1/d})$ flops on each of the N elements stored on a processor exceeds α.

For the sequential algorithm SA2, if latency is small enough, then the asymptotic performance model suggests the following:

1. The best speedup is bounded by $2 + (words/row)$, where $words/row$ is the number of 8-byte words per row of the matrix A–this includes the index entries too.
2. The optimal speedup is strongly dependent on the β/t_f ratio. For the specific case of stencils, the optimal speedup is expected to be close to the upper bound in the previous item when β/t_f is large.

2.1.5 Parallel algorithms for general sparse matrices

In this section, we present versions for general sparse matrices of the previously mentioned parallel matrix powers kernel algorithms, PA0 (the straightforward approach), PA1 (which minimizes the number of messages), and PA2 (which reduces the amount of redundant computation, but has less potential overlap of communication and computation than PA1). We begin by defining graph-theoretic notation for describing the general algorithms. This notation will also apply, unless otherwise specified, to the general sequential algorithms that we will describe in Section 2.1.6.

Graph-theoretic notation

Suppose that A is an $n \times n$ sparse matrix. It is natural to associate a directed graph with A, with one vertex for every row / column, and an edge from vertex i to vertex j if $A_{ij} \neq 0$.

Problem	Costs	PA0	PA1	PA2
1-D mesh $b \geq 1$	Mess	$2s$	2	2
	Words	$2bs$	$2bs$	$2bs$
	Flops	$(4b+1)s\frac{n}{P}$	$(4b+1)(s\frac{n}{P}+bs^2)$	$(4b+1)(s\frac{n}{P}+\frac{bs^2}{2})$
	MMem	$(2b+1)\frac{n}{P}$	$(2b+1)\frac{n}{P}+bs(4b+2)$	$(2b+1)\frac{n}{P}+bs(2b+1)$
	VMem	$(s+1)\frac{n}{P}+2b$	$(s+1)\frac{n}{P}+2bs$	$(s+1)\frac{n}{P}+2bs$
2-D mesh $(2b+1)^2$ pt stencil	Mess	$8s$	8	8
	Words	$4bs(\frac{n}{P^{1/2}}+b)$	$4bs(\frac{n}{P^{1/2}}+bs)$	$4bs(\frac{n}{P^{1/2}}+1.5bs)$
	Flops	$(8b^2+8b+1)s\frac{n^2}{P}$	$(8b^2+8b+1)\cdot$ $(s\frac{n^2}{P}+2bs^2\frac{n}{P^{1/2}}+\frac{4}{3}b^2s^3)$	$(8b^2+8b+1)\cdot$ $(s\frac{n^2}{P}+bs^2\frac{n}{P^{1/2}}+b^2s^3)$
	MMem	$(2b+1)^2\frac{n^2}{P}$	$(2b+1)^2(\frac{n^2}{P}+4bs\frac{n}{P^{1/2}}+4b^2s^2)$	$(2b+1)^2(\frac{n^2}{P}+2bs\frac{n}{P^{1/2}}+b^2s^2)$
	VMem	$(s+1)\frac{n^2}{P}+4b\frac{n}{P^{1/2}}$ $+4b^2$	$(s+1)\frac{n^2}{P}+4bs\frac{n}{P^{1/2}}$ $+4b^2s^2$	$(s+1)\frac{n^2}{P}+4bs\frac{n}{P^{1/2}}$ $+6b^2s^2$
3-D mesh $(2b+1)^3$ pt stencil	Mess	$26s$	26	26
	Words	$6bs\frac{n^2}{P^{2/3}}+12b^2s\frac{n}{P^{1/3}}$ $+O(b^3s)$	$6bs\frac{n^2}{P^{2/3}}+12b^2s^2\frac{n}{P^{1/3}}$ $+O(b^3s^3)$	$6bs\frac{n^2}{P^{2/3}}+12b^2s^2\frac{n}{P^{1/3}}$ $+O(b^3s^3)$
	Flops	$(2(2b+1)^3-1)s\frac{n^3}{P}$	$(2(2b+1)^3-1)\cdot$ $(s\frac{n^3}{P}+3bs^2\frac{n^2}{P^{2/3}}+O(b^2s^3\frac{n}{P^{1/3}}))$	$(2(2b+1)^3-1)\cdot$ $(s\frac{n^3}{P}+\frac{3}{2}bs^2\frac{n^2}{P^{2/3}}+O(b^2s^3\frac{n}{P^{1/3}}))$
	MMem	$(2b+1)^3\frac{n^3}{P}$	$(2b+1)^3\cdot$ $(\frac{n^3}{P}+6bs\frac{n^2}{P^{2/3}}+O(b^2s^2\frac{n}{P^{1/3}}))$	$(2b+1)^3\cdot$ $(\frac{n^3}{P}+3bs\frac{n^2}{P^{2/3}}+O(b^2s^2\frac{n}{P^{1/3}}))$
	VMem	$(s+1)\frac{n^3}{P}+6b\frac{n^2}{P^{2/3}}$ $+O(b^2\frac{n}{P^{1/3}})$	$(s+1)\frac{n^3}{P}+6bs\frac{n^2}{P^{2/3}}$ $+O(b^2s^2\frac{n}{P^{1/3}})$	$(s+1)\frac{n^3}{P}+6bs\frac{n^2}{P^{2/3}}$ $+O(b^2s^2\frac{n}{P^{1/3}})$

Table 2.1: Summary of performance models for parallel matrix powers kernel algorithms PA0, PA1, and PA2 on regular 1-D, 2-D, and 3-D meshes. Some lower-order terms are omitted.

When computing $y := A \cdot x$ (the SpMV operation), this means that the value of y_i depends on the value of x_j. In this section, we will omit "the value of," and say, for example, that y_i depends on x_j.

Communication-avoiding implementations of the matrix powers kernel require identifying dependencies across multiple sparse matrix-vector multiplications, for example the set of all j such that for $z = A^k x$, z_i depends on x_j. To do so, we build a graph $G = G(A)$ expressing such dependencies. Let $x_j^{(i)}$ be the j-th component of $x^{(i)} = A^i \cdot x^{(0)}$. We associate a vertex of G with each $x_j^{(i)}$ for $i = 0, \ldots, s$ and $j = 1, \ldots, n$, using the same notation $x_j^{(i)}$ to name the vertex of G as the element of $x^{(i)}$. We associate an edge from $x_j^{(i+1)}$ to $x_m^{(i)}$ when $A_{jm} \neq 0$, and call this graph of $n(s+1)$ vertices G. (We will not need to construct all of G in practice, but using G makes it easy to describe our algorithms.) We say that i is the *level* of vertex $x_j^{(i)}$.

Each vertex will also have an *affinity* q. For the parallel algorithms, q corresponds to the processor number ($q = 1, 2, \ldots, P$) to which $x_j^{(i)}$ has locality. For example, in the distributed-memory case, if $x_j^{(i)}$ has affinity q, then $x_j^{(i)}$ is stored on processor q. In the shared-memory case, "locality" might refer instead to memory affinity, where processor q can access $x_j^{(i)}$ with lower latency and / or higher bandwidth than other processors can. For the sequential algorithms, q will correspond to the number of the block of contiguously-stored entries which can be loaded from slow memory into fast memory with a single read operation. See Section 2.1.6 for details. In this section, our notation will refer to the parallel case. We assume all vertices $x_j^{(0)}$, $x_j^{(1)}$, $\ldots$, $x_j^{(s)}$ have the same affinity, depending only on j. We let G_q denote the subset of vertices of G with affinity q, $G^{(i)}$ to mean the subset of vertices of G with level i, and $G_q^{(i)}$ to mean the subset with affinity q and level i.

Let S be any subset of vertices of G. We let $R(S)$ denote the set of vertices reachable by directed paths starting at vertices in S (so $S \subset R(S)$). We need $R(S)$ to identify dependencies of sets of vertices on other vertices. We let $R(S, m)$ denote vertices reachable by paths of length at most m starting at vertices in S. We write $R_q(S)$, $R^{(i)}(S)$ and $R_q^{(i)}(S)$ as before to mean the subsets of $R(S)$ with affinity q, level i, and both affinity q and level i, respectively.

Next we need to identify each processor's set of *locally computable components*, that processor q can compute given only the values in $G_q^{(0)}$. We denote the set of locally computable components by L_q, where

$$L_q \equiv \{x \in G_q : R(x) \subset G_q\}. \tag{2.3}$$

As before $L_q^{(i)}$ will denote the vertices in L_q at level i.

Finally, for the PA2 algorithm which we will present below (Algorithm 6), we need to identify the minimal subset $B_{q,r}$ of vertex values of G that processor r needs to send processor q so that processor q can finish computing all its vertices G_q. We say that $x \in B_{q,r}$ if and only if $x \in L_r$, and there is a path from some $y \in G_q$ to x such that x is the first vertex of the path in L_r.

Parallel algorithms

Given all this notation, we can finally state versions of PA0 (Algorithm 4), PA1 (Algorithm 5), and PA2 (Algorithm 6) for general sparse matrices and partitions among processors. We state the algorithms assuming a distributed-memory programming model. We assume when describing the algorithms that the "send" and "receive" operations can execute asynchronously, allowing overlap of each communication event with both computation and other communication events. In a shared-memory model, the "send," "receive," and "wait for receive" operations would be handled automatically, without programmer intervention, by the shared-memory hardware. Otherwise, the shared-memory and distributed-memory algorithms are the same.

Algorithm 4 PA0 algorithm (code for processor q)

for $i = 1$ to s **do**
  for all processors $r \neq q$ **do**
    Send all $x_j^{(i-1)}$ in $R_q^{(i-1)}(G_r^{(i)})$ to processor r
  end for
  for all processors $r \neq q$ **do**
    Receive all $x_j^{(i-1)}$ in $R_r^{(i-1)}(G_q^{(i)})$ from processor r
  end for
  Compute all $x_j^{(i)}$ in $L_q^{(i)}$
  Wait for above receives to finish
  Compute remaining $x_j^{(i)}$ in $G_q^{(i)} \setminus L_q^{(i)}$
end for

Algorithm 5 PA1 algorithm (code for processor q)

for all processors $r \neq q$ **do**
  Send all $x_j^{(0)}$ in $R_q^{(0)}(G_r)$ to processor r
end for
for all processors $r \neq q$ **do**
  Receive all $x_j^{(0)}$ in $R_r^{(0)}(G_q)$ from proc. r
end for
for $i = 1$ to s **do**
  Compute all $x_j^{(i)}$ in L_q ▷ Black vertices connected by lines in Figure 2.1a
end for
Wait for above receives to finish
for $i = 1$ to s **do**
  Compute remaining $x_j^{(i)}$ in $R(G_q) \setminus L_q$ ▷ Black and red vertices connected by lines in Figure 2.1b
end for

We illustrate the algorithm PA1 on the matrix A whose graph is in Figure 2.2a. This sparse matrix corresponds to a linear finite-element discretization on a 2-D unstructured mesh. The vertices of the graph in Figure 2.2a represent rows / columns of A, and the edges

Algorithm 6 PA2 algorithm (Code for processor q)

for $i = 1$ to s **do** ▷ Phase I
  Compute $x_j^{(i)}$ in $\cup_{r \neq q}(R(G_r) \cap L_q)$ ▷ Blue circled vertices in Figure 2.1c
end for
for all processors $r \neq q$ **do**
  Send $x_j^{(i)}$ in $B_{r,q}$ to processor r ▷ Blue circled vertices with red interiors in Figure 2.1c
end for
for all processors $r \neq q$ **do**
  Receive $x_j^{(i)}$ in $B_{r,q}$ from processor r ▷ Blue circled vertices with red interiors in Figure 2.1c
end for
for $i = 1$ to s **do** ▷ Phase II
  Compute $x_j^{(i)}$ in $L_q \setminus \cup_{r \neq q}(R(G_r) \cap L_q)$ ▷ Locally computable vertices of right processor, minus blue circled vertices in Figure 2.1c
end for
Wait for above receives to finish
for $i = 1$ to s **do** ▷ Phase III
  Compute remaining $x_j^{(i)}$ in $R(G_q) \setminus L_q \setminus \cup_{r \neq q}(R(G_q) \cap L_r)$ ▷ Black circled vertices in Figure 2.1c that are connected by lines
end for

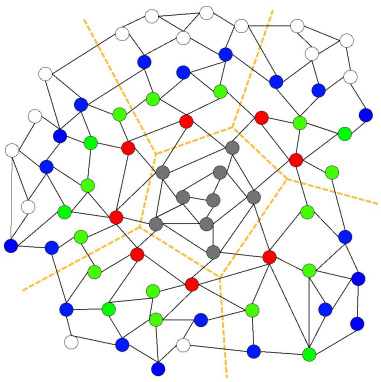
represent nonzeros. For simplicity, we use a symmetric matrix (corresponding to undirected edges), even though PA0, PA1, and PA2 all work for nonsymmetric sparse matrices. The dotted orange lines in Figure 2.2a separate vertices owned by different processors. We let q denote the processor owning the 9 gray vertices in the center of the figure. In other words, the gray vertices comprise the set $G_q^{(0)}$. For all the neighboring processors $r \neq q$, the red vertices form the set $R_r^{(0)}(G_q)$ for $s = 1$, the red and green vertices together comprise $R_r^{(0)}(G_q)$ for $s = 2$, and the red, green and blue vertices together form $R_r^{(0)}(G_q)$ for $s = 3$.

Figures 2.2b and 2.2c illustrate algorithm PA2 on the same sparse matrix as we used for PA1 in the previous paragraph, for the case $s = 3$. The "phases" of the PA2 algorithm are explained in Algorithm 6. In Figure 2.2b, the red vertices form the set $B_{q,r}^{(0)}$ (i.e., the members of $B_{q,r}$ at level 0), and in Figure 2.2c the green vertices comprise $B_{q,r}^{(1)}$ (the members of $B_{q,r}$ at level 1).

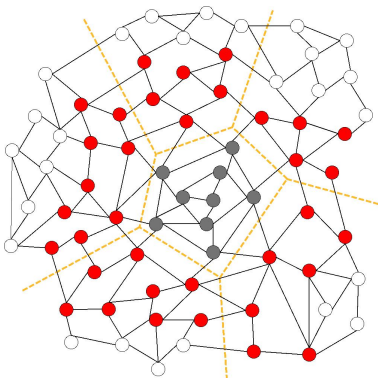
2.1.6 Sequential algorithms for general sparse matrices

In this section, we describe four sequential algorithms for the matrix powers kernel:

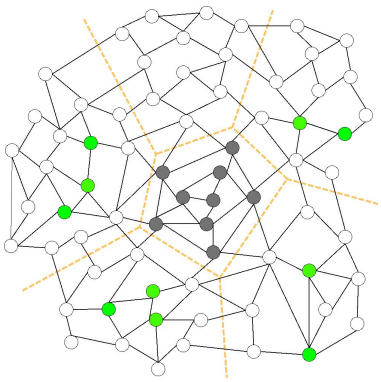
- SA0 (the straightforward approach)
- SA1 (avoids memory traffic for the sparse matrix, but not for the vectors; requires no redundant computation)
- SA2 (avoids memory traffic for both the sparse matrix and the vectors; "explicit" algorithm, in contrast to SA3)



(a) PA1 example: The red entries of $x^{(0)}$ are the ones needed when $s = 1$, the green entries are the additional ones needed when $s = 2$, and the blue entries are the additional ones needed when $s = 3$.



(b) PA2 example ($s = 3$): Entries of $x^{(0)}$ which need to be fetched are colored red.



(c) PA2 example ($s = 3$): Entries of $x^{(1)}$ which need to be fetched are colored green.

Figure 2.2: Illustration of PA1 and PA2 data dependencies on part of a graph of a sparse matrix A, corresponding to a linear finite element discretization on a 2-D unstructured mesh. The dotted lines define the different blocks, each of which has affinity to a different processor. The example is shown from the perspective of the processor q holding the central block.

- SA3 ("implicit" algorithm, for when fast memory is a cache; avoids memory traffic for both the sparse matrix and the vectors)

These algorithms reuse notation that we developed in the last section for the parallel algorithms PA0, PA1, and PA2. In fact, SA1 and SA2 both "emulate" PA1 (Algorithm 5 in Section 2.1.5). By "emulate," we mean that the matrix (and vectors, except for SA1) are partitioned into P blocks, as in the parallel algorithms, but only one processor executes those blocks in a particular order. When computing a block q, different blocks may need to be loaded redundantly, in order to fetch elements needed for computing q. The order in which the blocks are loaded can be optimized to reduce the number of messages (a latency term). The problem of choosing the optimal ordering of blocks can be posed as a Traveling Salesperson Problem (TSP), as we discuss in Demmel et al. discuss [77].

Here are the four sequential algorithms for the matrix powers kernel. In *SA0*, also called the "straightforward sequential approach," we assume that the matrix does not fit in fast memory but the vectors do. This algorithm will keep all the components of the s-step basis vectors in fast memory, and read all the entries of A from slow to fast memory to compute each basis vector, thereby reading the sparse matrix A s times in all.

In *SA1*, we assume that the matrix does not fit in fast memory but the vectors do. SA1 emulates PA1 by partitioning the sparse matrix A into P block rows, and looping from $i = 1$ to $i = P$, reading from slow memory those parts of the matrix needed to perform the same computations performed by processor i in PA1, and updating the appropriate components of the basis vectors in fast memory. Since all components of the vectors are in fast memory, no redundant computation is necessary. We choose P as small as possible, to minimize the

number of slow memory accesses (a latency term). We do not show performance results for SA1, since SA2 and SA3 require less memory traffic and therefore should perform better than SA1, unless the sparse matrix has a large number of nonzeros per row.

In *SA2*, we assume that neither the sparse matrix nor the vectors fit in memory. SA2 will still emulate PA1 by looping from $i = 1$ to $i = P$, but read from slow memory not just parts of the sparse matrix A but also those parts of the vectors needed to perform the same computations performed by processor i in PA1, and finally write back to slow memory the corresponding components of the basis vectors. Depending on the structure of A, redundant computation may or may not be necessary. We again choose P as small as possible. This algorithm is also called "explicit cache blocking," in contrast to the algorithm that follows.

SA3 is also called "implicit cache blocking." This algorithm improves upon SA2 by only computing entries of the basis vectors which have not already been computed. Thus, there is no redundant computation in SA3, unlike in SA2, which may possibly involve redundant computation. Note that in contrast to SA2, which makes explicit copies of the boundary entries on other blocks, the implicit algorithm maintains no such copies. It relies instead on the cache to deliver whatever remote data is necessary automatically. That means SA3 is only feasible to implement when fast memory is a cache, that is, when the programmer does not have to direct data transfers between slow and fast memory explicitly. Otherwise, SA2 is preferred.

SA3 performs no extra arithmetic operations. This has the potential advantage when the basis length s is sufficiently large, to result in significantly fewer flops than SA2. It may provide provide speedups over the straightforward algorithm SA0, even when SA2 does not. However, this improvement comes at the cost of bookkeeping to keep track of what entries need to be computed when looking at a given level of block q, i.e., the computation schedule. The computation schedule also includes the order in which the blocks are traversed, thus making SA3 more sensitive to the block ordering when compared to SA2.

An interesting problem that occurs in both SA2 and SA3 relates to the ordering of elements of the starting vector $x^{(0)}$. When elements of $x^{(0)}$ are needed from slow memory, they might not be contiguously placed. Thus, entries of the vector $x^{(0)}$ and the matrix A may be reordered to minimize latency. This can be posed as another Traveling Salesperson Problem, as we describe in Demmel et al. [77, Section 3.5]. This is a different TSP than the block ordering problem mentioned above: here the inputs to the TSP are the *entries* of the blocks (and thus the individual rows and columns of the sparse matrix A), rather than the blocks themselves. We leave the incorporation of solutions of these TSPs as future work.

2.1.7 Hybrid algorithms

The parallel and sequential matrix powers kernel algorithms above may be nested. This results in a hybrid algorithm with a hierarchical organization. (TSQR can also be nested in this way, as we will see in Section 2.3.) For example, our shared-memory parallel matrix powers kernel implementation, for which we show performance results in Section 3.6, uses the parallel algorithm PA1 on the top level (i.e., for multiple cores), and either SA2 or SA3 at the lower level (i.e., to reduce the volume of data movement between the core and DRAM). Thus, the matrix is "thread-blocked" first for the parallel algorithm, and then "cache-blocked" within each thread. We call the combination of PA1 and SA2 the "explicitly cache-blocked

parallel algorithm," or "PA1 / SA2" for short, and we call the combination of PA1 and SA3 the "implicitly cache-blocked parallel algorithm," or "PA1 / SA3."

For our hybrid algorithms, we define additional notation. We now define the *affinity* of a vertex as a pair (q, b), instead of just a single index. In the pair (q, b), q denotes the thread number, and b the cache block number on thread q. Given this definition, V_q means the set of vertices on thread q, and $V_{q,b}$ means the set of vertices in cache block b on thread q. For thread q, we let b_q denote the number of cache blocks having affinity to that thread.

We show PA1 / SA2 and PA1 / SA3 as Algorithms 7 resp. 8. One way to look at PA1 / SA2 is that it emulates PA1 with b blocks on p processors ($b \geq p$). For both algorithms, we distribute the cache blocks to different threads in a balanced manner. For PA1 / SA3, we note that it performs redundant arithmetic when compared to the implicit sequential algorithm SA3. These extra arithmetic operations are due to parallelization. Therefore, the computation schedule for the sequential part of PA1 / SA3 must also account for computing the redundant "ghost" entries.

Algorithm 7 PA1 / SA2: (code for processor q)

1: **for** $m = 1$ to b_q **do** ▷ b_q: number of cache blocks for thread q
2: Fetch entries in $R(V_{q,m}, k)^{(0)} - V_{q,m}^{(0)}$ to ghost zone
3: **for** $i = 1$ to s **do**
4: Compute all $x_j^{(i)} \in R(V_{q,m}, k)^{(i)}$
5: **end for**
6: **end for**

Algorithm 8 PA1 / SA3 (code for processor q)

1: $C = \varphi$ ▷ C = set of computed entries
2: Fetch entries in $R(V_q^{(0)}, k) - V_q^{(0)}$ to ghost zone
3: **for** $m = 1$ to b_q **do**
4: **for** $i = 1$ to k **do**
5: Compute all $x_j^{(i)} \in R(V_{q,m}, k)^{(i)} \setminus C$
6: $C \leftarrow C \cup R(V_{q,m}, k)^{(i)}$
7: **end for**
8: **end for**

One reason we describe both the explicit and implicit algorithms is because there is no clear winner among the two. The choice depends on the nonzero pattern of the matrix. The explicit algorithm has a memory access pattern which does not go beyond the current block after the data has been fetched in to the ghost zones and also admits the use of the cache bypass optimization. Because of the explicit copy of the entries on neighboring cache blocks, contiguous blocks of data are computed at each level. However, the number of redundant flops for the explicit algorithm can increase dramatically for matrices whose powers grow quickly in density, resulting in no performance gains. The implicit algorithm, on the other hand, only has redundant flops due to parallelization. Since the number of threads is typically small, the number of redundant flops is expected to grow slowly for the implicit algorithm. However, the memory accesses for the implicit algorithm can span

multiple cache blocks, making it more sensitive to the ordering in which the cache blocks are computed and stored. Furthermore, the implicit algorithm has a higher overhead, resulting in performance degradation if the kernel is memory bound. In general, if the matrix density grows slowly with successive powers of the sparse matrix A, then the explicit algorithm is better. Otherwise, the implicit algorithm is expected to win.

2.1.8 Optimizations

Given that the matrix A is sparse, a straightforward implementation of the matrix powers kernel may perform poorly. This is especially true since SpMV is memory bandwidth limited (see Williams et al. [241]). Although our sequential algorithms maximize cache reuse, the matrix powers kernel can still be memory bound for small basis lengths s. Thus, performance tuning is crucial for getting good performance for the matrix powers kernel.

Some of the optimizations we use are borrowed from a highly optimized SpMV implementation (Williams et al. [241]). This also serves as the baseline for our performance comparisons. We implement the optimizations of Williams et al. for sparse matrix computations and storage. These include, among others,

- eliminating branches from inner loops, to keep the instruction pipeline as full as possible,
- invocation of special hardware instructions for parallel arithmetic and data movement of 2 or 4 adjacent floating-point values,[1] by means of compiler-specific directives ("intrinsics")
- register tiling, and
- using shorter integer data types (than the standard 32-bit integers in C and Fortran) for indices in the cache blocks. This still produces correct code, if the dimensions and numbers of entries in the cache blocks are small enough.

Since writing the computational kernels with all possible combinations of these optimizations is too tedious and time consuming, we use code generators in the same manner as Williams et al. do. Given the difficulty of deciding the right optimizations and parameters by human analysis, our implementation has an *autotuning* phase, where it attempts to minimize runtime by searching over many possible combinations of data structures and tuning parameters. These data structures and parameters are described below.

Partitioning strategy

We use a recursive algorithm which first creates thread partitions and then recursively creates cache blocks for each thread partition. The recursion stops when the current partition is small enough to fit in the thread cache. For creating the partitions at each recursion level, we use one of two strategies:

[1] These elementwise parallel operations with a small number of adjacent values are commonly called "SIMD" (single instruction multiple data) instructions.

- invoking the graph partitioning library Metis (version 4.0 – see Karypis and Kumar [149]), or
- subpartitioning the current partition into contiguous blocks of rows, with equal work in each subpartition.

There are many different graph partitioning algorithms and implementations besides Metis. We chose this library because we are familiar with it, not necessarily because it is optimal for performance. We leave trying other partitioning strategies and their implementations to future work.

When Metis is used, the matrix and the vectors need to be permuted to make each thread block and cache block contiguous. Using Metis can result in less communication between blocks, and fewer (redundant) arithmetic operations per block. However, these benefits due to permuting the matrix can be offset by a more irregular memory access pattern. This is particularly true for the implicit algorithm, where memory accesses can be spread over multiple cache blocks. Thus, the decision of whether or not to use Metis is made during the autotuning phase, by timing the matrix powers kernel using both partitioning strategies. The version of Metis we used requires that nonsymmetric matrices be symmetrized before computing the reordering. We did so by giving the matrix $A + A^*$ to Metis. This does not affect the matrix powers kernel; it only affects the computation of the reordering itself.

Inner sequential algorithm

Since the choice of whether to use the explicit (SA2) or the implicit (SA3) implementation underneath PA1 (in the hybrid algorithm) depends on the matrix nonzero pattern, we make the decision by timing during the autotuning phase.

Register tile size

Register tiling a sparse matrix can avoid memory traffic by reducing the number of indices to read from slow memory (Williams et al. [241]). It can also improve instruction throughput (Vuduc [235]). Williams et al. point out that since SpMV is typically memory bound on modern multicore platforms, heuristics which try to minimize memory footprint of the matrix are sufficient when tuning SpMV alone. Thus, for ordinary SpMV, for sufficiently large sparse matrices, an elaborate and time-intensive automatic tuning step is often unnecessary.

In contrast, the matrix powers kernel aims to have the arithmetic intensity increase with the basis length s, so that the kernel eventually becomes compute bound. This is often the case. Furthermore, the use of register tiles may result in redundant arithmetic operations (due to explicitly stored zeros in the register tiles, that were not there in the original sparse matrix). If the matrix powers kernel is already compute bound, these extra arithmetic operations may actually reduce performance. Therefore, we autotune to decide whether a larger register tile should be used or not. Note that for PA1 / SA3 (Algorithm 8), the register blocks must be square, because we need to look at the quotient graph to figure out whether a register tile has already been computed.

Cache bypass optimization for PA1 / SA2

For the case of the PA1 / SA2 algorithm, we note that although we compute extra entries in the ghost zones, they do not need to be written back to slow memory. Furthermore, we do not need to keep all the $s + 1$ basis vectors in a given cache block in cache. We only need to keep the vectors that participate in the three-term recurrence (one term for the monomial basis, two terms for the Newton basis, and three terms for the Chebyshev and modified Newton bases – see Section 7 for details): in the worst case, three vectors. For our implementation, we restricted ourselves to the Newton basis, which only requires two vectors: one at the current level being computed, and one and the level below needed to be in cache. Thus, we compute by cycling over two buffers, which are always in cache. One buffer is used to compute the current level, and the other holds the previous level. Once the current level has been computed, we copy the buffer to the vector in slow memory by bypassing the cache, using special compiler directives to do so. This optimization is particularly useful in reducing memory traffic on write-allocate architectures, like the Intel Clovertown architecture for which we show performance results in Section 3.6.

Algorithm 9 PA1 / SA2 with cache bypass (code for processor q)

1: Preallocate two vector buffers z_0 and z_1 with size $\max_{m=1}^{b_q} |R(V_{q,m}, k)^{(1)}|$ each, to store any $R(V_{q,m}, k)^{(i)}$ $(1 \leq i \leq k)$ ▷ This can be generalized to three buffers for the modified Newton or Chebyshev bases (see Chapter 7).
2: **for** $m = 1$ to b_q **do**
3: Fetch entries in $R(V_{q,m}, k)^{(0)} \setminus V_{q,m}^{(0)}$ to ghost zone
4: Compute all rows in $R(V_{q,m}, k)^{(1)}$ using $R(V_{q,m}, k)^{(0)}$ and store in z_0
5: Copy from z_0 (in cache) to $x_{q,m}^{(1)}$ (in slow memory), using cache bypass
6: **for** $i = 2$ to s **do**
7: Compute all rows in $R(V_{q,m}, k)^{(1)}$ using $z_{i \bmod 2}$ and store in $z_{(i+1) \bmod 2}$
8: Copy $z_{(i+1) \bmod 2}$ (in cache) to $x_{q,m}^{(i)}$ (in slow memory), using cache bypass
9: **end for**
10: **end for**

Without cache bypass, due to write-allocate behavior, each output vector will contribute twice the bandwidth: Once for allocating cachelines when being written, and once again when evicted from cache while computing the next cache block. Furthermore, since all the rows in $V_{q,m}^{(i)}$ for level i on cache block m of thread q are stored contiguously, this copy to slow memory contiguous and therefore inexpensive. We show the version of PA1 / SA2 with the cache bypass optimization as Algorithm 9. Note that this optimization cannot be applied to PA1 / SA3, because the memory access pattern in the latter algorithm can span multiple blocks.

Stanza encoding

This is a memory optimization to minimize the bookkeeping overhead for PA1 / SA3 (Algorithm 8). Note that we need to iterate through the computation sequence for each level on each block. If most of the computed entries are contiguously placed, we can store the

computation sequence as a sequence of contiguous blocks or *stanzas*. Each contiguous block is encoded by its starting and ending entries. Since we stream through the computation sequence, this optimization can improve performance by reducing the volume of slow memory traffic. We also try to use integers of shorter length than the usual 32 bits for storing the stanzas when possible, to reduce the overhead further.

Software prefetching

Software prefetching can be particularly useful for a memory-bound kernel like SpMV. However, the prefetch distance and strategy can depend both on the algorithm (implicit and explicit algorithms have different data structures, and therefore different software prefetch strategies) and the sparse matrix. Thus, the right software prefetch distances are chosen during the autotuning phase. Since software prefetching is a streaming optimization, it was found to be useful only for $s = 1$, where there is no reuse of the matrix entries. Thus, we do not use it in the matrix powers kernel, only in the tuned SpMV implementation for comparison.

2.1.9 Performance results

In this section, we describe the performance results of our shared-memory parallel matrix powers kernel implementation on the Intel Clovertown platform described in Appendix 3.6, for a set of sparse matrices described in Appendix B.5. We show the data only for this platform and not for the Nehalem platform (also described in Appendix 3.6) to avoid confusion when illustrating points about matrix powers kernel performance. We consider two cases: the Newton basis, and the monomial basis (which is a special case of the Newton basis where all the shifts are zero). Our baseline is the highly optimized shared-memory parallel SpMV implementation of Williams et al. [241], which is equivalent to the $s = 1$ case in the matrix powers kernel.

For calculating speedup in this section, the time taken for the matrix powers kernel is normalized by dividing by s and compared with the time taken for a single SpMV, i.e., the straightforward algorithm. Thus, the speedup as a function of s for a particular matrix is defined as

$$\text{Speedup}(s) = \frac{\text{Time}_{\text{matrix powers kernel}}}{s \cdot \text{Time}_{\text{SpMV}}}. \tag{2.4}$$

Platform Summary

The machine for which we show performance results in this section is an 8-core 2.33 GHz Intel Clovertown processor, described in more detail in Appendix A. The peak double-precision floating-point performance of this processor is 75 Gflop/s. However, because of the overheads associated with sparse matrix storage, peak the performance of an in-cache SpMV computation (for a dense matrix in compressed sparse row format) is only 10 Gflop/s. Note that this in-cache performance number incurs no memory bandwidth cost, since all the data fits in cache. Thus, 10 Gflop/s is an upper bound on the raw flop rate achievable (including redundant computation) when the register tile size is 1×1. However, this upper bound only applies to the monomial basis case. For the Newton basis, we must scale the upper bound

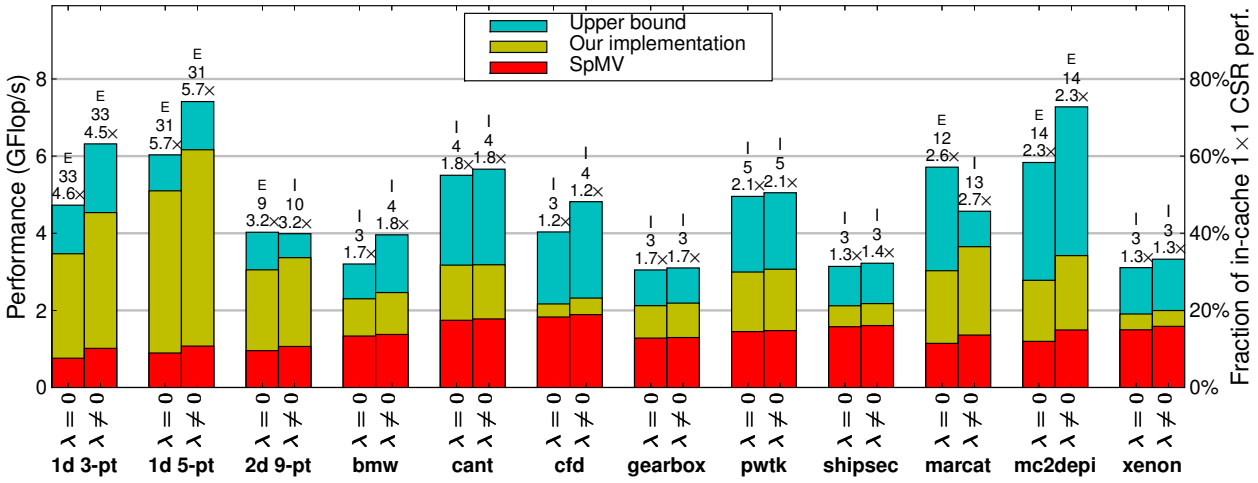


Figure 2.3: Performance of the matrix powers kernel on Intel Clovertown. The yellow bars indicate the best performance over all possible s. Each bar is annotated with some of the parameters for the best performance: whether implicit or explicit ('I' resp. 'E' on top of the bar), the corresponding value of s (in the middle, shown as 'k' on the plot) and the speedup with respect to the straightforward implementation, which is the highly optimized SpMV code. "$\lambda = 0$" indicates the monomial-basis kernel, whereas "$\lambda \neq 0$" indicates the Newton-basis kernel. The "upper bound" indicates the performance predicted by scaling the straightforward implementation's performance by the change in arithmetic intensity.

for each matrix in proportion to the increase in arithmetic intensity due to register tiling. The upper bound for a matrix with n rows and nnz nonzeros is

$$\text{Performance}(n, \text{nnz}) = 10 \cdot \left(1 + \frac{n}{\text{nnz}}\right) \text{ Gflop/s.} \tag{2.5}$$

Performance Results

Figure 2.3 shows the performance of the matrix powers kernel for different sparse matrices on the Intel Clovertown platform. As expected, the speedups for the three matrices corresponding to regular meshes – `1d3pt`, `1d5pt`, and `2d9pt` – are quite good due to the regularity of memory accesses, even though their SpMV performances are among the lowest. We note that the performance of the Newton-basis kernel was better than that for the monomial-basis kernel, simply because of the increase in arithmetic intensity. This difference is marginal when the average number of nonzeros per row of the matrix is large, e.g., the `pwtk` matrix, but is significant when it is small, e.g., the `1d3pt` matrix.

Note that SpMV performance and the best performance across the different matrices is quite different. Therefore, even though `1d5pt` had the second lowest SpMV performance, it was able to achieve the best performance. In fact, `cfd`, which had the best SpMV performance, gains the least from our implementation. We also note that although we get more than 2× speedups for some of the matrices, we are still far below the upper bound of 10 Gflop/s on this platform. As a special case, we note that the optimal performance of `1d5pt` corresponds to a 2×2 register tile, which has an upper bound of 16 Gflop/s. Figure 2.3 also

shows the upper bound on performance corresponding to the optimal basis length s. This upper bound was calculated by scaling the performance of the straightforward implementation, by the factor of increase in arithmetic intensity of the optimized matrix powers kernel over the straightforward implementation. That is, if the optimized matrix powers kernel has arithmetic intensity $\iota_{\text{Optimized}}$ and the straightforward implementation has arithmetic intensity $\iota_{\text{Straightforward}}$, then the upper bound is

$$\frac{\iota_{\text{Optimized}}}{\iota_{\text{Straightforward}}} \cdot \text{Performance}_{\text{Straightforward}}. \tag{2.6}$$

Note that the upper bound of 10 Gflop/s on the raw floating-point arithmetic performance was not reached for any of the matrices, i.e., the upper bound of Equation (2.6) was always lower than 10 Gflop/s on this platform.

We observe that the performance is within 75% of peak for the matrices corresponding to regular meshes. However, for the rest of the matrices the gap between the upper bound and the measured can be quite large – sometimes as much as 60%. For some of the matrices like `marcat` and `mc2depi`, part of this difference comes from the reduced instruction throughput due to the explicit copy for the cache bypass optimization. Since the number of nonzeros per row for these matrices is small, the cost of copying is significant. It is also interesting to note that the implicit algorithm provided the best speedups for most of the matrices. In fact, the explicit algorithm failed to obtain any speedups at all for matrices like `bmw` and `xenon`, as the increase in redundant flops was significant.

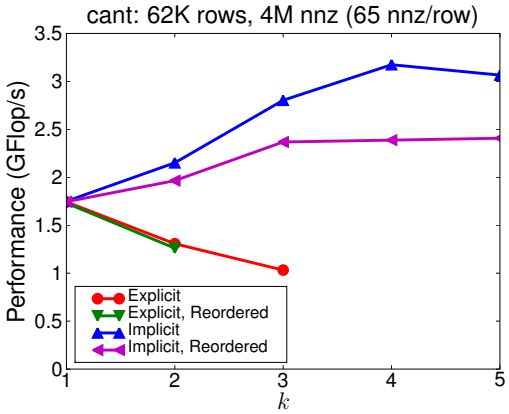
As we stated earlier, our implementation has an autotuning phase where it searches for the cache blocking scheme and other parameters that minimize the matrix powers kernel's runtime. Figure 2.4 illustrates why we need to autotune. We show performance on the Clovertown platform for three of the matrices: `cant`, `mc2depi`, and `pwtk`. We can see that the use of Metis to partition, which resulted in the rows of the matrix and the vectors being reordered, did not always improve performance. For example, for `mc2depi` reordering improved performance for the explicit algorithm, whereas it decreased performance for the implicit algorithm. In fact, for the `cant` matrix, reordering resulted in a significant performance penalty. We also note that the implicit algorithm provided good speedups for `cant` and `pwtk`, whereas the explicit algorithm was actually worse off for $s > 1$. The fact that the density of `cant` and `pwtk` grew quickly with s is also demonstrated by their relatively small optimal $s = 4$. In contrast, performance of `mc2depi` improved for a larger range of s.

2.1.10 Variations

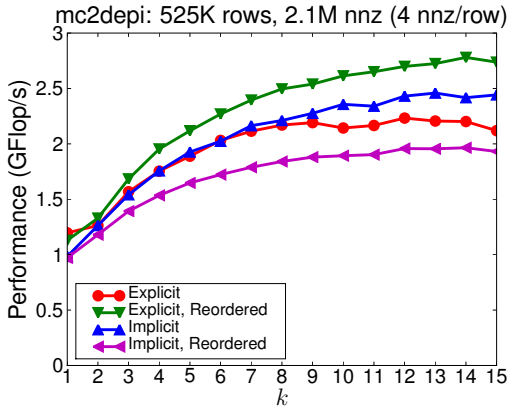
The matrix powers kernel has several variations. We distinguish them by the representation of the sparse matrix, and on whether all the vectors in the Krylov basis, or just the last vector, are needed. We describe both categories of variations in the sections below.

Explicit / implicit structure / values

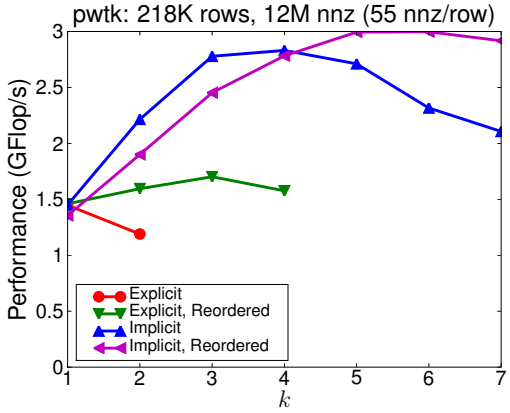
Sparse matrices are called "sparse" because they have many more zero entries than nonzero entries. The different ways of storing sparse matrices exploit this fact in different ways. We can classify the storage representations of sparse matrices into four categories, based on how



(a) Performance vs. s (shown as k in the plot) for the matrix `cant`



(b) Performance vs. s (shown as k in the plot) for the matrix `mc2depi`



(c) Performance vs. s (shown as k in the plot) for the matrix `pwtk`

Figure 2.4: Variation of performance as a function of the basis length s (shown as k in the plots) and the tuning parameters, for three selected matrices. "Explicit" and "Implicit" indicate whether the cache-blocking was explicit (SA2) resp. implicit (SA3). "Reordered" indicates that Metis was used to partition the rows of the matrix and the vectors. The missing points for some of the s values correspond to when the number of redundant flops was so large that no performance gains were possible, so those cases were not timed at all.

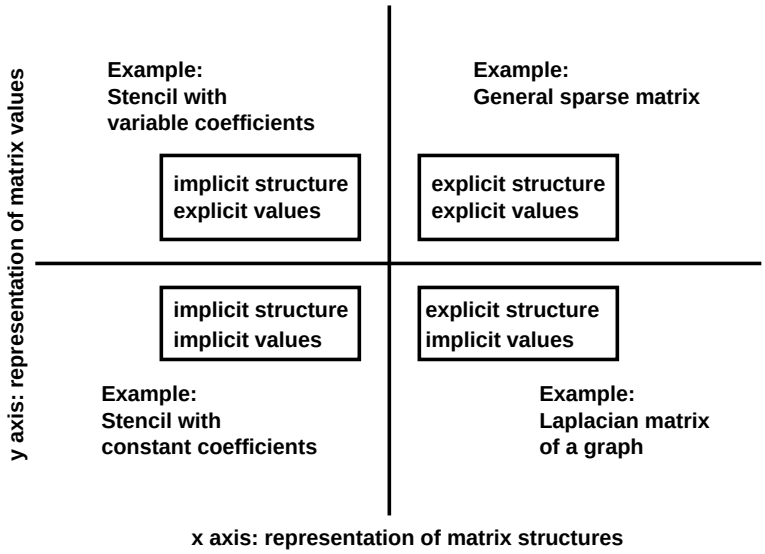


Figure 2.5: Four different types of sparse matrix representations, shown on a Cartesian plane.

they store the nonzero values and their positions in the matrix. These are illustrated by Figure 2.5. The Figure refers both to the *structure* of a sparse matrix, that is locations of its nonzero entries, and to the values of those nonzero entries. Either of these may be stored *explicitly*, that is in a general data structure which can be accessed without further assumptions on the matrix, or *implicitly*, that is not in a data structure, but in a way which must be computed based on assumptions not part of the data structure.

To clarify the distinction between implicit and explicit storage of structure or values, Figure 2.5 gives an example of each of the four categories. Two examples refer to a a *stencil*, which is a regular, repeating geometric arrangement of edges between neighboring vertices in a *regular mesh* (a graph whose vertices correspond to grid points in a subset of Cartesian space with a simple topology). Stencils form a special kind of sparse matrix, whose structure depends only on the regular stencil pattern and the positions of the vertices. Thus, it is not necessary to store the sparse matrix structure corresponding to a stencil explicitly. Stencils may have "constant coefficients," which means that the nonzero values in the matrix corresponding to the stencil edges do not depend on the stencil's position, or "variable coefficients," which means that the nonzero values may change depending on the stencil's position. Constant coefficients need not be explicitly stored, which is why Figure 2.5 calls them "implicit" in that case. A *Laplacian matrix* of a graph (without self-loops or multiple edges) is a (possibly sparse) matrix which has a nonzero entry A_{ij} for every edge e_{ij} between two vertices v_i and v_j in the graph. A_{ij} is the degree of v_i if $i = j$, $A_{ij} = -1$ if $i \neq j$ and edge e_{ij} exists, and $A_{ij} = 0$ otherwise. A Laplacian matrix's structure may correspond to a general graph, whose structure must be stored explicitly. However, the values of the nonzero entries in a Laplacian matrix only depend on the graph structure, so the values need not be stored explicitly. Finally, a "general sparse matrix" needs to have both its structure and values stored explicitly, since no assumptions on either are stated.

The matrix powers kernel algorithms and implementations described in Sections 2.1.5, 2.1.6, and 2.1.7 operate on "general sparse matrices," that is, on sparse matrices whose struc-

ture and nonzero values are both stored explicitly. The algorithms themselves are perfectly valid for the other three ways to represent sparse matrices. When discussing related work in Section 2.1.12, we explain the origins of the matrix powers kernel in compiler optimizations for stencil computations, either for constant or variable coefficients. Some previous work discussed in that section avoids memory hierarchy traffic for general sparse matrices. Our algorithms do so also, and also combine this with avoiding communication between processors in parallel.

As far as we know, no previous work has addressed the case of avoiding communication for sparse matrices like Laplacian graphs (with implicitly stored values and explicitly stored structure). We do not address such sparse matrices in this thesis. It is interesting to note, though, that Laplacian graphs of a sparse matrix's graph structure are sometimes used in partitioning the sparse matrix (see e.g., Pothen et al. [199]), yet partitioning is an essential step in the matrix powers kernel preprocessing phase. This is an interesting chicken-and-egg problem which we only state here and do not attempt to solve.

Last vector or all vectors

Our matrix powers kernel computes all the basis vectors of a particular Krylov subspace. Given A and v, it computes $s+1$ vectors $p_0(A)v$, $p_1(A)v$, $p_2(A)v$, ..., $p_s(A)v$. One variation on this is just to compute the last vector $p_s(A)v$. Some applications only need this last vector. For example, in algebraic multigrid, applying a smoother s times amounts to computing $p_s(\tilde{A})v$ for a particular polynomial $p_s(z)$, where $\tilde{A}$ is related to the original matrix in the linear system $Ax = b$ to solve. (See e.g., Briggs et al. [38] for an explanation of the smoothing step in algebraic multigrid.) The power method (see e.g., Bai et al. [16]) for estimating the largest singular value of a matrix A and its corresponding eigenvector amounts to a similar computation: only the last vector is needed for computing the eigenvalue and eigenvector approximation.

Some previous work that we mention in Section 2.1.12 addresses this "last-vector-only" variation of the matrix powers kernel, either for matrices that come from stencils (see above), or for general sparse matrices (see e.g., Strout et al. [219]). Our matrix powers kernel could be used for this application, but it is designed and optimized for the case in which all the basis vectors are needed.

2.1.11 Related kernels

The matrix powers kernel is one of many kernels intended to exploit reuse of sparse matrix entries in sparse matrix, dense vector products. For example, the tuned library of sparse matrix routines OSKI [237, 236] offers three different kernels:

1. $x \mapsto A^T A x$
2. $x \mapsto (Ax, A^T A x)$
3. $(x, y) \mapsto (Ax, A^T y)$

where A is a general sparse matrix and x and y are dense vectors of appropriate dimensions. All three of these kernels have applications. For example, Kernel 1 may be used to

solve the normal equations via CG (the CGNE method, see e.g., Saad [208]), Kernel 2 for Golub-Kahan-Lanczos bidiagonalization for solving sparse singular value problems (Golub and Kahan [113], see also Bai et al. [16]), and Kernel 3 for nonsymmetric Lanczos [161] and BiCG [99]. (In some cases, the original algorithm must be rearranged slightly to permit use of the optimized kernel.) Vuduc [235] shows how to exploit reuse in the matrix A in these kernels. Their optimizations are complementary to our matrix powers kernel optimizations.

The preprocessing phase of the matrix powers kernel requires partitioning the graph of the sparse matrix into "subdomains," and for each subdomain, finding all nodes which are at distances 1, 2, ..., s from the boundary nodes of the subdomain. This amounts to determining the graphs of A^2, A^3, ..., A^s. One could consider this an approximation of the transitive closure of the graph of A, which is the same as the graph of A^∞. Naturally it is impossible to compute the transitive closure of a general graph just by reading its entries once, because there exist graphs with $O(n)$ edges whose transitive closures have $\Theta(n^2)$ edges. (The graph of a tridiagonal matrix is one example.) Those $\Theta(n^2)$ entries must be read in order to be computed. However, there is a randomized $O(nnz)$ algorithm for estimating the number of edges in the transitive closure of a graph with nnz edges [63]. That algorithm is very different from the matrix powers kernel preprocessing step, but it might be useful for estimating the amount of memory to allocate. Penner and Prasanna [193, 194] present a cache-friendly version of the Floyd-Warshall all-pairs shortest path algorithm, but that algorithm is intended for dense graphs and requires $\theta(n^3)$ comparison operations for a graph with n nodes.

2.1.12 Related work

Precursors of the matrix powers kernel

The matrix powers kernel optimizations summarized in this section generalize known techniques for improving the performance of applying the same stencil repeatedly to a regular discrete domain, or multiplying a vector repeatedly by the same sparse matrix. These techniques are called *tiling* or *blocking*. They involve decomposing the d-dimensional domain into d-dimensional subdomains, and rearranging the order of operations on the domain in order to exploit the parallelism and/or temporal locality implicit in those subdomains. The different order of operations does not change the result in exact arithmetic, although it may affect rounding error in important ways.

Tiling research falls into three general categories. The first encompasses performance-oriented implementations and implementation suggestions, as well as practical performance models thereof. See, for example, [195, 192, 141, 242, 204, 174, 213, 244, 89, 219, 82, 245, 235, 148, 147]. Many of these techniques have been independently rediscovered several times. The second category consists of theoretical algorithms and asymptotic performance analyses. These are based on sequential or parallel processing models which account for the memory hierarchy and/or inter-processor communication costs. Works that specifically discuss stencils or more general sparse matrices include [137], [165], and [222]. Works which may be more generally applicable include [7, 29, 30]. The third category contains suggested applications that call for repeated application of the same stencil (resp. sparse matrix) to a domain (resp. vector). These include papers on s-step Krylov methods, such as [228, 207,

57, 71, 19], as well as papers on repeated smoother applications in multigrid, such as [219].

Tiling was possibly inspired by two existing techniques. The first, *domain decompositions* for solving partial differential equations (PDEs), originated in an 1870 paper by H. A. Schwarz [210]. Typical domain decompositions work directly with the continuous PDE. They iterate between invoking an arbitrary PDE solver on each of the subdomains, and correcting for the subdomain boundary conditions. Unlike tiling, domain decompositions change the solution method (even in exact arithmetic) as well as the order of computations. The second inspiration may be *out-of-core* (OOC) stencil codes,[2] which date back to the early days of electronic computing [146, 195]. Small-capacity primary memories and the lack of a virtual memory system on early computers necessitated operating only on small subsets of a domain at a time.

Tiling of stencil codes became popular for performance as a result of two developments in computer hardware: increasingly deep memory hierarchies, and multiple independent processors [141]. Existing techniques for reorganizing stencil codes on vector supercomputers divided up a d-dimensional domain into hyperplanes of dimension $d-1$. The resulting long streams of data suited the vector architectures of the time, which were typically cacheless and relied on a careful balance between memory bandwidth and floating-point rate in order to achieve maximum performance. In contrast, cache-based architectures usually lack the memory bandwidth necessary to occupy their floating-point units. Achieving near-peak floating-point performance thus requires exploiting locality. Furthermore, iterating over a domain by hyperplanes failed to take advantage of the potential temporal locality of many stencil codes. Tiling decreases the number of connections between subdomains, which reduces the number of cache misses, and decouples data dependencies that hinder extraction of non-vector parallelism.

The idea of using redundant computation to avoid communication or slow memory accesses in stencil codes may be as old as OOC stencil codes themselves. Leiserson et al. cite a reference from 1963 [165, 195], and Hong and Kung analyze a typical strategy on a 2-D grid [137]. (We codify this strategy in our PA1 algorithm.) Nevertheless, many tilings do not involve redundant computation. For example, Douglas et al. describe a parallel tiling algorithm that works on the interiors of the tiles in parallel, and then finishes the boundaries sequentially [89]. Many sequential tilings do not require redundant computations [147]; our SA1 algorithm does not.

However, at least in the parallel case, tilings with redundant computation have the advantage of requiring only a single round of messages, if the stencil is applied several times. The latency penalty is thus independent of the number of applications of the operator, though depending on the mesh or matrix structure, the total number of words communicated may increase somewhat. (For example, for a regular square 2-D mesh, the total number of words sent and received increases by a lower-order term; see e.g., Demmel et al. [77].) Furthermore, Strout et al. point out that the sequential fill-in of boundary regions suggested by Douglas et al. suffers from poor locality [219]. Most importantly, redundant computation to save messages is becoming more and more acceptable, given the divergence in hardware improvements between latency, bandwidth, and floating-point rate.

Application of stencil tiling to more general sparse matrices seems natural, as many types

[2] For some of these codes, "core" is an anachronism, since they predate the widespread use of core memory.

of sparse matrices express the same kind of local connections and domain topology as stencils. However, typical tiling approaches for stencils rely on static code transformations, based on analysis of loop bounds. Sparse matrix-vector multiplication usually requires indirect memory references in order to express the structure of the matrix, and loop bounds are dependent on this structure. Thus, implementations usually require runtime analysis of the sparse matrix structure, using a graph partitioner to decompose the domain implied by the matrix. This was perhaps somewhat alien to the researchers pursuing a compiler-based approach to tuning stencil codes. Furthermore, graph partitioning has a nontrivial runtime cost, and finding an optimal one is an NP-complete problem which must be approximated in practice. Theoretical algorithms for the out-of-core sequential case already existed (see e.g., [165]), but Douglas et al. were apparently the first to attempt an implementation of parallel tiling of a general sparse matrix, in the context of repeated applications of a multigrid smoother [89]. This was extended by Strout et al. into a sequential cache optimization which is most like our SA1 algorithm.

Our work differs from existing approaches in many ways. First, we developed our methods in tandem with an algorithmic justification: communication-avoiding (also called s-step) Krylov subspace methods. Toledo had suggested an s-step variant of conjugate gradient iteration, based on a generalization of PA1, but he did not supply an implementation for matrices more general than tridiagonal matrices [222]. We have a full implementation of PA1 for general sparse matrices, and have demonstrated significant performance increases on a wide variety of platforms. Douglas et al. and Strout developed their matrix powers kernel for classical iterations like Gauss-Seidel [89, 219]. However, these iterations' most common use in modern linear solvers are as multigrid smoothers. The payoff of applying a smoother s times in a row decreases rapidly with s; this is, in fact, the main inspiration for multigrid. Douglas et al. acknowledge that usually $1 \leq s \leq 5$ [89]. In contrast, communication-avoiding Krylov subspace methods are potentially much more scalable in s. Saad also suggested applying something like a matrix powers kernel to polynomial preconditioning, but here again, increasing the degree of the polynomial preconditioner tends to have a decreasing payoff, in terms of the number of CG iterations required for convergence [207].

We have also expanded the space of possible algorithms by including PA2, SA2, SA3, and hybrid algorithms. PA2 requires less redundant computation than PA1, at the cost of less opportunity for overlapping communication and computation. SA2 extends SA1 for the case in which the vectors (as well as the matrix) do not fit entirely in fast memory. SA3 is a variation of SA2 that works well for some classes of sparse matrices, thus offering another tuning parameter (choice of algorithm) for the matrix powers kernel. As far as we can tell, all of these are novel, as well as their hierarchical compositions to form hybrid algorithms. Furthermore, our use of the TSP to optimize the ordering of blocks and of entries within a block in the sequential algorithms is also novel.

Furthermore, our performance models use latency, bandwidth, and floating-point rates from actual or predicted machines. Where possible, these rates were measured rather than peak rates. The models show that our techniques can be useful in a wide variety of situations. These include single-processor out-of-core, distributed-memory parallel, and shared-memory parallel. The latter case is especially relevant considering the advent of multi- and manycore shared-memory parallel processors, and predictions that Moore's Law performance increases will now be expressed in "number of processors per chip." Finally, the models demonstrate

that applying our algorithms may often be just as good or better for performance than improving the communication hardware while using the naive algorithm. This opens the door to parallel configurations previously thought impractical for Krylov methods due to high latency, such as internet-based grid computing or wireless devices.

SpMV communication bounds

Bender et al. [27] develop lower and upper bounds for the number of messages between two levels of the memory hierarchy, for a single SpMV operation $y := A \cdot x$. They do so for two different memory hierarchy models: the *disk access machine* model, which resembles our sequential model except that block transfers between fast and slow memory always have the same number of words B, and the *cache-oblivious* model, which assumes the same two-level model but without specifying the fast memory capacity or block transfer size. They assume that the sparse matrix A is stored as (i, j, A_{ij}) triples, possibly in some index-sorted order. Otherwise, they assume nothing about the structure of the sparse matrix. Since the authors are only concerned about a single SpMV operation in isolation, their communication lower bounds do not apply to our matrix powers kernel.

2.2 Preconditioned matrix powers kernel

The matrix powers kernel summarized in the previous section computes $[p_0(A)v,\ p_1(A)v,\ \ldots,\ p_s(A)v]$ for a sparse matrix A, a vector v, and a set of polynomials $p_0(z),\ \ldots,\ p_s(z)$ with certain properties. The vectors produced form a basis for a Krylov subspace $\text{span}\{v,\ Av,\ A^2v,\ \ldots,\ A^sv\}$ in exact arithmetic, assuming that the Krylov subspace has dimension at least $s + 1$. This kernel is useful for implementing unpreconditioned iterative methods for solving $Ax = \lambda x$ and $Ax = b$. In practice, though, solving $Ax = b$ efficiently with a Krylov method often calls for *preconditioning*. Including a preconditioner changes the required matrix powers kernel, in a way that depends on the iterative method and the form of preconditioning. In this section, we will show the new kernels required, and also present two different approaches to their efficient implementation.

2.2.1 Preconditioning

Preconditioning transforms the linear system $Ax = b$ into another one, with the goal of reducing the number of iterations required for a Krylov method to attain the desired accuracy. (See e.g., [23, 120, 208].) The form of the resulting linear system depends on the type of preconditioner. If M is some nonsingular operator of the same dimensions as A, then *left preconditioning* changes the system to $M^{-1}Ax = M^{-1}b$, and *right preconditioning* changes the system to $AM^{-1}x = M^{-1}b$. The intuition in either case is that if $M^{-1}A \approx I$ in some sense (see the next paragraph), then the iterative method converges quickly.[3] Similarly, if M can be factored as $M = M_1M_2$, then *split preconditioning* changes the system to $M_1^{-1}AM_2^{-1}(M_2x) = M_1^{-1}b$. (Here, the expression M_2x is interpreted as "Solve $M_1^{-1}AM_2^{-1}y = M_1^{-1}b$ using the

[3] According to convention, M is the preconditioner, and "Solve $Mu = v$ for u" is how one applies the preconditioner. This is the standard notation even if we have an explicit representation for M^{-1}.

iterative method, and then solve $M_2 x = y$.") If A and M are self-adjoint and $M_1 = M_2^*$ (as in, for example, an incomplete Cholesky decomposition), then $M_1^{-1} A M_2^{-1}$ is also self-adjoint. We define the *preconditioned matrix* for a left preconditioner as $M^{-1}A$, for a right preconditioner as AM^{-1}, and for a split preconditioner as $M_1^{-1} A M_2^{-1}$. In all cases, in this section we denote the preconditioned matrix by $\tilde{A}$. Context will indicate whether this means left, right, or split preconditioning.

Behind the statement $M^{-1}A \approx I$ lies an entire field of study, which we cannot hope to summarize here. We can only mention certain properties, in order to justify our analysis and conjectures. Often, developing an effective preconditioner requires domain-specific knowledge, either of the physical system which A represents, or of the continuous mathematical problem which the discrete operator A approximates. There is no "general preconditioner" that almost always works well.

One class of preconditioners tries to make the condition number of the preconditioned matrix much smaller than the condition number of A, as a function of some parameter (such as the number of nodes in the discretization) that determines the size of A. The motivation for this approach is that reducing the condition number reduces the worst-case number of iterations required by many preconditioned Krylov methods to achieve a fixed accuracy. Many preconditioners can be proven to reduce the condition number for a certain class of discretizations. The most successful preconditioner of this type is multigrid (when used as a preconditioner for an iterative method, rather than as a direct solver). For some discretizations, multigrid preconditioning can make the condition number a constant relative to the problem size, which means the problem size can grow without significantly increasing the number of iterations. However, not many problems have such preconditioners. We talk about other authors' techniques for reducing communication in multigrid in Section 1.6.4.

Another approach to preconditioning seeks to imitate or approximate the inverse of A, while constraining the sparsity structure of the preconditioner in order to save memory and computation. For example, incomplete factorizations (such as incomplete LU (ILU) or incomplete Cholesky) perform a sparse factorization, but restrict fill-in of the sparse factors. Sparse approximate inverse (SPAI) preconditioners try to minimize the normwise difference between the preconditioned matrix and the identity, while constraining the sparsity structure of the approximate inverse. This is important because the exact inverse of the discrete operator A may be dense, even if A itself is sparse. This is a property shared by many discretizations of continuous problems. We will mention SPAI preconditioners again in Section 2.2.3.

Another class of preconditioners approximates the inverse of a continuous integral equation, in order to approximate or precondition the discretization of that integral equation. Hierarchical matrices are a good example (see e.g., [37, 126, 36]). As mentioned above, the exact inverse of the discrete operator A may be dense, even if A itself is sparse. Furthermore, many integral equation discretizations produce a dense matrix A. However, that matrix has a structure which can be exploited, either to produce an approximate factorization or a preconditioner. We will mention this structure again in Section 2.2.4.

2.2.2 New kernels

Preconditioning changes the linear system, thus changing the Krylov subspace(s) which the iterative method builds. Therefore, preconditioning calls for new matrix powers kernels. Different types of iterative methods require different kernels. These differ from the kernel presented in Demmel et al. [77, Section 7], because there we only consider the monomial basis. For example, for our CA-GMRES algorithm (see Section 3.4.3), the required kernel is

$$\underline{V} = [p_0(\tilde{A})v, p_1(\tilde{A})v, \ldots, p_s(\tilde{A})v]. \tag{2.7}$$

For our preconditioned symmetric Lanczos (Section 4.3) and preconditioned CA-CG algorithms (Sections 5.4 and 5.5) that require A to be self-adjoint, the kernel(s) required depend on whether left, right, or split preconditioning is used. For a split self-adjoint preconditioner with $M_1 = M_2^*$, the kernel is the same as in Equation (2.7); only the preconditioned matrix $\tilde{A}$ is different:

$$\underline{V} = [p_0(\tilde{A})v, p_1(\tilde{A})v, \ldots, p_s(\tilde{A})v]. \tag{2.8}$$

(We do not consider the case where A is self-adjoint but $M_1 \neq M_2^*$.) For left preconditioning, two kernels are needed: one for the *left-side basis*

$$\underline{V} = [p_0(M^{-1}A)v, p_1(M^{-1}A)v, \ldots, p_s(M^{-1}A)v], \tag{2.9}$$

and one for the *right-side basis*

$$\underline{W} = [p_0(AM^{-1})w, p_1(AM^{-1})w, \ldots, p_s(AM^{-1})w], \tag{2.10}$$

where the starting vectors v and w satisfy $Mv = w$. For right preconditioning, the kernels needed are analogous. We explain the reason for this in detail in Section 4.3.1.

A straightforward implementation of each of the kernels (2.8), (2.9), and (2.10) would involve s sparse matrix-vector products with the matrix A, and s preconditioner solves (or $2s$, in the case of the split preconditioner). In this section, we will show how to compute the left-preconditioned kernel with much less communication, if the preconditioner and the sparse matrix satisfy certain structural properties. In particular, we can compute each kernel in parallel for only $1 + o(1)$ times more messages than a single SpMV with A and a single preconditioner solve. We can also compute the left-preconditioned kernel for only $1 + o(1)$ times more words transferred between levels of the memory hierarchy than a single SpMV with A and a single preconditioner solve. (The right-preconditioned kernel uses an analogous technique, so we do not describe it here.) We will describe two different approaches to implementing the left-preconditioned kernel: in Section 2.2.3, an approach that exploits sparsity of the preconditioned matrix, and in Section 2.2.4, an approach that exploits low-rank off-diagonal blocks in a partitioning of the sparse matrix and preconditioner, for the left- or right-preconditioned kernels.

2.2.3 Exploiting sparsity

Our first approach to accelerating the preconditioned kernel exploits sparsity in the matrix and preconditioner. More specifically, for left or right preconditioning, it exploits sparsity

in $M^{-1}A$ and AM^{-1}. For split preconditioning, it exploits sparsity in the preconditioned matrix. This approach in this section should work best for the simplest preconditioners, such as diagonal or block diagonal preconditioners with small blocks. It may also work for more complicated preconditioners that can be constrained to have the same or nearly the same sparsity pattern as the matrix A, such as sparse approximate inverse preconditioners.

All the techniques described in Section 2.1 can be applied in order to compute the preconditioned kernel, if we know the sparsity structure of the preconditioned matrices involved. For example, for the split-preconditioned kernel (Equation (2.8)), the sparsity structure of $M_1^{-1}AM_2^{-1}$ would be needed. For the two left-preconditioned kernels, the techniques of Section 2.1 would be applied separately to both the matrices $M^{-1}A$ (for kernel (2.9)) and AM^{-1} (for kernel (2.10)). This is particularly easy when M^{-1} is (block) diagonal. Note that we may apply the techniques of Section 2.1 to the preconditioned case without needing to compute matrix products like $M^{-1}A$ explicitly; we need only know the sparsity pattern, which gives the communication pattern via the connections between subdomains.

Some sparse approximate inverse preconditioners have a structure which could work naturally with the preconditioned matrix powers kernel. For example, Chow's ParaSails preconditioner [54] constraints the sparsity pattern to that of a small power A^k of the sparse matrix A to precondition. The (unpreconditioned) matrix powers kernel computes the structure of powers of the matrix A anyway, more or less, so one could imagine combining the two structural preprocessing steps to save time. Of course an important part of Chow's preconditioner is the "filtering" step, which removes small nonzeros from the structure of the preconditioner in order to save computation and bandwidth costs, so such a combined structural computation would require significant work in order to be useful. We only propose this as a topic for future investigation.

2.2.4 Exploiting low-rank off-diagonal blocks

Many good preconditioners are in fact dense, even if they are cheap to apply. Again using tridiagonals as a motivating example, consider M as a tridiagonal matrix. Even though tridiagonal systems may be solved in linear time, M^{-1} is dense in general, making our techniques presented so far inapplicable. However, the inverse of a tridiagonal has another important property: any submatrix strictly above or strictly below the diagonal has rank 1. This off-diagonal low-rank property is shared by many good preconditioners, and we may exploit it to avoid communication too.

2.2.5 A simple example

In this section, we use a simple example to illustrate how matrices with low-rank off-diagonal blocks can be used to avoid communication in a parallel version of the matrix powers kernel. We do not consider the sequential version in this example, although it may be developed analogously. Also in this example, we neglect the preconditioner and assume that we want to compute a simple monomial basis x, Ax, A^2x, ..., A^sx with the $n \times n$ matrix A. Demmel et al. [77] explain how to extend this example to the left-preconditioned matrix powers kernel, for the case of the monomial basis.

We assume that the matrix A has the following structure. It is divided into row blocks of $n/2$ rows each among two different processors, Processor 0 and Processor 1. The diagonal blocks A_{00} and A_{11} are general sparse matrices, and the off-diagonal blocks A_{01} and A_{10} are rank r matrices, as shown in Equation (2.11):

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & u_1 \cdot v_1^* \\ u_0 \cdot v_0^* & A_{11} \end{pmatrix} \tag{2.11}$$

This structure can be used to represent many kinds of matrices, both sparse and dense. For example, if A is a tridiagonal matrix, then the rank r of the off-diagonal blocks is 1, the vectors u_0 and v_1 are nonzero multiples of the first column of the identity (e_1), and the vectors u_1 and v_0 are nonzero multiples of column $n/2$ of the identity ($e_{n/2}$).

Sparse matrix-vector multiply

Suppose that we want to compute the SpMV operation $y := A \cdot x$ in parallel using both Processors 0 and 1. Just like the matrix A, we divide up x and y into commensurate row blocks x_0, x_1, y_0, and y_1, in which Processor 0 only has access to x_0 and y_0, and Processor 1 only has access to x_1 and y_1. Equation (2.12) shows the resulting SpMV formula:

$$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} := \begin{pmatrix} A_{00} & u_1 \cdot v_1^* \\ u_0 \cdot v_0^* & A_{11} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}. \tag{2.12}$$

It turns out that Processor 0 only needs access to A_{00}, v_0, and u_1, whereas Processor 1 only needs access to A_{11}, v_1, and u_0. We compute $y := A \cdot x$ as follows:

1. Processor 0 computes the $r \times 1$ vector $\chi_0^{(0)} := v_0^* x_0$, and meanwhile Processor 1 computes the $r \times 1$ vector $\chi_1^{(0)} := v_1^* x_1$.

2. Processor 0 sends $\chi_0^{(0)}$ to Processor 1, and meanwhile Processor 1 sends $\chi_1^{(0)}$ to Processor 0.

3. Processor 0 computes $y_0 := A_{00} x_0 + u_1 \chi_1^{(0)}$, and meanwhile, Processor 1 computes $y_1 := A_{11} x_1 + u_0 \chi_0^{(0)}$.

In the above SpMV procedure, the $n/2$ by r matrix v_0 acts an an operator $x_0 \mapsto v_0^* x_0$ that "restricts" the vector x_0 to be expressed in terms of the right singular vectors of A_{01}. Similarly, v_1 "restricts" x_1 to the right singular vectors of A_{10}. We may therefore call v_0 and v_1 *restriction operators*. Analogously, we may think of u_1 and u_0 as *interpolation* operators, which project the "restricted" data onto the left singular vectors of A_{01} resp. A_{10}. We chose these names for a reason: this matrix representation resembles one level of a simple multigrid-like procedure, in that data from the fine grid is restricted to the coarse grid, and then interpolated back as an additive correction to the fine grid. Of course, the above SpMV procedure lacks many of the elements of multigrid. In particular, the analogous smoothing operation on the "fine grid" would amount to adding in the results of SpMV or SpTS operations with A (or related matrices). This would complicate the above simple SpMV procedure, especially for the matrix powers kernel operation that we explain next. We

will not pursue this analogy further in this thesis, but it may offer a clue towards developing a "communication-avoiding" version of multigrid.[4]

Matrix powers kernel

The "straightforward" approach to computing the matrix powers kernel would involve s invocations of the above SpMV procedure. Each invocation requires each processor to send and receive one message of r words each. Thus, the straightforward approach requires each processor to send and receive $r \cdot s$ messages. It turns out, though, that we can compute the matrix powers kernel with A in the above representation, with each processor only needing to send and receive one message of $r \cdot s$ words each. We will demonstrate this for the simple case of $s = 2$, computing the monomial basis x, Ax, A^2x.

First, we define $\chi_0^{(k)} = v_0^*(A_{00}^k x_0)$ and $\chi_1^{(k)} = v_1^*(A_{00}^k x_1)$ for $k = 0, 1, \ldots, s-1$. (Here, we only use $s = 2$ and therefore $k = 0, 1$, but we introduce the notation to illustrate the generalization to arbitrary s.) Using the same algebra as in Equation (2.12), we find that

$$A^2 x = \begin{pmatrix} A_{00}\left(A_{00}x_0 + u_1\chi_1^{(0)}\right) + u_1\left((v_1^* u_0)\chi_0^{(0)} + \chi_1^{(1)}\right) \\ A_{11}\left(A_{11}x_1 + u_0\chi_0^{(0)}\right) + u_0\left((v_0^* u_1)\chi_1^{(0)} + \chi_0^{(1)}\right) \end{pmatrix}. \tag{2.13}$$

We may write this as a procedure:

1. Processor 0 computes $\chi_0^{(k)} = v_0^*(A_{00}^k x_0)$ for $k = 0, 1$. Meanwhile, Processor 1 computes $\chi_1^{(k)} = v_1^*(A_{11}^k x_1)$ for $k = 0, 1$.
2. Processor 0 sends $\chi_0^{(k)}$ for $k = 0, 1$, bundled into a single message of $2r$ words, to Processor 1. Meanwhile, Processor 1 sends $\chi_1^{(k)}$ for $k = 0, 1$, bundled into a single message of $2r$ words, to Processor 0.
3. Processor 0 computes $y_0 := A_{00}\left(A_{00}x_0 + u_1\chi_1^{(0)}\right) + u_1\left((v_1^* u_0)\chi_0^{(0)} + \chi_1^{(1)}\right)$. Meanwhile, Processor 1 computes $y_1 := A_{11}\left(A_{11}x_1 + u_0\chi_0^{(0)}\right) + u_0\left((v_0^* u_1)\chi_1^{(0)} + \chi_0^{(1)}\right)$.

This only shows this matrix powers kernel for $s = 2$, but by labeling $x_p^{(k)} = (A^k x)_p$ for $p = 0, 1$ and $k = 0, 1, \ldots, s$, it is easy to generalize the third line of the above procedure into a recursion for arbitrary s:

1. Processor 0 computes $\chi_0^{(k)} = v_0^*(A_{00}^k x_0)$ for $k = 0, 1, \ldots, s-1$. Meanwhile, Processor 1 computes $\chi_1^{(k)} = v_1^*(A_{11}^k x_1)$ for $k = 0, 1, \ldots, s-1$.
2. Processor 0 sends $\chi_0^{(k)}$ for $k = 0, 1, \ldots, s-1$, bundled into a single message of rs words, to Processor 1. Meanwhile, Processor 1 sends $\chi_1^{(k)}$ for $k = 0, 1, \ldots, s-1$, bundled into a single message of rs words, to Processor 0.

[4] This idea was developed in summer 2006 during discussions with Dr. Panayot Vassilevski of Lawrence Livermore National Laboratory, Dr. Xiaoye Li of Lawrence Berkeley National Laboratory, Prof. Ming Gu of the University of California Berkeley Mathematics Department, and Prof. Xianlin Xia of Purdue University's Mathematics Department (who was Prof. Gu's student at the time).

3. Processor 0 computes $x_0^{(k)} := A_{00}x_0^{(k-1)} + u_1 \left((v_1^* u_0)\chi_0^{(k-1)} + \chi_1^{(k)} \right)$, for $k = 1, 2, \ldots, s$. Meanwhile, Processor 1 computes $x_1^{(k)} := A_{11}x_1^{(k-1)} + u_0 \left((v_0^* u_1)\chi_1^{(k-1)} + \chi_0^{(k)} \right)$, for $k = 1, 2, \ldots, s$.

The result is the same: each processor only sends one message of rs words, as opposed to the straightforward approach, where each processor sends s messages of r words each.

2.2.6 Problems and solutions

The above simple example only illustrates the low-rank off-diagonal blocks representation of A for 2 processors. Generalizing it to larger numbers of processors requires special care, in order to avoid too much communication. Furthermore, representing the off-diagonal blocks as dense results in more local computation. We will discuss these issues and their solutions below.

Too many messages?

The above simple example only shows the computation for $P = 2$ processors. Generalizing it to larger numbers of processors requires care to avoid too much communication. For example, the straightforward generalization of Equation (2.11) from a 2×2 block matrix into a $P \times P$ matrix of blocks A_{IJ} (with $0 \leq I, J \leq P - 1$) would represent each of the A_{IJ} blocks (for $I \neq J$) as a dense rank-r outer product, whether or not A_{IJ} is zero in the sparse matrix representation of A. As we observed in Section 1.3.1 when discussing parallel SpMV with ordinary sparse matrices, each A_{IJ} represented as a nonzero block entails communication from Processor J to Processor I. If all the A_{IJ} are considered as nonzero blocks in the communication step, then every processor must send a message to every other processor. This would make the number of messages, even in the matrix powers kernel procedure described above, proportional to P^2. This is unacceptable, but thankfully it is unnecessary. For example, if $P = 2$ and A is block diagonal, then there is no need to compute and send the $\chi_0^{(k)}$ and $\chi_1^{(k)}$ terms, since they are zero. Just as with standard sparse matrices, each processor I should only communicate with its "neighbors," that is, with processors J such that A_{IJ} is *actually* nonzero (not just represented in a nonzero way). This is not hard to do.

Too much arithmetic?

A more serious potential problem in the above matrix powers kernel, is that it seems to assume that the outer product representations of the off-diagonal blocks A_{IJ} are dense, even if they are quite sparse. For example, for $P = 2$ processors and a tridiagonal matrix A, there should only be one nonzero entry in each of A_{01} and A_{10}, but the representation of Equation (2.11) requires n words of storage per processor for each of those blocks. This does not increase the number of messages or the volume of communication, but it does increase the amount of local storage and arithmetic.

We will illustrate this for the example of A being tridiagonal, for a general number of processors P and a general matrix powers kernel length s. If we represent A as an ordinary sparse matrix and invoke SpMV s times, each processor must perform $5sn/P + o(sn/P)$

floating-point operations. Suppose instead that we represent A in the $P \times P$ block format, such that for any particular processor I, that only $A_{I,I-1}$ (for $I > 0$), A_{II}, and $A_{I,I+1}$ (for $I < P - 1$) are represented as nonzero rank r outer products. In this case of a tridiagonal matrix A, the rank r of those outer products is 1. Processor I must compute the following quantities:

- $A_{II}^k x_I$ for $k = 0, 1, \ldots, s$, at a cost of $5sn/P + o(sn/P)$ flops (since A_{II} is tridiagonal and n/P by n/P).
- $\chi_I^{(k)} = v_I^*(A_{II}^k x_I)$ for $k = 0, 1, \ldots, s$, at a cost of $2(s+1)n/P + o(sn/P)$ flops (since we have already counted computing $A_{II}^k x_I$).
- $x_I^{(k)} := A_{II} x_I^{(k-1)} + u_{I-1}\chi_{I-1}^{(k-1)} + u_{I+1}\chi_{I+1}^{(k-1)}$, for $k = 1, 2, \ldots, s$. The operation here requires a total of $2sn/P + o(sn/P)$ flops.

The total is $9sn/P + o(sn/P)$ flops, which is almost twice as many as when representing A as a standard sparse matrix and performing s SpMV operations (the "straightforward sparse matrix" approach). The problem comes from the second and third steps, since v_I and u_I are dense n/P by r matrices. If they were represented in a sparse form, the amount of computation would be $5sn/P$ flops per processor, just as in the straightforward sparse matrix approach. Note that sparsity only matters for the computation, not the communication: there is likely little benefit from representing the $r \times 1$ $\chi_I^{(k)}$ vectors sparsely, as they should have no zero entries (not counting possible cancellation) if the rank r was chosen correctly.

The way to fix this problem is, therefore, to represent the restriction and interpolation operators v_I resp. u_I as sparse n/P by r matrices, when performing arithmetic operations with them. Doing so is obvious when A can be represented efficiently as a standard sparse matrix, but not every matrix A that has a natural low-rank off-diagonal blocks representation is naturally sparse. The whole motivation for structures with low-rank off-diagonal blocks – such as semiseparable matrices, hierarchical matrices (see Börm et al. [37] and Hackbusch [126]), and hierarchically semiseparable (HSS) matrices (see e.g., Chandrasekaran et al. [49]) – is to be able to represent dense $n \times n$ matrices with $O(n \log n)$ data, so that matrix-vector multiplication requires only $O(n \log n)$ operations. These structures can be used to represent matrices which are not at all sparse, such as the inverse of a tridiagonal matrix.

If we cannot accept the resulting additional computational requirements of the matrix powers kernel, we have to approximate the dense restriction and interpolation operators by sparse matrices. This relates to the problem of computing a sparse solution to a dense underdetermined linear system of equations. See e.g., Bruckstein et al. [41]. Further investigations along these lines are future work for us; we do not propose any solutions in this thesis.

2.2.7 Related work

In Section 2.1.12, we mentioned how kernels similar to the nonpreconditioned matrix powers kernel appear in computations on regular meshes. Prior work describes how to optimize such kernels via tiling techniques. As far as we know, however, there is no prior work on such optimizations for computations resembling the preconditioned matrix powers kernel.

Authors who developed s-step Krylov methods (a direct precursor to our communication-avoiding KSMs) recognized the need for preconditioning. However, they did not show what basis vectors the KSM should compute, given a preconditioner and a basis other than the monomial basis. Some authors, such as Van Rosendale [228], Chronopoulos and Gear [56], and Toledo [222] suggested using either block diagonal parallel preconditioners that require no communication between processors, or polynomial preconditioners. The latter commute with the matrix A, thus avoiding the problems we discuss in Section 4.3. We show the correct form of the matrix kernel for an arbitrary left or split preconditioner, for many classes of Krylov methods. (Of course, we do not show how to avoid communication for arbitrary preconditioners, only for those that fit into one of two patterns.)

Previous authors developing s-step methods did not suggest how to avoid communication when preconditioning. Saad [207], though not discussing s-step KSMs, did suggest applying something like a matrix powers kernel to polynomial preconditioning, when the matrix has the form of a stencil on a regular mesh. However, polynomial preconditioning tends to have a decreasing payoff with the degree of the polynomial, in terms of the number of CG iterations required for convergence [207]. We are the first, as far as we can tell, to identify two sets of nontrivial preconditioners – left or right preconditioners with nearly the same sparsity structure as the matrix A, and left or right preconditioners with a low-rank off-diagonal blocks structure – for which we can avoid communication in the preconditioned matrix powers kernel.

2.3 Tall Skinny QR

In this section, we present *Tall Skinny QR* (TSQR), a communication-avoiding QR factorization for dense matrices with many more rows than columns. This section summarizes parts of Demmel et al. ([75] and [79]). In parallel, for an $n \times s$ matrix with $n \gg s$, TSQR requires only $O(\log P)$ messages on P processors. This is a factor of $\Theta(s)$ fewer messages than Householder QR or Modified Gram-Schmidt, and attains the lower bound for a matrix whose data is distributed among P processors. Sequentially, TSQR only reads and writes the matrix once, which saves a factor of $\Theta(s)$ words transferred between levels of the memory hierarchy over Householder QR or Modified Gram-Schmidt, and attains the lower bound on communication. In both cases, TSQR has significantly lower latency costs in the parallel case, and significantly lower latency and bandwidth costs in the sequential case, than existing algorithms in LAPACK and ScaLAPACK. TSQR is also numerically stable in the same senses as in LAPACK and ScaLAPACK, both theoretically and in practice.

We have implemented parallel TSQR, sequential TSQR, and a hybrid version of TSQR (with both parallel and sequential optimizations, described in Demmel et al. [79]) on several machines, with significant speedups:

- 6.7× on 16 processors of a Pentium III cluster, for a $100,000 \times 200$ matrix
- 4× on 32 processors of a BlueGene/L, for a $1,000,000 \times 50$ matrix
- 8× on 8 threads of a single node of an Intel Clovertown processor (8 cores, 2 sockets, 4 cores per socket) for a $10,000,000 \times 10$ matrix, compared with LAPACK's QR factorization `DGEQRF` (see [79] for details)

Some of this speedup for the cases is enabled by TSQR being able to use a much better local QR decomposition than ScaLAPACK's QR factorization routine `PDGEQRF` can use, such as the recursive variant by Elmroth and Gustavson (see [93]. We have also implemented sequential TSQR on a laptop for matrices that do not fit in DRAM, so that slow memory is disk. This requires a special implementation in order to run at all, since virtual memory does not accommodate matrices of the sizes we tried. By extrapolating runtime from matrices that do fit in DRAM, we can say that our out-of-DRAM implementation was as little as $2\times$ slower than the predicted runtime as though DRAM were infinite.

We begin our discussion of TSQR with Section 2.3.1, where we explain our motivations for developing a communication-avoiding QR factorization of tall skinny matrices. In Section 2.3.2, we explain how the algorithm works in the parallel and sequential cases, and how it can be easily adapted to different computer architectures. In Section 2.3.3, we compare the numerical stability of TSQR to that of other QR factorization methods for tall skinny matrices. Finally, in Section 2.3.4, we show performance results from our parallel shared-memory implementation of TSQR.

2.3.1 Motivation for TSQR

Communication-avoiding Krylov methods

Our main motivation for developing a communication-avoiding QR factorization of tall skinny matrices was for orthogonalizing Krylov subspace basis vectors, in our communication-avoiding versions of Arnoldi (Section 3.3), GMRES (Section 3.4), and Lanczos (Section 4.2). We combine TSQR with Block Gram-Schmidt (BGS – Section 2.4), both for orthogonalization, and for reorthogonalization if necessary. We show how to do reorthogonalization in an accurate and communication-avoiding way, using TSQR and BGS, in Sections 2.4.6, 2.4.7, and 2.4.8. See especially Algorithm 14 in the latter section.

Block Krylov methods

TSQR has other applications as well. Block Krylov methods are one example, which we discuss in more detail in Section 1.6.2. These methods include algorithms for solving linear systems $Ax = B$ with multiple right-hand sides (such as variants of GMRES, QMR, or CG [233, 105, 183]), as well as block iterative eigensolvers (for a summary of such methods, see [15, 163]). Many of these methods have widely used implementations, on which a large community of scientists and engineers depends for their computational tasks. Examples include TRLAN (Thick Restart Lanczos), BLZPACK (Block Lanczos), Anasazi (various block methods), and PRIMME (block Jacobi-Davidson methods) [246, 173, 155, 13, 21, 215]. Eigenvalue computation is particularly sensitive to the accuracy of the orthogonalization; two recent papers suggest that large-scale eigenvalue applications require a stable QR factorization [132, 156].

Panel factorization in general QR

Householder QR decompositions of tall and skinny matrices also comprise the panel factorization step for typical QR factorizations of matrices in a more general, two-dimensional

layout. This includes the current parallel QR factorization routine `PDGEQRF` in ScaLAPACK, as well as ScaLAPACK's experimental `PFDGEQRF` parallel routine for factoring matrices that do not fit in memory. Both algorithms use a standard column-based Householder QR for the panel factorizations, but in the parallel case this is a latency bottleneck, and in the out-of-DRAM case it is a bandwidth bottleneck. Replacing the existing panel factorization with TSQR reduces this cost by a factor equal to the number of columns in a panel, thus removing the bottleneck. We present the resulting general QR factorization as the Communication-Avoiding QR (CAQR) algorithm in Demmel et al. [75], where we show that it attains lower bounds on communication in both the parallel and sequential cases. We do not discuss CAQR further in this thesis, though the combination of TSQR and BGS has performance similar to CAQR.

2.3.2 TSQR algorithms

In this section, we illustrate the insight behind TSQR, and use it to explain variants of TSQR that avoid communication in different circumstances. TSQR uses a reduction-like operation to compute the QR factorization of an $m \times n$ matrix A, stored in a 1-D block row layout.[5] The shape of the reduction tree may be chosen to avoid communication for different computer architectures. For example, TSQR on a binary tree minimizes the number of messages in parallel, and TSQR on a linear tree minimizes memory traffic in the sequential case. We illustrate both with diagrams and simple examples. Finally, we show two examples of more complicated reduction trees. We implemented one of these (sequential nested within parallel), which as we describe in Demmel et al. [79] achieved significant speedups over alternative parallel QR factorizations on two different 8-core Intel processors.

Parallel TSQR on a binary tree

The basic idea of using a reduction on a binary tree to compute a tall skinny QR factorization has been rediscovered multiple times. See e.g., e.g., Pothen and Raghavan [198] and da Cunha et al. [67]. (TSQR was also suggested by Golub et al. [115], but they did not reduce the number of messages from $n \log P$ to $\log P$.) Here, we repeat the basic insight in order to show its generalization to a whole space of algorithms, which is one of our novel contributions. Suppose that we want to factor the $m \times n$ matrix A, with $m \gg n$, on a parallel computer with $P = 4$ processors. First, we decompose A into four $m/4 \times n$ block rows:

$$A = \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix},$$

[5] The ScaLAPACK Users' Guide has a good explanation of 1-D and 2-D block and block cyclic layouts of dense matrices [33]. In particular, refer to the section entitled "Details of Example Program #1."

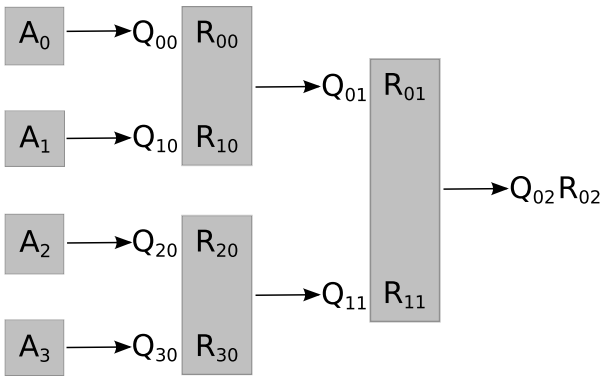


Figure 2.6: Execution of the parallel TSQR factorization on a binary tree of four processors. The gray boxes indicate where local QR factorizations take place. The Q and R factors each have two subscripts: the first is the sequence number within that stage, and the second is the stage number.

such that processor q "owns" block A_q. Then, we independently compute the QR factorization of each block row on each processor, any fast and accurate sequential QR factorization:

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \begin{pmatrix} Q_{00}R_{00} \\ Q_{10}R_{10} \\ Q_{20}R_{20} \\ Q_{30}R_{30} \end{pmatrix}.$$

This is "stage 0" of the computation, hence the second subscript 0 of the Q and R factors. The first subscript indicates the block index at that stage. (Abstractly, we use the Fortran convention that the first index changes "more frequently" than the second index.) Stage 0 operates on the $P = 4$ leaves of the tree. We can write this decomposition instead as a block diagonal orthogonal matrix times a column of blocks:

$$A = \begin{pmatrix} Q_{00}R_{00} \\ Q_{10}R_{10} \\ Q_{20}R_{20} \\ Q_{30}R_{30} \end{pmatrix} = \left(\begin{array}{c|c|c|c} Q_{00} & & & \\ \hline & Q_{10} & & \\ \hline & & Q_{20} & \\ \hline & & & Q_{30} \end{array}\right) \cdot \left(\begin{array}{c} R_{00} \\ \hline R_{10} \\ \hline R_{20} \\ \hline R_{30} \end{array}\right),$$

although we do not have to store it this way. After this stage 0, there are $P = 4$ of the R factors. We group them into successive pairs $R_{i,0}$ and $R_{i+1,0}$, and do the QR factorizations of grouped pairs in parallel:

$$\left(\begin{array}{c} R_{00} \\ R_{10} \\ \hline R_{20} \\ R_{30} \end{array}\right) = \left(\begin{array}{c} \begin{pmatrix} R_{00} \\ R_{10} \end{pmatrix} \\ \hline \begin{pmatrix} R_{20} \\ R_{30} \end{pmatrix} \end{array}\right) = \left(\begin{array}{c} Q_{01}R_{01} \\ \hline Q_{11}R_{11} \end{array}\right).$$

As before, we can rewrite the last term as a block diagonal orthogonal matrix times a column of blocks:

$$\left(\begin{array}{c} Q_{01}R_{01} \\ \hline Q_{11}R_{11} \end{array}\right) = \left(\begin{array}{c|c} Q_{01} & \\ \hline & Q_{11} \end{array}\right) \cdot \left(\begin{array}{c} R_{01} \\ \hline R_{11} \end{array}\right).$$

This is stage 1, as the second subscript of the Q and R factors indicates. We iteratively perform stages until there is only one R factor left, which is the root of the tree:

$$\begin{pmatrix} R_{01} \\ R_{11} \end{pmatrix} = Q_{02} R_{02}.$$

Equation (2.14) shows the whole factorization:

$$A = \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \left(\begin{array}{c|c|c|c} Q_{00} & & & \\ \hline & Q_{10} & & \\ \hline & & Q_{20} & \\ \hline & & & Q_{30} \end{array}\right) \cdot \left(\begin{array}{c|c} Q_{01} & \\ \hline & Q_{11} \end{array}\right) \cdot Q_{02} \cdot R_{02}, \tag{2.14}$$

in which the product of the first three matrices has orthogonal columns, since each of these three matrices does. Note the binary tree structure in the nested pairs of R factors.

Figure 2.6 illustrates the binary tree on which the above factorization executes. Gray boxes highlight where local QR factorizations take place. By "local," we refer to a factorization performed by any one processor at one node of the tree; it may involve one or more than one block row. If we were to compute all the above Q factors explicitly as square matrices, each of the Q_{i0} would be $m/P \times m/P$, and Q_{ij} for $j > 0$ would be $2n \times 2n$. The final R factor would be upper triangular and $m \times n$, with $m - n$ rows of zeros. In a "thin QR" factorization, in which the final Q factor has the same dimensions as A, the final R factor would be upper triangular and $n \times n$. In practice, we prefer to store all the local Q factors implicitly until the factorization is complete. In that case, the implicit representation of Q_{i0} fits in an $m/P \times n$ lower triangular matrix, and the implicit representation of Q_{ij} (for $j > 0$) fits in an $n \times n$ lower triangular matrix (due to optimizations that we discuss in Demmel et al. [75]).

Note that the maximum per-processor memory requirement is $\max\{mn/P, n^2 + O(n)\}$, since any one processor need only factor two $n \times n$ upper triangular matrices at once, or a single $m/P \times n$ matrix.

Sequential TSQR on a flat tree

Sequential TSQR uses a similar factorization process, but with a "flat tree" (a linear chain). It may also handle the leaf nodes of the tree slightly differently, as we will show below. Again, the basic idea is not new; see e.g., [43, 44, 124, 160, 200, 201]. (Some authors (e.g., [43, 160, 200]) refer to sequential TSQR as "tiled QR." We use the phrase "sequential TSQR" because both our parallel and sequential algorithms could be said to use tiles.) In particular, Gunter and van de Geijn develop a parallel out-of-DRAM QR factorization algorithm that uses a flat tree for the panel factorizations [124]. Buttari et al. suggest using a QR factorization of this type to improve performance of parallel QR on commodity multicore processors [43]. Quintana-Orti et al. develop two variations on block QR factorization algorithms, and use them with a dynamic task scheduling system to parallelize the QR factorization on shared-memory machines [200].

Kurzak and Dongarra use similar algorithms, but with static task scheduling, to parallelize the QR factorization on Cell processors [160]. (There, the parallelization happens only

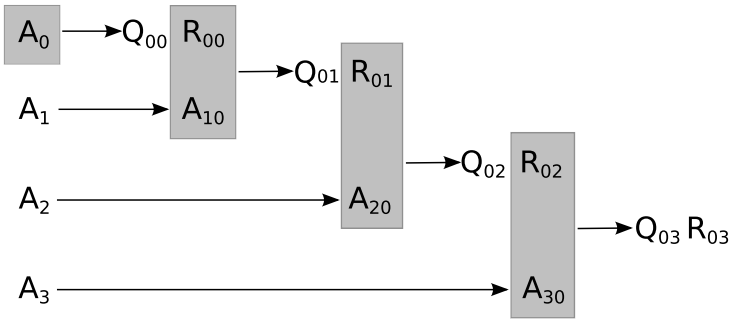


Figure 2.7: Execution of the sequential TSQR factorization on a flat tree with four submatrices. The gray boxes indicate where local QR factorizations take place The Q and R factors each have two subscripts: the first is the sequence number for that stage, and the second is the stage number.

in the trailing matrix updates, not in the panel factorization. Recent work by these authors has shown that this does not expose sufficient parallelism, so that parallel TSQR is required for the panel factorization.)

We will show that the basic idea of sequential TSQR fits into the same general framework as the parallel QR decomposition illustrated above, and also how this generalization expands the tuning space of QR factorization algorithms. In addition, we will develop detailed performance models of sequential TSQR and the current sequential QR factorization implemented in LAPACK.

We start with the same block row decomposition as with parallel TSQR above:

$$A = \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix}$$

but begin with a QR factorization of A_0, rather than of all the block rows:

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \begin{pmatrix} Q_{00}R_{00} \\ A_1 \\ A_2 \\ A_3 \end{pmatrix}.$$

This is "stage 0" of the computation, hence the second subscript 0 of the Q and R factor. We retain the first subscript for generality, though in this example it is always zero. We can write this decomposition instead as a block diagonal matrix times a column of blocks:

$$\left(\begin{array}{c} Q_{00}R_{00} \\ A_1 \\ A_2 \\ A_3 \end{array}\right) = \left(\begin{array}{c|c|c|c} Q_{00} & & & \\ \hline & I & & \\ \hline & & I & \\ \hline & & & I \end{array}\right) \cdot \left(\begin{array}{c} R_{00} \\ \hline A_1 \\ \hline A_2 \\ \hline A_3 \end{array}\right).$$

We then combine R_{00} and A_1 using a QR factorization:

$$\begin{pmatrix} R_{00} \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \left(\begin{array}{c} R_{00} \\ \hline A_1 \\ \hline A_2 \\ A_3 \end{array}\right) = \left(\begin{array}{c} Q_{01}R_{01} \\ \hline A_2 \\ A_3 \end{array}\right)$$

This can be rewritten as a block diagonal matrix times a column of blocks:

$$\left(\begin{array}{c} Q_{01}R_{01} \\ \hline A_2 \\ A_3 \end{array}\right) = \left(\begin{array}{c|c|c} Q_{01} & & \\ \hline & I & \\ \hline & & I \end{array}\right) \cdot \left(\begin{array}{c} R_{01} \\ \hline A_2 \\ \hline A_3 \end{array}\right).$$

We continue this process until we run out of A_i factors. The resulting factorization has the following structure:

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \left(\begin{array}{c|c|c|c} Q_{00} & & & \\ \hline & I & & \\ \hline & & I & \\ \hline & & & I \end{array}\right) \cdot \left(\begin{array}{c|c|c} Q_{01} & & \\ \hline & I & \\ \hline & & I \end{array}\right) \cdot \left(\begin{array}{c|c|c} I & & \\ \hline & Q_{02} & \\ \hline & & I \end{array}\right) \cdot \left(\begin{array}{c|c|c} I & & \\ \hline & I & \\ \hline & & Q_{03} \end{array}\right) R_{30}. \quad (2.15)$$

Here, the A_i blocks are $m/P \times n$. If we were to compute all the above Q factors explicitly as square matrices, then Q_{00} would be $m/P \times m/P$ and Q_{0j} for $j > 0$ would be $2m/P \times 2m/P$. The above I factors would be $m/P \times m/P$. The final R factor, as in the parallel case, would be upper triangular and $m \times n$, with $m - n$ rows of zeros. In a "thin QR" factorization, in which the final Q factor has the same dimensions as A, the final R factor would be upper triangular and $n \times n$. In practice, we prefer to store all the local Q factors implicitly until the factorization is complete. In that case, the implicit representation of Q_{00} fits in an $m/P \times n$ lower triangular matrix, and the implicit representation of Q_{0j} (for $j > 0$) fits in an $m/P \times n$ lower triangular matrix as well (due to optimizations that we discuss in Demmel et al. [75]).

Figure 2.7 illustrates the flat tree on which the above factorization executes. Gray boxes highlight where "local" QR factorizations take place.

The sequential algorithm differs from the parallel one in that it does not factor the individual blocks of the input matrix A, excepting A_0. This is because in the sequential case, the input matrix has not yet been loaded into working memory. In the fully parallel case, each block of A resides in some processor's working memory. It then pays to factor all the blocks before combining them, as this reduces the volume of communication (only the triangular R factors need to be exchanged) and reduces the amount of arithmetic performed at the next level of the tree. In contrast, the sequential algorithm never writes out the intermediate R factors, so it does not need to convert the individual A_i into upper triangular factors. Factoring each A_i separately would require writing out an additional Q factor for each block of A. It would also add another level to the tree, corresponding to the first block A_0.

Note that the maximum per-processor memory requirement is $mn/P + n^2/2 + O(n)$, since only an $m/P \times n$ block and an $n \times n$ upper triangular block reside in fast memory at one time. We could save some fast memory by factoring each A_i block separately before

combining it with the next block's R factor, as long as each block's Q and R factors are written back to slow memory before the next block is loaded. One would then only need to fit no more than two $n \times n$ upper triangular factors in fast memory at once. However, this would result in more writes, as each R factor (except the last) would need to be written to slow memory and read back into fact memory, rather than just left in fast memory for the next step.

In both the parallel and sequential algorithms, a vector or matrix is multiplied by Q or Q^T by using the implicit representation of the Q factor, as shown in Equation (2.14) for the parallel case, and Equation (2.15) for the sequential case. This is analogous to using the Householder vectors computed by Householder QR as an implicit representation of the Q factor.

TSQR on general trees

The above two algorithms are extreme points in a large set of possible QR factorization methods, parametrized by the tree structure. Our version of TSQR is novel because it works on any tree. In general, the optimal tree may depend on both the architecture and the matrix dimensions. This is because TSQR is a reduction (as we discuss in detail in Demmel et al. [75]). Trees of types other than binary often result in better reduction performance, depending on the architecture (see e.g., [178]). Throughout this paper, we discuss two examples – the binary tree and the flat tree – as easy extremes for illustration. We will show that the binary tree minimizes the number of stages and messages in the parallel case, and that the flat tree minimizes the number and volume of input matrix reads and writes in the sequential case. We show in Demmel et al. [75] how to perform TSQR on any tree. Methods for finding the best tree in the case of TSQR are future work. Nevertheless, we can identify two regimes in which a "nonstandard" tree could improve performance significantly: parallel memory-limited CPUs, and large distributed-memory supercomputers.

The advent of desktop and even laptop multicore processors suggests a revival of parallel out-of-DRAM algorithms, for solving cluster-sized problems while saving power and avoiding the hassle of debugging on a cluster. TSQR could execute efficiently on a parallel memory-limited device if a sequential flat tree were used to bring blocks into memory, and a parallel tree (with a structure that reflects the multicore memory hierarchy) were used to factor the blocks. Figure 2.8 shows an example with 16 blocks executing on four processors, in which the factorizations are pipelined for maximum utilization of the processors. The algorithm itself needs no modification, since the tree structure itself encodes the pipelining. This is, we believe, a novel extension of the parallel out-of-core QR factorization of Gunter et al. [124].

TSQR's choice of tree shape can also be optimized for modern supercomputers. A tree with different branching factors at different levels could naturally accommodate the heterogeneous communication network of a cluster of multicores. The subtrees at the lowest level may have the same branching factor as the number of cores per node (or per socket, for a multisocket shared-memory architecture).

Note that the maximum per-processor memory requirement of all TSQR variations is bounded above by

$$\frac{qn(n+1)}{2} + \frac{mn}{P},$$

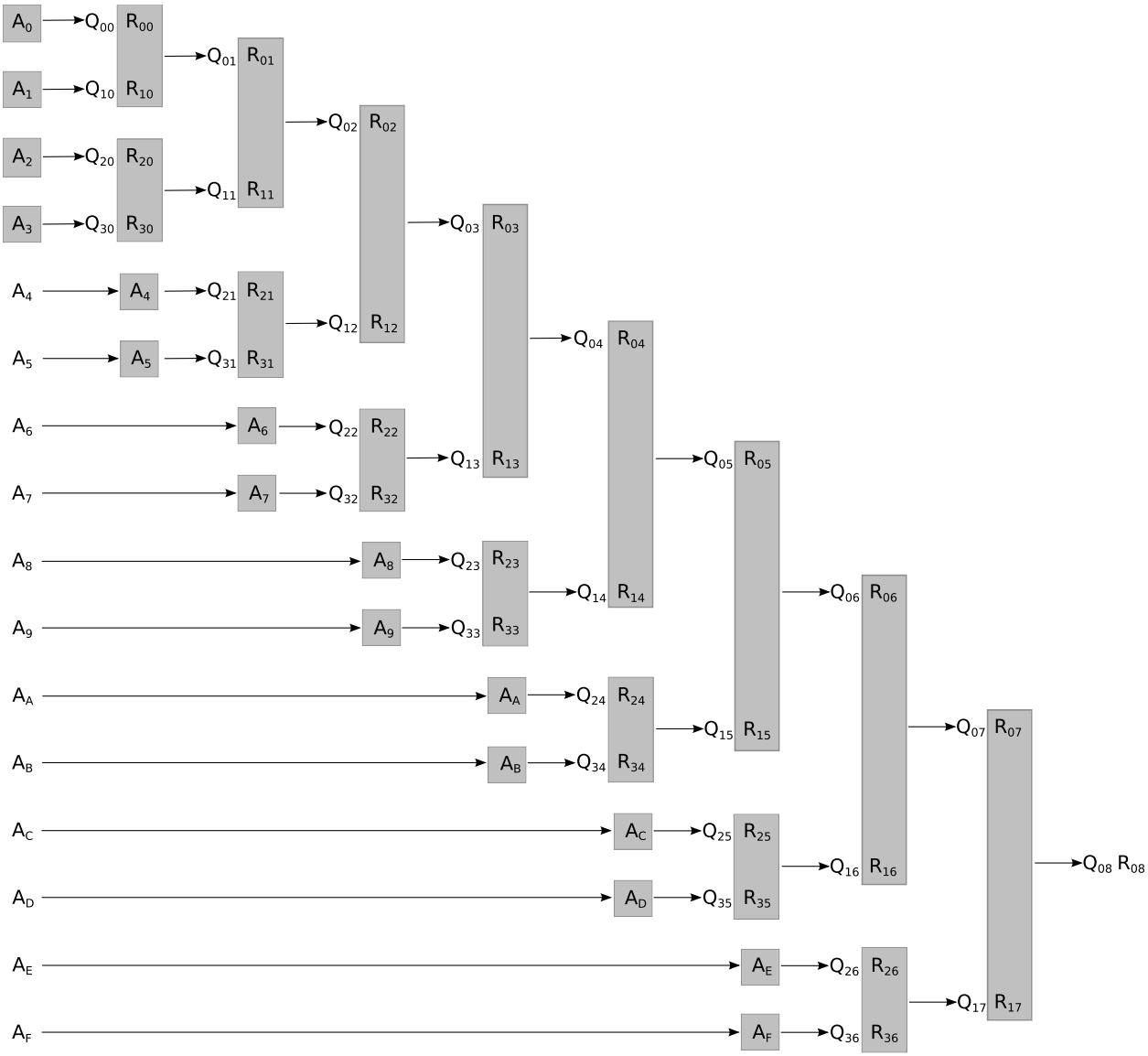


Figure 2.8: Execution of a hybrid parallel / out-of-core TSQR factorization. The matrix has 16 blocks, and four processors can execute local QR factorizations simultaneously. The gray boxes indicate where local QR factorizations take place. We number the blocks of the input matrix A in hexadecimal to save space (which means that the subscript letter A is the number 10_{10}, but the non-subscript letter A is a matrix block). The Q and R factors each have two subscripts: the first is the sequence number for that stage, and the second is the stage number.

in which q is the maximum branching factor in the tree.

2.3.3 Accuracy and stability

In Demmel et al. [75], we model the performance of various QR factorization algorithms for tall and skinny matrices on a block row layout. Our models there show that CholeskyQR (see Section 2.5) should have better performance than all the other methods. However, numerical accuracy is also an important consideration for many users. For example, in CholeskyQR, the loss of orthogonality of the computed Q factor depends quadratically on the condition number of the input matrix (see Table 2.2). This is because computing the Gram matrix $A^T A$ squares the condition number of A. One can avoid this stability loss by computing and storing $A^T A$ in doubled precision. However, this doubles the communication volume. It also increases the cost of arithmetic operations by a hardware-dependent factor.

Algorithm	$\|I - Q^T Q\|_2$ bound	Assumption on $\kappa(A)$	Reference(s)
Householder QR	$O(\varepsilon)$	None	[114]
TSQR	$O(\varepsilon)$	None	[114]
CGS2	$O(\varepsilon)$	$O(\varepsilon\kappa(A)) < 1$	[1, 150]
MGS	$O(\varepsilon\kappa(A))$	None	[32]
CholeskyQR	$O(\varepsilon\kappa(A)^2)$	None	[216]
CGS	$O(\varepsilon\kappa(A)^{n-1})$	None	[150, 211]

Table 2.2: Upper bounds on deviation from orthogonality of the Q factor from various QR algorithms. Machine precision is ε. "Assumption on $\kappa(A)$" refers to any constraints which $\kappa(A)$ must satisfy in order for the accuracy bound in the previous column to hold (for example, that the matrix must be numerically full rank).

Unlike CholeskyQR or the two variants of Gram-Schmidt orthogonalization presented in Table 2.2, Householder QR is *unconditionally stable*. That is, the computed Q factors are always orthogonal to machine precision, regardless of the properties of the input matrix [114]. This also holds for TSQR, because the algorithm is composed entirely of no more than P Householder QR factorizations, in which P is the number of input blocks. Each of these factorizations is itself unconditionally stable. In contrast, the orthogonality of the Q factor computed by CGS, MGS, or CholeskyQR depends on the condition number of the input matrix. Reorthogonalization in MGS and CGS can make the computed Q factor orthogonal to machine precision, but only if the input matrix A is numerically full rank, i.e., if $O(\varepsilon\kappa(A)) < 1$. Reorthogonalization also doubles the cost of the algorithm.

However, sometimes some loss of accuracy can be tolerated, either to improve performance, or for the algorithm to have a desirable property. For example, in some cases the input vectors are sufficiently well-conditioned to allow using CholeskyQR, and the accuracy of the orthogonalization is not so important. Another example is GMRES. Its backward stability was proven first for Householder QR orthogonalization, and only later for modified Gram-Schmidt orthogonalization [121]. Users traditionally prefer the latter formulation, mainly because the Householder QR version requires about twice as many floating-point operations (as the Q matrix must be computed explicitly). Another reason is that most

GMRES descriptions make the vectors available for orthogonalization one at a time, rather than all at once, as Householder QR would require (see e.g., [238]). Thus, it must use a version of Gram-Schmidt orthogonalization. Our CA-GMRES algorithm (Section 3.4) orthogonalizes groups of s basis vectors at a time, which is why it can use TSQR instead.

We care about stability for two reasons. First, an important application of TSQR is the orthogonalization of basis vectors in Krylov methods. When using Krylov methods to compute eigenvalues of large, ill-conditioned matrices, the whole solver can fail to converge or have a considerably slower convergence when the orthogonality of the Ritz vectors is poor [132, 156]. Second, in Demmel et al. [75], we develop CAQR, which uses TSQR as the panel factorization for computing the QR decomposition of matrices of general shape. Basing CAQR on TSQR, rather than say CholeskyQR, means that CAQR is just as accurate as HouseholderQR.

Table 2.2 summarizes known upper bounds on the deviation from orthogonality $\|I - Q^T Q\|_2$ of the computed Q factor, as a function of the machine precision ε and the input matrix's two-norm condition number $\kappa(A)$, for various QR factorization algorithms. Except for CGS, all these bounds are sharp. Smoktunowicz et al. demonstrate a matrix satisfying $O(\varepsilon\kappa(A)^2) < 1$ for which $\|I - Q^T Q\|_2$ is not $O(\varepsilon\kappa(A)^2)$, but as far as we know, no matrix has yet been found for which the $\|I - Q^T Q\|_2$ is $O(\varepsilon\kappa(A)^{n-1})$ bound is sharp [211].

In the table, "CGS2" refers to classical Gram-Schmidt with one reorthogonalization pass. A single reorthogonalization pass suffices to make the Q factor orthogonal to machine precision, as long as the input matrix is numerically full rank, i.e., if $O(\varepsilon\kappa(A)) < 1$. This is the source of Kahan's maxim, "Twice is enough" [189]: the accuracy reaches its theoretical best after one reorthogonalization pass (see also [1]), and further reorthogonalizations do not improve orthogonality. However, TSQR needs only half as many messages to do just as well as CGS2. In terms of communication, TSQR's stability comes for free.

2.3.4 TSQR performance

Our communication-avoiding Krylov methods CA-Arnoldi, CA-GMRES, and CA-Lanczos (Sections 3.3, 3.4, resp. 4.2) require a fast but accurate method for orthogonalizing a group of $s + 1$ Krylov basis vectors. The standard versions of these Krylov methods use Modified Gram-Schmidt (MGS) to orthogonalize their basis vectors,[6] but this requires asymptotically more messages and synchronization in parallel, and asymptotically more data traffic between levels of the memory hierarchy, than TSQR. GMRES and Arnoldi can benefit especially from a faster orthogonalization method, since the full orthogonalization used by these methods often dominates the cost of the method, for restart lengths used in practice. Furthermore, the data dependencies in standard GMRES, Arnoldi, and Lanczos force these algorithms to use orthogonalization methods based on vector-vector or matrix-vector operations, rather than the matrix-matrix operations used by TSQR.

We implemented a parallel shared-memory version of TSQR, as part of a CA-GMRES solver. We first described this solver in Demmel et al. [79], and we report performance results for the whole solver in Section 3.6 of this thesis. In this section, we report perfor-

[6] As we discuss in Chapter 4, the Lanczos recurrence is a truncated version of the Arnoldi recurrence, which itself is a particular form of MGS.

mance results just for that TSQR implementation. TSQR achieved speedups of up to 3.3× over MGS, and up to 2.7× over LAPACK's QR factorization `DGEQRF`, on the suite of test problems, for the same number (8) of processors. TSQR costs here include both computing the factorization and forming the explicit Q factor, which means that TSQR must read the vectors to orthogonalize twice; nevertheless, TSQR is still always faster than MGS for CA-GMRES. In addition, TSQR works naturally with the blocked output of the matrix powers kernel (see Section 2.1) in CA-GMRES (see Section 3.4 for the algorithm, and Section 3.6 for performance results). LAPACK's or ScaLAPACK's QR factorizations would require an additional unpacking step before they could use such data.

Hybrid algorithm

The TSQR factorization described in this section is a hybrid of the sequential and parallel TSQR algorithms described in Section 2.3.2. It begins with an $m \times n$ dense matrix with $m \gg n$, divided into blocks of rows. In our case, each block consists of those components of a cache block from the matrix powers kernel that do not overlap with another cache block. TSQR distributes the blocks so the P processors get disjoint sets of blocks. (The implementation lets each processor initialize its own blocks. In the case of NUMA hardware, this associates that block to a page of memory local to that processor, if the operating system observes the "first touch" memory affinity assignment policy.) Then, each processor performs sequential TSQR on its set of blocks in a sequence of steps, one per block. Each intermediate step requires combining a small $n \times n$ R factor from the previous step with the current block, by factoring the two matrices "stacked" on top of each other. We improve the performance of this factorization by a factor of about two by performing it in place, rather than copying the R factor and the current cache block into a working block and running LAPACK's standard QR factorization on it. The sequential TSQR algorithms running on the P processors require no synchronization, because the cores operate on disjoint sets of data. Once all P processors are done with their sets of blocks, P small R factors are left. The processors first synchronize, and then one processor stacks these into a single $nP \times n$ matrix and invokes LAPACK's QR factorization on it. As this matrix is small for P and n of interest, parallelizing this step is not worth the synchronization overhead. The result of this whole process is a single $n \times n$ R factor, and a Q factor which is implicitly represented as a collection of orthogonal operators. Assembling the Q factor in explicit form uses almost the same algorithm, but in reverse order.

Implementation details

TSQR invokes an ordinary Householder QR factorization at least once per processor (the factor that processor's first cache block), and then once more to combine the processors' R factors. Ordinarily, TSQR could use any QR implementation. However, CA-Arnoldi, CA-GMRES, and CA-Lanczos (Sections 3.3, 3.4, resp. 4.2) all require that the R factor produced by the QR factorization has a nonnegative real diagonal. LAPACK has only satisfied this requirement since version 3.2 (see Demmel et al. [76]). Most hardware vendors' LAPACK libraries (such as Intel's MKL 10.1) have not yet incorporated this update. Thus, we had to use a source build of LAPACK 3.2, but were free to use any BLAS implementation. (We

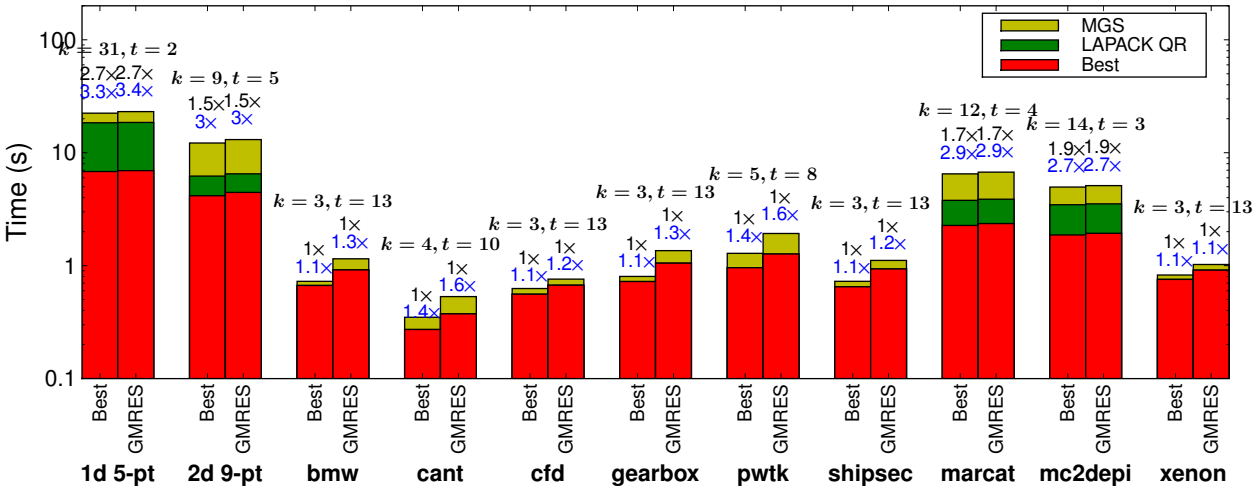


Figure 2.9: TSQR performance and GMRES modeled performance. Each bar is annotated with two speedup numbers: the one colored blue is w.r.t. MGS and the one colored black is w.r.t. LAPACK QR. 'Best' indicates the lowest time taken among LAPACK QR and TSQR—TSQR is the best when the black colored speedup numbers are more than 1.

also benchmarked LAPACK's QR factorization DGEQRF with the version of LAPACK in the MKL, but that did not exploit parallelism any more effectively than when we used a source build of LAPACK 3.2 with the MKL, so we did not use it.) For the in-place local QR factorization routine, we chose not to implement the BLAS 3 optimizations described in Demmel et al. [75] and inspired by the recursive QR factorizations of Elmroth and Gustavson [93], as we expected the expected number of columns to be too small to justify the additional floating-point operations entailed by these optimizations.

We implemented TSQR using the POSIX Threads (Pthreads) API with a SPMD-style algorithm. We used Pthreads rather than OpenMP in order to avoid harmful interactions between our parallelism and the OpenMP parallelism found in many BLAS implementations (such as Intel's MKL and Goto's BLAS). The computational routines were written in Fortran 2003, and drivers were written in C. The TSQR factorization and applying TSQR's Q factor to a matrix each only require two barriers, and for the problem sizes of interest, the barrier overhead did not contribute significantly to the runtime. Therefore, we used the Pthread barriers implementation, although it is known to be slow (see Nishtala and Yelick [180]).

Experiments

We compared the performance of TSQR against other QR factorizations, as they would be used in practice in CA-GMRES on the test matrices. We compared parallel TSQR against the following QR factorization methods: LAPACK's QR factorization `DGEQRF` as found in LAPACK 3.2, and MGS in both column-oriented (BLAS 1) and row-oriented (BLAS 2) forms. For the parallel implementations, we experimented to find the best number of processors. For TSQR, we also experimented to find the best total number of blocks. We not only benchmarked the factorization routines (`DGEQRF` in the case of LAPACK), but also the routines for assembling the explicit Q factor (`DORMQR` in the case of LAPACK), as that

is invoked in the CA-GMRES algorithm. We measured both the relative forward error $\|QR - A\|_1 / \|A\|_1$ and the orthogonality $\|Q^T Q - I\|_1 / \|A\|_1$, and found both to be within 100 of machine epsilon for all methods tested.

For our experiments, we built our benchmark with gfortran (version 4.3) and Goto's BLAS (version 1.26) [117]. Although this version of the BLAS uses OpenMP internally, we disabled this when calling the BLAS from TSQR, to avoid harmful interactions between the two levels of parallelism. We allowed the other factorization methods to use multiple threads in the BLAS, but using more than one thread resulted in no performance improvements. Thus, we only reported performance of the other orthogonalization methods on one processor.

Figure 2.9 gives the performance results. For each test matrix, we show the actual runtime of the orthogonalization process alone in the left bar ("TSQR"), and the modeled runtime of all of CA-GMRES on the right bar ("GMRES"). Above each bar, we show the optimal number of steps k in the matrix powers kernel for that matrix, as well as the number of outer iterations t chosen to make the total number of iterations before restart about 40. Each color in the bar represents the relative amount of time taken by that algorithm: red for TSQR, green for LAPACK QR, and yellow for MGS. The runtimes are on a log plot. Also above each bar, we show two speedup factors: on top, the TSQR speedup (if bigger than one) over LAPACK QR, and below that the speedup of the best algorithm (either LAPACK or TSQR) over MGS.

TSQR demonstrated speedups of up to 2.7× faster than LAPACK QR and up to 3.3× faster than MGS for large matrices, in particular when the matrix powers kernel output does not fit in cache. TSQR was designed to communicate the minimum amount between cache and main memory. However, for smaller problems, methods like MGS that perform fewer floating-point operations and have less setup overhead tend to perform better. This is entirely expected. In fact, CA-GMRES can use any QR factorization method that produces an R factor with a positive diagonal – which all the factorizations tested here support.

2.4 Block Gram-Schmidt

In this section, we introduce block Gram-Schmidt (BGS) algorithms. These are variants of existing Gram-Schmidt orthogonalization methods that work on blocks of columns at a time, instead of one column at a time as standard Gram-Schmidt methods do. If the number of columns in each block is s, BGS algorithms require a factor of $\Theta(s)$ fewer messages in parallel than their analogous nonblocked versions, and also requires a factor of $\Theta(s)$ fewer words transferred between slow and fast memory. The reason we discuss BGS in this thesis is that we will use these algorithms to avoid communication in our CA-Arnoldi and CA-GMRES Krylov subspace methods that we present in Chapter 3.

We begin our discussion of BGS by justifying blocked versions of Gram-Schmidt in Section 2.4.1, introducing notation in Section 2.4.2, and then describing two different approaches to BGS algorithms in Section 2.4.3. We call these two approaches "skeletons," because they leave important details unspecified, like how to orthogonalize individual vectors within the current block of vectors, and how to perform reorthogonalization if at all. The first skeleton we call "Block CGS," because it is analogous to Classical Gram-Schmidt (CGS). The other, which we call "Block MGS," is analogous to Modified Gram-Schmidt (MGS). After that,

we fill in the skeletons in Section 2.4.4 by describing Block CGS with TSQR, which is the orthogonalization scheme we use in our CA-GMRES algorithm. This scheme does not include reorthogonalization, because reorthogonalization is not necessary when solving linear systems with GMRES. We then include performance models and a discussion of data layout in Section 2.4.5.

The material that follows Section 2.4.5 includes discussion of numerical accuracy and reorthogonalization. In Section 2.4.6, we discuss the accuracy of unblocked versions of Gram-Schmidt in floating-point arithmetic. Previous authors observed that naïve reorthogonalization in the blocked case can destroy orthogonality of the resulting vectors; we explain this failure case in Section 2.4.7. We also summarize a recent approach that addresses this problem correctly but slowly, in that it recovers using standard nonblocked Gram-Schmidt reorthogonalization. In Section 2.4.8, we overcome this problem without sacrificing performance, with a novel BGS algorithm with reorthogonalization ("RR-TSQR-BGS") that uses a new rank-revealing version of TSQR ("RR-TSQR," see Section 2.3).

Blocked versions of Gram-Schmidt are not new; see e.g., [143, 230, 217]. However, combining them with TSQR (Section 2.3) is a new idea as far as we know. Furthermore, our algorithms differ from previous work in that we use TSQR, rather than ordinary MGS or CGS, to orthogonalize the current block of vectors internally.

2.4.1 Introduction

Orthogonalizing a vector against an orthogonal basis in order to add the vector to the basis is an important kernel in most Krylov subspace methods. Some iterative methods, such as Arnoldi and GMRES, explicitly orthogonalize each new basis vector against all the previous basis vectors. Others, such as symmetric Lanczos and CG, use short recurrences that produce basis vectors that are orthogonal in exact arithmetic. Even so, in many cases the basis vectors produced by such recurrences lose their orthogonality in finite-precision arithmetic. This is often unacceptable when solving eigenvalue problems, in which case explicit reorthogonalization is required.

Many Krylov subspace methods use a form of the modified Gram-Schmidt (MGS) orthogonalization procedure. The usual implementations of Arnoldi iteration and GMRES use MGS. Symmetric Lanczos uses MGS implicitly; the symmetry of the matrix A means that most of the dot products in MGS are zero in exact arithmetic, and the algorithm is structured to exploit these zeros.

We showed in Demmel et al. [75] that MGS communicates asymptotically more than the lower bound, both in parallel and between levels of the memory hierarchy. This is a problem for iterative methods such as Arnoldi and GMRES, that may spend a large fraction of their time orthogonalizing basis vectors. For other Krylov iterations, reducing the cost of reorthogonalization may also improve performance, or even make reorthogonalization feasible when it was not before. Whatever progress we make replacing SpMV with the matrix powers kernel, we also have to address the (re)orthogonalization phase in order to make our methods truly communication-avoiding.

2.4.2 Notation

The block Gram-Schmidt algorithms we discuss in this section share the following notation. This notation may differ slightly from that used in the iterative methods in this thesis, such as when we describe CA-Arnoldi in Section 3.3. We use a simpler notation here that conveys the algorithms, and later introduce more specific notation as needed in the context of particular iterative methods.

We begin with a finite collection of length n vectors $V = [v_1, v_2, \ldots, v_m]$, where $m \leq n$. In our applications, $m \ll n$. We then group these vectors into blocks $V = [V_1, V_2, \ldots, V_M]$ with $M \leq m$. There is no particular requirement that the blocks all have the same number of columns; in fact, Vanderstraeten [230] chooses the number of columns in each block dynamically, to improve numerical stability. Our iterative methods will have the option to exploit this fact, but we present them as if the number of columns in each block is the same (except perhaps for the first block, but we will neglect that difference in this section).

All the variants of Gram-Schmidt presented in this section have the goal of producing a set of vectors $Q = [q_1, q_2, \ldots, q_m]$ such that for $1 \leq j \leq m$, if the vectors $V = [v_1, \ldots, v_j]$ are linearly independent, then the vectors $q_1, \ldots, q_j$ form a unitary basis for $\text{span}\{v_1, \ldots, v_j\}$. The method should also be able to detect for a particular j, if $v_j \in \text{span}\{v_1, \ldots, v_{j-1}\}$. In that case, the method should stop. If the m input vectors are linearly independent, the final result should be a "thin" QR factorization $V = QR$ where $Q = [q_1, \ldots, q_m]$ and R is an $m \times m$ upper triangular matrix with positive real diagonal entries. The columns of Q are grouped into blocks $Q = [Q_1, Q_2, \ldots, Q_M]$, using the same grouping as $V = [V_1, V_2, \ldots, V_M]$. We write the corresponding blocks of R as R_{jk}, where in exact arithmetic, $R_{jk} = Q_j^* V_k$.

2.4.3 Algorithmic skeletons

We discuss two approaches to block Gram-Schmidt algorithms in this section. We call the two approaches "skeletons," because they leave important details unspecified, like how to orthogonalize individual vectors within the current block of vectors, and how to perform reorthogonalization if at all. The latter are like "muscles," in that they are are implementation choices to be filled in. Those choices affect both numerical stability and performance.

Algorithm 10 Block CGS skeleton

Input: $V = [V_1, V_2, \ldots, V_M]$, where V is $n \times m$. Each V_k is $n \times m_k$, with $\sum_{k=1}^{M} m_k = m$.

Output: $Q = [Q_1, \ldots, Q_M]$, where $\mathcal{R}(Q) = \mathcal{R}(V)$ and $\mathcal{R}([Q_1, \ldots, Q_k]) = \mathcal{R}([V_1, \ldots, V_k])$ for $k = 1, \ldots, M$. Each Q_k has m_k columns.

Output: R: $m \times m$ upper triangular matrix, laid out in blocks so that R_{ij} is $m_i \times m_j$. $V = QR$ is the thin QR factorization of V.

1: **for** $k = 1$ to M **do**
2: $R_{1:k-1,k} := [Q_1, \ldots, Q_{k-1}]^* V_k$
3: $Y_k := V_k - [Q_1, \ldots, Q_{k-1}] R_{1:k-1,k}$
4: Orthogonalize Y_k internally (if possible – see below for explanation), producing a (thin) QR factorization $Q_k R_{kk} = Y_k$
5: **end for**

Algorithm 11 Block MGS skeleton

Input: $V = [V_1, V_2, \ldots, V_M]$, where V is $n \times m$. Each V_k is $n \times m_k$, with $\sum_{k=1}^{M} m_k = m$.
Output: $Q = [Q_1, \ldots, Q_M]$, where $\mathcal{R}(Q) = \mathcal{R}(V)$ and $\mathcal{R}([Q_1, \ldots, Q_k]) = \mathcal{R}([V_1, \ldots, V_k])$ for $k = 1, \ldots, M$. Each Q_k has m_k columns.
Output: R: $m \times m$ upper triangular matrix, laid out in blocks so that R_{ij} is $m_i \times m_j$. $V = QR$ is the thin QR factorization of V.

1: **for** $k = 1$ to M **do**
2: $\quad V_k^{(1)} := V_k$
3: $\quad$ **for** $j = 1$ to $k - 1$ **do**
4: $\quad\quad R_{jk} := Q_j^* V_k^{(j)}$
5: $\quad\quad V_k^{(j+1)} := V_k^{(j)} - Q_j R_{jk}$
6: $\quad$ **end for**
7: $\quad Y_k := V_k^{(k)}$
8: $\quad$ Orthogonalize Y internally (if possible – see below for explanation), producing a (thin) QR factorization $Q_k R_{kk} = Y_k$
9: **end for**

The first skeleton we call "Block Classical Gram-Schmidt" (Block CGS), and the second we call "Block Modified Gram-Schmidt" (Block MGS). The Block CGS skeleton is shown as Algorithm 10, and the Block MGS skeleton is shown as Algorithm 10. All the operations in both of these, except the final "orthogonalize the block internally" step, are performed using dense matrix-matrix multiplications. For example, the computation $R_{ij} := Q_j^* V_k$ in Block MGS (Algorithm 11) is a matrix-matrix multiplication, in which one expects Q_j and V_k to be "tall and skinny" (to have many more rows than columns). This expectation influences implementation details, like data layout. For example, in order to achieve the minimum number of block transfers between levels of the memory hierarchy, in which each block transfer is allowed to consist only of contiguous data, a specific data layout may be required. We discuss this further in Section 2.4.5.

The block parts of block Gram-Schmidt algorithms only perform matrix-matrix multiplications, in order to improve performance. The most "interesting" thing that could happen in the block phase, from a numerical analysis perspective, is a reorthogonalization pass. In contrast, most of the interest focuses on the "orthogonalize the current block column internally" operation. This could be called the *unblocked case*, by analogy with other dense one-sided factorizations in LAPACK, which have a main BLAS 3 algorithm that works on block columns (such as `DGETRF` for LU factorization), and a BLAS 2 algorithm that factors each block column (such as `DGETF2` for LU factorization). The choice of algorithm for the unblocked case has a large effect on numerical stability, which we will discuss in Sections 2.4.6 and 2.4.7. It also affects performance, which we will discuss in Section 2.4.5. This is true especially if the number of blocks M is small, which is typically the case for our iterative methods. The "if possible" mentioned for the unblocked case in Algorithms 10 and 10 refers to the possibility of breakdown, which we will discuss along with the loss of orthogonality in Section 2.4.6.

2.4.4 Block CGS with TSQR

In this section, we present Block CGS with TSQR as Algorithm 12. This is the algorithm we use in CA-GMRES in Chapter 3. Block CGS with TSQR does not include reorthogonalization. Since the standard version of GMRES (that uses nonblocked MGS orthogonalization) does not require reorthogonalization when solving linear systems (see Paige et al. [186]), we expect that it is required even less in CA-GMRES, where the use of TSQR improves orthogonality of the block columns. If reorthogonalization is necessary, it should not be applied naïvely to Block CGS with TSQR, as we explain in Section 2.4.7. While we only present Block CGS with TSQR here, it is not hard to derive Block MGS with TSQR analogously.

Algorithm 12 Block CGS with TSQR

Input: $V = [V_1, V_2, \ldots, V_M]$, where V is $n \times m$. Each V_k is $n \times m_k$, with $\sum_{k=1}^{M} m_k = m$.
Output: $Q = [Q_1, \ldots, Q_M]$, where $\mathcal{R}(Q) = \mathcal{R}(V)$ and $\mathcal{R}([Q_1, \ldots, Q_k]) = \mathcal{R}([V_1, \ldots, V_k])$ for $k = 1, \ldots, M$. Each Q_k has m_k columns.
Output: R: $m \times m$ upper triangular matrix, laid out in blocks so that R_{ij} is $m_i \times m_j$. $V = QR$ is the thin QR factorization of V.

1: **for** $k = 1$ to M **do**
2: $\quad R_{1:k-1,k} := [Q_1, \ldots, Q_{k-1}]^* V_k$
3: $\quad Y_k := V_k - [Q_1, \ldots, Q_{k-1}] R_{1:k-1,k}$
4: $\quad$ Compute the thin QR factorization $Y_k = Q_k R_{kk}$ via TSQR
5: **end for**

2.4.5 Performance models

In this section, we present simple performance models for Block CGS (Algorithm 10) and Block MGS (Algorithm 11). These neglect the unblocked case. We present performance models for the "tall skinny" unblocked case in Demmel et al. [75]. Our models use the "latency-bandwidth" parallel and sequential hardware models described in Section 1.2.2 of this thesis.

Developing the performance models requires discussion of data layout. As we mentioned in Section 1.2.2, our sequential hardware model assumes that data movement between levels of the memory hierarchy requires that the data be stored contiguously in memory. The whole point of developing and analyzing BGS is for our CA-Arnoldi (Section 3.3) CA-GMRES (Section 3.4) algorithms. These Krylov methods use BGS with TSQR (Algorithm 12) to orthogonalize their Krylov basis vectors. In order to avoid copying data between layouts (which would add both bandwidth and latency costs), CA-Arnoldi and CA-GMRES require the following block layout for the V_k and Q_k blocks. Each V_k dimension $n \times m_k$ is laid out in N blocks of rows, with the same number of rows n/N in each block (if possible):

$$V_k = \begin{pmatrix} V_{1k} \\ V_{2k} \\ \vdots \\ V_{Nk} \end{pmatrix}$$

Thus, each block V_{jk} is N/m_k by m_k. The corresponding orthogonalized block column Q_k should have the same block layout as V_k. The consequence of using TSQR is that three row blocks will be accessed at once. For best performance, those blocks should all fit in fast memory. Thus, in order to make the BGS row block dimensions consistent with TSQR, we size the row blocks so that up to three may reside in fast memory at once. TSQR also requires that each block have at least as many rows as it has columns. Thus, $W \geq 3 \max_k m_k^2$ is required. If N is the number of row blocks in a block column, then we need $3n \max_k m_k/N$ words in fast memory to hold three row blocks, and thus $3n \max_k m_k/N \leq W$ is required. Fewer row blocks means lower latency costs, so the best number of row blocks is the maximum given that constraint:

$$N = \max \left\{ \frac{3n \max_k m_k}{W}, 1 \right\}.$$

The second term in the max is if W is large enough to hold three entire block columns, in which case the block columns do not need to be divided into row blocks. We neglect this case for now. The result is that the best number of row blocks N_{best} in sequential BGS is given by

$$N_{\text{best}} = \frac{3n \max_k m_k}{W}. \tag{2.16}$$

For Block MGS, it is obvious that $N = 3n \max_k m_k/W$ is optimal or nearly so, since sequential Block MGS works on two blocks (V_{ik} and Q_{ij}) in fast memory at any one time. For Block CGS, this is not obvious, since the matrix-matrix multiplications involve several blocks $[Q_1, \ldots, Q_{k-1}$ at once. Perhaps some other layout might result in optimal communication. However, for Block CGS we require the same layout as for Block MGS. We do so because TSQR (and the matrix powers kernel, in CA-Arnoldi and CA-GMRES) requires that layout. Copying data between different layouts to optimize each of Block CGS and TSQR separately might make each faster separately, but make the total run time slower. It would also make the implementation more complicated, so we do not do it.

Performance models

We develop parallel and sequential performance models for Block CGS and Block MGS below. As we observe in [75] and [79], there are many different implementation strategies for TSQR (and thus for BGS), depending on the arrangement of processors and the memory hierarchies available. In [79], we present and implement one possible arrangement, which for a single-node shared-memory parallel computer minimizes the number of data transfers between each processor's private cache and DRAM. It also ensures that each processor only accesses row blocks of the V_k and Q_j block columns that reside on DRAM pages to which it has (NUMA) affinity, and maximizes parallelism. In this section, rather than try to model that arrangement, we assume two simplified hardware models. One is a sequential processor with two levels of memory hierarchy and a fast memory of capacity W floating-point words, and the second parallel model is a collection of P processors (in which we do not consider memory hierarchies). For details on the performance models used in this thesis, see Section 1.2.2. These performance models do not account for reorthogonalization, which we discuss starting in Section 2.4.6.

Algorithm	# flops	# messages	# words
Block CGS	$2nm^2$	$\frac{6nm^2}{Ws}$	$\frac{3nm^2}{s}$
Block MGS	$2nm^2$	$\frac{6nm^2}{Ws}$	$\frac{3nm^2}{s}$
Standard MGS	$2nm^2$	$\frac{3nm}{2} + \frac{n^2m^2}{2W-m(m+1)}$	$\frac{2nm^2}{2W-m(m+1)}$

Table 2.3: Highest-order terms in the sequential performance models for Block CGS, Block MGS, and standard MGS. W is the fast memory capacity in words.

Algorithm	# flops	# messages	# words
Block CGS	$\frac{2nm^2}{P} + \frac{m^2}{2}\log P$	$\frac{m}{s}\log P$	$\frac{m^2}{2}\log P$
Block MGS	$\frac{2nm^2}{P} + \frac{m^2}{2}\log P$	$\frac{m^2}{s^2}\log P$	$\frac{m^2}{2}\log P$
Standard MGS	$\frac{2nm^2}{P}$	$2m\log P$	$\frac{m^2}{2}\log P$

Table 2.4: Highest-order terms in the parallel performance models for Block CGS, Block MGS, and standard MGS. P is the number of processors.

In the models below, we assume for simplicity that the block widths m_k of the blocks V_1, $V_2, \ldots, V_M$ are all the same: $m_k = s$ for $k = 1, 2, \ldots, M$. We also assume the number of rows n/N in each block is the same. In the parallel model, $N = P$ (the number of processors).

Sequential model

The above argument about BGS needing the same row block dimensions as TSQR means that Block MGS and Block CGS have the same sequential performance model. They merely perform the computations in a different order. Thus, it suffices to analyze Block MGS. We do so now. Table 2.3 shows the resulting performance models. The standard MGS performance model in the table comes from Demmel et al. [75]. We skip the derivation here, since it is not hard to do. The total number of floating-point operations is $2nm^2$, the number of words transferred from slow memory to fast memory and back is $6nsM(M-1) = 3nm^2/s$, and the total number of messages is $4N = 6nm^2/(Ws)$.

Parallel model

In the sequential model above, the data layout requirements meant that Block CGS and Block MGS had the same performance model, since the row blocks in both algorithms must have the same layout and dimensions. In the parallel model below, Block CGS and Block MGS will differ, since messages between processors need not be contiguous in the memory of one processor.

We begin with Block MGS. As in the analysis above, the fundamental operations of Block MGS are $R_{jk} := Q_j^* V_k$ and $V_k := V_k - Q_j R_{ij}$. The first of these, $R_{jk} := Q_j^* V_k$, is a parallel matrix-matrix multiply, and would be invoked in the following way:

1: Each processor i ($i = 1, \ldots, P$) computes the $s \times s$ matrix $R_{jk}^{(i)} := Q_{ij}^* V_{ik}$ in parallel
2: In $\log P$ messages, using a binary reduction tree, compute $\sum_{i=1}^{P} R_{jk}^{(i)}$

The first step $R_{jk}^{(i)} := Q_{ij}^* V_{ik}$ is an n/P by s times s by n/P matrix-matrix multiply, requiring

$2ns^2/P$ floating-point operations. The second step requires $s^2 \log P$ floating-point operations (along the critical path), $\log P$ messages, and $s^2 \log P$ total words transferred. Both of these steps are executed $M(M-1)/2$ times.

The second operation, $V_k := V_k - Q_j R_{ij}$, would proceed as follows:

1: Each processor i ($i = 1, \ldots, P$) computes the $n/P \times s$ matrix $V_{ik} := V_{ij} - Q_{ij} R_{jk}$ in parallel

This involves no communication (not counting any synchronization at the end). The $V_{ij} := V_{ij} - Q_{ij} R_{ij}$ operation is an n/P by s times $s \times s$ matrix-matrix multiply and an $n/P \times s$ matrix sum, costing a total of $2ns^2/P + O(ns/P)$ floating-point operations. This step is executed $M(M-1)/2$ times.

The total number of floating-point operations for Block MGS is therefore $2nm^2/P + (m^2/2) \log P$, the total number of words sent and received is $(m^2/2) \log P$, and the total number of messages is $(m^2/s^2) \log P$.

The Block CGS analysis is analogous, so we skip it. Parallel Block CGS requires the same number of floating-point operations and total words sent and received as Block MGS, but Block CGS requires only $(m/s) \log P$ messages.

2.4.6 Accuracy in the unblocked case

From this point on in our presentation of BGS, we leave the discussion of performance and focus on accuracy in finite-precision arithmetic and reorthogonalization. All of the Gram-Schmidt algorithms discussed in this thesis are equivalent in exact arithmetic, as long as none of the input vectors are linearly dependent. However, the accuracy of the algorithms differ in finite-precision arithmetic. In this section, we first explain what we mean by accuracy. We then briefly summarize known results about nonblocked variants of Gram-Schmidt and other orthogonalization methods. In later sections, we will explain how block Gram-Schmidt differs. In particular, a simple example in Section 2.4.7 shows how a naïve version of BGS can produce vectors that are not at all orthogonal.

The "unblocked case" refers to the operation that makes V_k internally orthogonal, after it has been made orthogonal to previous groups of orthogonal basis vectors $Q_1, \ldots, Q_{k-1}$. This amounts to a QR factorization of a tall and skinny matrix. Our technical report [75] refers to many different choices for this operation: Classical Gram-Schmidt (CGS), Modified Gram-Schmidt (MGS) in either column-oriented (BLAS 1) or row-oriented (BLAS 2) forms, CholeskyQR, Householder QR, or TSQR. The report gives performance models and summarizes the expected loss of orthogonality for each, assuming no reorthogonalization. In this section, we discuss both reorthogonalization, and tests that show whether it is necessary.

The previous usual approach for the unblocked case is the column-oriented form of MGS, which may or may not include reorthogonalization. This compares to the blocked LU and QR matrix factorizations in LAPACK and ScaLAPACK, for which the unblocked case uses the analogous textbook column-by-column version of the factorization. In [75], we show that using TSQR for the unblocked case, rather than the column-by-column algorithms, reduces the amount of communication asymptotically, without sacrificing numerical stability. This does not apply solely to QR; in [123], for example, our collaborators Grigori et al. present an analogous algorithm for LU factorization [123].

What does accuracy mean?

Stewart [217] gives a concise summary and bibliography of numerical stability concerns when performing unblocked and blocked Gram-Schmidt orthogonalization in finite-precision arithmetic. If we think of Gram-Schmidt as computing the QR factorization $V = QR$, where V is a matrix whose columns are the input vectors to orthogonalize, then there are at least two ways of defining the accuracy of the orthogonalization:

- The residual error $\|V - Q \cdot R\|_2$
- Orthogonality of the columns of Q

Stewart explains that almost all orthogonalization methods of interest guarantee a small residual error under certain conditions (such as V being of full numerical rank). However, fewer algorithms are able to guarantee that the vectors they produce are orthogonal to machine precision. One measure for the loss of orthogonality is the following norm:

$$\text{Orthogonality}(Q) = \|I - Q^*Q\|_2. \tag{2.17}$$

In exact arithmetic, for all Gram-Schmidt schemes, the above is zero as long as the input vectors are linearly independent. In finite-precision computations, however, this norm many grow larger and larger, at least until the supposedly orthogonal vectors become linearly dependent. Schemes for improving the stability of orthogonalization schemes thus focus on bounding the loss of orthogonality. Note that Equation (2.17) is feasible to compute in the usual case of interest, where Q has many more rows than columns (so that the matrix $I - Q^*Q$ has small dimensions).

Accuracy in unblocked Gram-Schmidt

In Demmel et al. [75, Table 11], we summarized the accuracy of various orthogonalization schemes. The analysis quotes works by Smoktunowicz et al. [211] and Giraud et al. [112], among others (which those works in turn cite). The various orthogonalization schemes we summarized in [75] include Householder QR, TSQR, unblocked MGS and CGS (with and without reorthogonalization), and an algorithm we called "CholeskyQR" (which entails computing the Cholesky factorization $V^*V = LL^*$ and computing $Q := VL^{-*}$). We repeat the summary in this section as Table 2.5. Note that Householder QR and TSQR produce Q factors that are *unconditionally orthogonal*, that is, that are orthogonal whether or not the matrix V is numerically full rank. We do not make that distinction in Table 2.5 here, because in an s-step Krylov method, the s-step basis should always be numerically full rank. Otherwise orthogonalizing the basis may produce vectors not in the desired Krylov subspace.

Reorthogonalization criteria

In (nonblocked) CGS and MGS, reorthogonalization is not always necessary. One may choose to do it anyway, to be safe and also to avoid the expense or trouble of checking; this is called *full reorthogonalization*. To avoid unnecessary reorthogonalization, one may perform *selective reorthogonalization*. There, one tests each vector before and after orthogonalizing it against

Algorithm	$\|I - Q^T Q\|_2$ upper bound	Reference(s)
Householder QR	$O(\varepsilon)$	[114]
TSQR	$O(\varepsilon)$	[75]
CGS2 or MGS2	$O(\varepsilon)$	[1, 150]
MGS	$O(\varepsilon\kappa(A))$	[32]
CholeskyQR	$O(\varepsilon\kappa(A)^2)$	[216]
CGS	$O(\varepsilon\kappa(A)^{m-1})$	[150, 211]

Table 2.5: Upper bounds on deviation from orthogonality of the Q factor from various QR factorization algorithms of the $n \times m$ matrix V (with $n \geq m$). Machine precision is ε. CGS2 means CGS with one full reorthogonalization pass and MGS2 means MGS with one full reorthogonalization pass. CholeskyQR means computing the Cholesky factorization $V^*V = LL^*$ and computing $Q := VL^{-*}$.

the previously orthogonalized vectors. If the test fails, one reorthogonalizes it against all the previously orthogonalized vectors. A typical criterion involves comparing the norm of the vector before and after the orthogonalization pass; if it drops by more than a certain factor K (say, $K = 1/10$), then the result may be mostly noise and thus reorthogonalization is required. For a review and evaluation of various reorthogonalization criteria, see Giraud et al. [111]. There are also reorthogonalization tests specific to Krylov subspace methods like symmetric Lanczos; Bai et al. [16] give an overview.

We do not evaluate the effectiveness (in terms of accuracy of the orthogonalization) of different reorthogonalization criteria in this thesis. However, the criterion matters to us because they way it is computed may affect performance. For example, many criteria require the norm of each vector before the first orthogonalization pass. Computing this is not already part of most orthogonalization schemes (except for Householder QR, which does not require reorthogonalization), so it may require additional messages in parallel. We will discuss this further below, where we c

What about "Twice is Enough"?

One reorthogonalization pass is a standard approach in unblocked orthogonalization methods. It is based on the "Twice is Enough" observation by Kahan (cited in Parlett [189, Section 6.9]). This says that when orthogonalizing a basis of a Krylov subspace, if the current vector v_k is orthogonalized twice to produce q_k, only two possibilities remain:

- q_k is orthogonal to $q_1, \ldots, q_{k-1}$, or
- an invariant subspace has been reached, so that $v_k \in \text{span}\{q_1, \ldots, q_{k-1}\}$. This is the "lucky breakdown" case that we describe later.

The latter can be detected if the norm of v_k drops significantly as a result of reorthogonalization (but before normalization). Note that "Twice is Enough" applies only in the context of Krylov subspace methods, where linear dependence of the vectors has a useful diagnostic meaning. For a general collection of vectors, some of which may be zero or otherwise dependent on the other vectors, other techniques are required to ensure that the computed vectors are orthogonal (see e.g., Stewart [217]).

2.4.7 Naïve block reorthogonalization may fail

It turns out that in block versions of Gram-Schmidt, the block operations in either the Block CGS or Block MGS forms matter little for accuracy of the resulting vectors. For instance, Stewart [217] explains that simply performing Block CGS twice, as follows:

1: $R_{1:k-1,k} := [Q_1, \ldots, Q_{k-1}]^* V_k$
2: $Y_k := V_k - [Q_1, \ldots, Q_{k-1}] R_{1:k-1,k}$
3: $R'_{1:k-1,k} := [Q_1, \ldots, Q_{k-1}]^* Y_k$
4: $Z_k := Y_k - [Q_1, \ldots, Q_{k-1}] R'_{1:k-1,k}$
5: $R_{1:k-1,k} := R_{1:k-1,k} + R'_{1:k-1,k}$

can cover for inaccuracies in the orthogonality of the previously orthogonalized blocks Q_1, $\ldots$, Q_{k-1}. What affects accuracy more is how we handle the unblocked case – that is, how we orthogonalize the result Z_k of the above process.

The obvious thing to do with Z_k is to orthogonalize it with (nonblocked) MGS, and then repeat the MGS step if necessary. Jalby and Philippe [143], in their analysis of blocked algorithms, present two variants of Block MGS. They differ only in their handling of the unblocked case. The first variant, which they call "Block Gram-Schmidt" (BGS), just uses MGS on V_k, after a round of block orthogonalizations. In the worst case, the loss of orthogonality in this variant could be as bad as (unblocked) CGS. Thus, the authors propose the B2GS algorithm, in which the current block V_k gets one MGS reorthogonalization pass.

Stewart [217] points out a way that Jalby and Philippe's B2GS algorithm can fail. Imagine that as a result of the block orthogonalization phase, the columns of V_k are made orthogonal to machine precision against the columns of Q_1, $\ldots$, Q_{k-1}. Suppose then that we apply unblocked MGS to the columns of V_k. If the norm of a column of V_k decreases significantly during this process, then the corresponding column may no longer be orthogonal to the columns of $[Q_1, \ldots, Q_{k-1}]$. Reorthogonalizing the resulting Q_k internally, even if done in exact arithmetic, could then result in a Q_k which is not orthogonal to $[Q_1, \ldots, Q_{k-1}]$. Also, reapplying the block reorthogonalization of the resulting Q_k against Q_1, $\ldots$, Q_{k-1} could mess up the "correct" columns of Q_k by mixing them with the "bad" column(s). For a simple example, see Appendix C.2.

Stewart proposes a fix to the above problem, which we paraphrase in our terms as Algorithm 13. We refer the interested reader to Stewart's recent paper [217].

Dynamically sized blocks

Vanderstraeten [230], citing Jalby and Philippe, also recognizes that BGS can be unstable. Rather than reorthogonalization, he takes a different approach: he chooses the size of each block dynamically, to bound an estimate of the condition number of that block. We cannot easily use this approach in our iterative methods, because there, the number of columns in a block is determined by the matrix powers kernel (Section 2.1) before that block is orthogonalized. We would have to split up the block into subblocks. Furthermore, Vanderstraeten's condition number estimator requires column norms, which would require more communication than we want to spend.

Algorithm 13 Block Gram-Schmidt with reorthogonalization (Stewart [217])

Input: $V = [V_1, V_2, \ldots, V_M]$, where V is $n \times m$. Each V_k is $n \times m_k$, with $\sum_{k=1}^{M} m_k = m$.
Output: $Q = [Q_1, \ldots, Q_M]$, where $\mathcal{R}(Q) = \mathcal{R}(V)$ and $\mathcal{R}([Q_1, \ldots, Q_k]) = \mathcal{R}([V_1, \ldots, V_k])$ for $k = 1, \ldots, M$. Each Q_k has r_k columns with $0 \leq r_k \leq m_k$.

1: **for** $k = 1$ to M **do**
2: Apply one pass of block orthogonalization (in either Block CGS or Block MGS form) to V_k, resulting in Y_k.
3: Apply one pass of CGS with reorthogonalization to Y_k, resulting in Z_k.
4: **if** the norm of each column of Z_k is no smaller than half the norm of the corresponding column of Y_k **then**
5: The orthogonalization of V_k is acceptable.
6: **else**
7: Apply one pass of block orthogonalization (in either Block CGS or Block MGS form) to Z_k, resulting in W_k.
8: Perform CGS with reorthogonalization to each column of W_k in turn.
9: **if** the norm of any column of W_k drops by more than half as a result **then**
10: Declare an "orthogonalization fault."
11: Perform CGS with reorthogonalization against all the columns of $Q_1, \ldots, Q_{k-1}$ as well as all the previously orthogonalized columns of W_k. The result is Q_k.
12: **end if**
13: **end if**
14: **end for**

2.4.8 Rank-revealing TSQR and BGS

The example in Section 2.4.7 shows that even if a block Y_k given by

$$Y_k := (I - [Q_1, \ldots, Q_{k-1}][Q_1, \ldots, Q_{k-1}]^*)V_k$$

has been made orthogonal to machine precision against all the previously orthogonalized blocks $Q_1, \ldots, Q_{k-1}$, any orthogonalization procedure on the columns of the block Y_k, no matter how accurate, may magnify tiny error terms from the block orthogonalization phase. This may negate the effect of the block orthogonalization, so that if $Y_k = Q_k R_{kk}$ is the thin QR factorization of Y_k, the columns of Q_k may no longer be orthogonal (to machine precision) to $\mathcal{R}([Q_1, \ldots, Q_{k-1}])$.

The example above also suggests a solution. Since the problem comes from Y_k having nearly linearly dependent columns, a rank-revealing factorization may be used to detect and remove such columns. Da Cunha et al. [67] observed that one can construct a rank-revealing factorization algorithm for tall skinny matrices, using any fast and accurate QR factorization algorithm (such as the one they propose, which as we discuss in Demmel et al. [75] is a precursor to our TSQR). One first computes the thin QR factorization using the fast and accurate algorithm, and then applies any suitable rank-revealing decomposition to the resulting small R factor. We call "RR-TSQR" the resulting rank-revealing decomposition, when the fast QR factorization algorithm used in TSQR.

RR-TSQR begins by computing the thin QR factorization of Y_k using TSQR. (This works because, as we mentioned in [75], TSQR produces a Q factor which is internally orthogonal to machine precision, even if the input matrix is rank deficient. This is the *unconditional orthogonality* property that TSQR shares with Householder QR.) Then, we use a rank-revealing decomposition on the much smaller R factor that results. Since the R factor is small, factoring it requires little or no communication (compared to computing the QR factorization of Y_k). Finally, since R is small and triangular, it is inexpensive to estimate its condition number using a 1-norm condition number estimator, such as `DTRCON` in LAPACK [5]. If R is sufficiently well-conditioned, no further work is required, since Y_k must also be well-conditioned. Otherwise, the columns of Y_k must be nearly dependent, so we can use any suitable rank-revealing factorization on R (such as the SVD, or QR with column pivoting) to identify and remove the redundant columns of Y_k.

We now describe in detail how to use RR-TSQR to construct a robust version of Block Gram-Schmidt. Suppose that the current $n \times m_k$ block of columns is V_k. First, we compute the norm of each column of V_k, using a single reduction operation. Then, we orthogonalize V_k against the previous blocks $Q_1, \ldots, Q_{k-1}$, resulting in Y_k. This may be done either in "Block MGS" form, or "Block CGS" form, as we discussed in Section 2.4.4. After this step, we need to check whether or not block reorthogonalization is necessary. There are two different ways to do this, depending on how often we expect block reorthogonalization to be needed:

1. If we expect to need frequent reorthogonalization, we may compute the column norms of Y_k in the same way as we computed the column norms of V_k, using a single reduction operation.

2. If we do not expect to reorthogonalize often, we may perform two tasks in one by computing the QR factorization (via TSQR) $Y_k = Q_{Y_k} R_{Y_k}$. Since TSQR computes the columns of Q_{Y_k} orthogonal to machine precision, the 2-norm of column j of R_{Y_k} gives (to within machine precision) the 2-norm of column j of Y_k: that is, for $j = 1, 2, \ldots, m_k$, $\|Y_k(:, j)\|_2 = \|R_{Y_k}(:, 1 : j)\|_2$.

For those columns of Y_k for which reorthogonalization is necessary (e.g., because their column norms drop too much – see Section 2.4.6 for a brief discussion of reorthogonalization schemes) we perform a second block orthogonalization pass. We only need to reorthogonalize those columns, and restricting the reorthogonalization to those columns will save us both bandwidth and flops (though not messages). However, there is no harm done in reorthogonalizing all the columns of Y_k. We omit this optimization from our explanation for simplicity's sake. We call the result of this second orthogonalization pass Z_k.

If we did not need to perform reorthogonalization, we already have the thin QR factorization $Y_k = Q_{Y_k} R_{Y_k}$, and we set $Z_k := Y_k$, $Q_{Z_k} := Q_{Y_k}$, and $R_{Z_k} := R_{Y_k}$. Otherwise, we compute the thin QR factorization $Z_k = Q_{Z_k} R_{Z_k}$ via TSQR. (If reorthogonalization is sufficiently infrequent, the occasional redundant TSQR will avoid the computation of the norms of all columns of Y_k for each k.)

We then use LAPACK's `DTRCON` condition number estimator to estimate the condition number of the upper triangular matrix R_{Z_k}. If it is not sufficiently well-conditioned, then

we must perform the rank-revealing part of RR-TSQR. Let

$$R_{Z_k} = U_k \Sigma_k W_k = U_k \begin{pmatrix} \Sigma_k^{(1)} & 0 \\ 0 & \Sigma_k^{(2)} \end{pmatrix} \begin{pmatrix} W_k^{(1)} \\ W_k^{(2)} \end{pmatrix}. \tag{2.18}$$

be the SVD decomposition of R_{Z_k}. In Equation (2.18), $W_k^{(1)}$ is $r_k \times m_k$, $W_k^{(2)}$ is $r_k \times m_k$, and Σ_k is $m_k \times m_k$. Furthermore, the positive diagonal matrix $\Sigma_k^{(1)}$ is $r_k \times r_k$, the nonnegative diagonal matrix $\Sigma_k^{(1)}$ is $m_k - r_k$ by $m_k - r_k$, the smallest element of $\Sigma_k^{(1)}$ is larger than the largest element of $\Sigma_k^{(2)}$, and the largest element of $\Sigma_k^{(2)}$ is smaller than some prespecified tolerance ϵ. Thus, we are justified in calling r_k the *numerical rank* of the matrix V_k', relative to some predetermined tolerance ϵ.

If $r_k = m_k$, we do not need the results of the SVD decomposition. In that case, we may take

$$\begin{aligned} Q_k &:= Q_{Z_k}, \text{ and} \\ R_{kk} &:= R_{Z_k}, \end{aligned}$$

and continue with the next block V_{k+1}. Otherwise, we must use the results of the SVD. Let $\tilde{Q}_k = Q_{Z_k} U_k$. This $n \times m_k$ matrix is also orthogonal, since Q_{Z_k} and U_k both are orthogonal. Partition $\tilde{Q}_k$ by columns as $\tilde{Q}_k = [\tilde{Q}_k^{(1)}, \tilde{Q}_k^{(2)}]$, where $\tilde{Q}_k^{(1)}$ is $n \times r_k$. Now, we can write

$$Z_k = \left(\tilde{Q}_k^{(1)} \Sigma_k^{(1)} W_k^{(1)}, \quad \tilde{Q}_k^{(2)} \Sigma_k^{(2)} W_k^{(2)} \right).$$

Since each column of the block $\tilde{Q}_k^{(2)} \Sigma_k^{(2)} W_k^{(2)}$ is smaller than the given tolerance in norm, we may set this block to zero without affecting the numerical rank of Z_k. This discards the "noise" terms. If we further compute the QR factorization $\Sigma_k^{(1)} W_k^{(1)} = \tilde{W}_k \tilde{R}_k$ (where $\tilde{W}_k$ is the $r_k \times r_k$ Q factor, and $\tilde{R}_k$ is the $r_k \times r_k$ upper triangular R factor), we may finish the process:

$$\begin{aligned} Q_k &= \tilde{Q}_k^{(1)} \tilde{W}_k, \text{ and} \\ R_{kk} &= \tilde{R}_k. \end{aligned} \tag{2.19}$$

Note that after the elimination of the "noise" terms, the resulting Q factor Q_k only has r_k columns, not m_k columns. If we apply this inductively to all block columns of $V = [V_1, V_2, \ldots, V_M]$, each orthogonalized block Q_j has only r_j columns, and the final result $Q = [Q_1, Q_2, \ldots, Q_M]$ has only $\sum_j r_j$ columns. This also means that in order for the final results

$$\begin{aligned} Q &= [Q_1, \ldots, Q_M], \text{ and} \\ R &= \begin{pmatrix} R_{11} & R_{12} & \ldots & R_{1M} \\ 0 & R_{22} & \ldots & R_{2M} \\ \vdots & \ddots & \ddots & \vdots \\ & & & R_{MM} \end{pmatrix} \end{aligned} \tag{2.20}$$

to form a QR decomposition of

$$V = [V_1, \ldots, V_M],$$

each V_k with numerical rank less than m_k should be replaced with Q_k, and the corresponding R_{kj} blocks in R (with $1 \leq j \leq k-1$) should be replaced with zeros (as Q_k is orthogonal to $Q_1, \ldots, Q_{k-1}$).

Algorithm 14 shows the resulting RR-TSQR – based block Gram-Schmidt procedure, which we call "RR-TSQR-BGS." We show the block orthogonalization passes in Block CGS form for conciseness, though they may also be done in Block MGS form. We are justified in calling RR-TSQR-BGS "communication-avoiding," as the algorithm is more or less the same as CAQR in Demmel et al. [75].

RR-TSQR-BGS and future work

We leave the testing and implementation of RR-TSQR-BGS (Algorithm 14) to future work. This is justified in part by this thesis' emphasis on solving linear systems, rather than solving eigenvalue problems. Reorthogonalization is unnecessary in finite-precision arithmetic when solving $Ax = b$ with standard GMRES (see Paige et al. [186]), for example, but it is an important part of the Arnoldi and Lanczos iterations. We include the algorithm here because we plan to do more experiments with our communication-avoiding versions of Arnoldi and Lanczos in the future, whereupon we will likely need a communication-avoiding reorthogonalization procedure like Algorithm 14.

2.5 Block orthogonalization in the M inner product

In this section, we present the M-CholBGS algorithm for orthogonalization in an inner product $\langle u, v\rangle \equiv v^* M u$ defined by an SPD matrix M. This is a combination of two algorithms:

M-BGS A variant of Block Gram-Schmidt.

M-CholeskyQR A factorization analogous to QR, based on first computing a block inner product of two bases and then performing a Cholesky decomposition. This is used to orthogonalize each block column internally, after the M-BGS step.

The resulting M-CholBGS algorithm communicates no more than BGS with TSQR (Algorithm 12 in Section 2.4.4) up to a constant factor, though it is not necessarily as accurate.

As far as we know, M-CholBGS is a novel algorithm. Thomas [221] presents a version of Block Gram-Schmidt for the M inner product. The M-BGS part of his algorithm is the same as ours, but instead of using M-CholeskyQR to factor each block column after the BGS step, he uses a variant of Modified Gram-Schmidt called "PGSM" (Product Gram-Schmidt in the M-norm). PGSM assumes that a Cholesky factorization $M = B^* B$ of the SPD matrix M is available. This is not always feasible to compute when M is large and sparse, or when it is only available as an operator M^{-1}. Furthermore, PGSM has performance similar to Modified Gram-Schmidt, since it orthogonalizes one column at a time. M-CholeskyQR communicates a factor $\Theta(s)$ less both in parallel (number of messages) and sequentially (number of words transferred between slow and fast memory) than MGS.

The main reason we developed M-CholBGS is to simplify left-preconditioned CA-Lanczos (Algorithm 29 in Section 4.3.4), which requires orthogonalizing Krylov basis vectors in the M inner product. There, we are solving the generalized eigenvalue problem $Ax = \lambda M x$, where A and M are SPD, and M is only available as the operator M^{-1}. We did not have time to incorporate M-CholBGS into left-preconditioned CA-Lanczos, but we record it here as a novel algorithm for future use. M-CholBGS can also be modified in order to biorthogonalize

two sets of vectors, for example in communication-avoiding versions of nonsymmetric Lanczos (see Chapter 6) or communication-avoiding versions of the "Two-Sided Arnoldi" algorithm of Ruhe [205] for solving nonsymmetric eigenvalue problems.

2.5.1 Review: CholeskyQR

In this section, we review the CholeskyQR algorithm, for computing the QR factorization $V = QR$ of an $m \times n$ dense matrix V, where $m \geq n$. Generally we assume that $m \gg n$, i.e., that V is "tall and skinny." We first mentioned CholeskyQR in this thesis in Section 1.2.5, as an example of a communication-avoiding algorithm. It is discussed by Stathopoulos and Wu [216] as a preprocessing step for efficiently computing the SVD of a tall skinny matrix, although CholeskyQR comes from the far older idea of using the normal equations to solve linear least squares problems. We show it here as Algorithm 15.

In Demmel et al. [75], we discuss CholeskyQR along with other communication-avoiding QR factorization algorithms that we develop. There, we show that CholeskyQR requires the minimum amount of communication, both in parallel and sequentially. The parallel algorithm requires only $\log P$ messages, and the sequential algorithm only reads and writes the data $1 + o(1)$ times. This is the same as TSQR, and as we mentioned in Section 2.3, is asymptotically less than the parallel version of Householder QR (`PDGEQRF`) implemented in ScaLAPACK, and the sequential version of Householder QR (`DGEQRF`) implemented in LAPACK. For details, see Demmel et al. [75, Table 4].

Line 1 of Algorithm 15 refers to the "Gram[7] matrix" of the columns of V. In general, the *Gram matrix* of a collection of vectors $v_1, v_2, \ldots$ in an inner product space is the symmetric matrix of inner products of those vectors: $G_{ij} \equiv \langle v_j, v_i \rangle$. In Algorithm 15, the inner product used is the standard Euclidean inner product, but this need not be the case, as we will see in the M-CholeskyQR algorithm in the next section.

CholeskyQR is not the most accurate communication-optimal version of the "thin" QR factorization. In fact, it can produce a Q factor which is not at all orthogonal, as we discuss in Demmel et al. [75]. In contrast, TSQR always produces a Q factor as orthogonal as that produced by Householder QR, even if the matrix V is not numerically full rank. TSQR also communicates asymptotically no more than CholeskyQR. The reason why we discuss CholeskyQR is because TSQR only can orthogonalize vectors with respect to the usual Euclidean inner product. CholeskyQR may be adapted to orthogonalize vectors with respect to a more general inner product. We show below how to adapt CholeskyQR in this way.

2.5.2 M-CholeskyQR

The "more general inner product" relevant to this thesis is an inner product defined by an SPD matrix M: the *M inner product* $\langle u, v \rangle_M = v^* M u$. For example, suppose that we are solving the SPD eigenvalue problem $Ax = \lambda x$ with symmetric Lanczos (Algorithm 24 in Section 4.1). In Lanczos, the Krylov basis vectors q_k are orthonormal in exact arithmetic

[7] Jørgen Pedersen Gram (1850–1916), a Danish actuary and mathematician, worked on many theoretical and applied problems. He (re)invented the Gram-Schmidt process when solving least-squares problems [181].

with respect to the Euclidean inner product. Thus, in exact arithmetic, $\langle q_i, q_j \rangle = 1$ if $i = j$ and is zero if $i \neq j$. Now suppose that we are solving the generalized eigenvalue problem $Ax = \lambda Mx$ with the same SPD matrix A, and with the SPD matrix M. As mentioned in Section 4.3, we may think of this as solving the "preconditioned" eigenvalue problem $M^{-1}Ax = \lambda x$ with left preconditioner M. Then, in left-preconditioned Lanczos (Algorithm 26 in Section 4.3.1), the preconditioned basis vectors q_k are orthonormal in exact arithmetic with respect to the M inner product. We say then that they are *M-orthogonal*: $\langle q_i, q_j \rangle_M = 1$ (in exact arithmetic) if $i = j$ and is zero if $i \neq j$.

As we mentioned above, TSQR cannot be used to M-orthogonalize vectors. Rather, we do this using Algorithm 16, which we call "M-CholeskyQR." Line 1 computes a Gram matrix with respect to the M inner product.

2.5.3 M-CholBGS

M-CholeskyQR (Algorithm 16) is good for M-orthogonalizing a single block V. However, in many of the algorithms in this thesis, such as left-preconditioned CA-Lanczos, we want to M-orthogonalize a current block column V_k against previously M-orthogonalized block columns $Q_0, \ldots, Q_{k-1}$. Here is an example. Suppose that we have an $n \times s_{k-1}$ dense matrix Q_{k-1} with $n \gg s_{k-1}$, whose columns are orthonormal with respect to the M inner product: $Q^*MQ = I$. Suppose also that we have an $n \times s_k$ matrix V_k such that $[Q_{k-1}, V_k]$ has full rank and $n \gg s_{k-1} + s_k$. Let W_k be an $n \times s_k$ matrix satisfying $V_k = M^{-1}W_k$. We call the columns of V "preconditioned basis vectors" and the columns of W "unpreconditioned basis vectors." Our task is to produce an $n \times s_k$ matrix Q_k whose columns are M-orthogonal among themselves and M-orthogonal to the columns of Q_{k-1}, so that

$$[Q_{k-1}, Q_k]^* M [Q_{k-1}, Q_k] = I. \tag{2.21}$$

However, we do not have access to M here. We have instead an $n \times s_{k-1}$ matrix Z_{k-1} such that $Z_{k-1} = MQ_{k-1}$: the columns of Z_{k-1} are preconditioned, and the columns of Q_{k-1} are unpreconditioned. Similarly, $Z_k = MQ_k$. Thus, we may rewrite Equation (2.21) as

$$[Z_{k-1}, Z_k]^* [Q_{k-1}, Q_k] = I. \tag{2.22}$$

We begin solving this problem by computing the Gram matrix G_k of the columns of $[Q_{k-1}, V_k]$:

$$G_k \equiv \begin{pmatrix} I & G_{k,k-1}^* \\ G_{k,k-1} & G_{kk} \end{pmatrix} = \begin{pmatrix} Q_{k-1}^* \\ V_k^* \end{pmatrix} \begin{pmatrix} Z_{k-1} & W_k \end{pmatrix} = \begin{pmatrix} I & Q_{k-1}^* W_k \\ (Q_{K-1}^* W_k)^* & V_k^* W_k \end{pmatrix}. \tag{2.23}$$

In this case, we know that the Gram matrix is SPD, since the columns of $[Q_{k-1}, V_k]$ are linearly independent. Thus, we may compute the Cholesky factorization of G_k:

$$G_k = L_k L_k^* = \begin{pmatrix} I & 0 \\ L_{k,k-1} & L_{kk} \end{pmatrix} \begin{pmatrix} I & L_{k,k-1}^* \\ 0 & L_{kk}^* \end{pmatrix}, \tag{2.24}$$

. In Equation (2.24), $L_{k,k-1} = G_{k,k-1}$ and L_{kk} is the Cholesky factor of the *Schur complement* $G_{kk} - G_{k,k-1}G_{k,k-1}^*$. Furthermore,

$$L_k^{-*} = \begin{pmatrix} I & -L_{kk}^{-*} L_{k,k-1}^* \\ 0 & L_{kk}^* \end{pmatrix}. \tag{2.25}$$

The $s_{k-1}+s_k$ by $s_{k-1}+s_k$ upper triangular matrix L_k^{-*} corresponds to the R factor computed by M-CholeskyQR (Algorithm 15). The corresponding "Q" factor is $[Q_{k-1}, Q_k]$, where

$$Q_k = \left(V_k - Q_{k-1} L_{k,k-1}^*\right) L_{kk}^{-*}. \tag{2.26}$$

This is the result of multiplying $[Q_{k-1}, V_k]$ on the right by L_k^{-*}. Since $L_k^{-1} G_k L_k^{-*} = I$, $[Q_{k-1}, Q_k]$ is M-orthogonal.

We call the resulting Algorithm 17 "M-CholBGS," because it combines M-CholeskyQR with an update that, like Block Gram-Schmidt, spends most of its time in BLAS 3 operations like matrix-matrix multiply and triangular solve with multiple vectors. We show the algorithm here in an "update" form, where we only need to M-orthogonalize the current block column V_k. However, though it can be easily adapted into a form like that the Block Gram-Schmidt algorithms in Section 2.4, where an entire collection of block columns must be M-orthogonalized. The performance model for M-CholBGS in that case is the same as that of the corresponding algorithms of Section 2.4, up to constant factors.

Algorithm 14 RR-TSQR-BGS

Input: $V = [V_1, V_2, \ldots, V_M]$, where V is $n \times m$. Each V_k is $n \times m_k$, with $\sum_{k=1}^{M} m_k = m$.
Output: $Q = [Q_1, \ldots, Q_M]$, where $\mathcal{R}(Q) = \mathcal{R}(V)$ and $\mathcal{R}([Q_1, \ldots, Q_k]) = \mathcal{R}([V_1, \ldots, V_k])$ for $k = 1, \ldots, M$. Each Q_k has r_k columns with $0 \leq r_k \leq m_k$.
Output: R: $\sum_j r_j \times \sum_j r_j$ upper triangular matrix, with a block layout given by Equation (2.20). $V = QR$ is the thin QR factorization of V, if for each k where $r_k < m_k$, we replace V_k (which has m_k columns) with Q_k (which has r_k columns).

for $k = 1$ to M **do**
  Using a single reduction, compute 2-norm of each column of V_k, for example via $\text{Norms}_k := \sqrt{\sum_{i=1}^{n} (V_k(i,:))^2}$
  $R_{k,1:k-1} := [Q_1, \ldots, Q_k]^* V_k$
  $Y_k := V_k - [Q_1, \ldots, Q_k] R_{k,1:k-1}$ ▷ First block orthogonalization pass
  Compute TSQR factorization $Y_k = Q_{Y_k} R_{Y_k}$
  if for all $j = 1, \ldots, m_k$, $Y_k(:, j)$ does not need reorthogonalization **then**
    $Z_k := Y_k$, $Q_{Z_k} := Q_{Y_k}$, $R_{Z_k} := R_{Y_k}$
  else
    $R'_{k,1:k-1} := [Q_1, \ldots, Q_k]^* Y_k$
    $Z_k := Y_k - [Q_1, \ldots, Q_k] R'_{k,1:k-1}$ ▷ Block reorthogonalization
    $R_{k,1:k-1} := R_{k,1:k-1} + R'_{k,1:k-1}$
    Compute TSQR factorization $Z_k = Q_{Z_k} R_{Z_k}$
  end if
  Compute SVD of R_{Z_k} (Equation (2.18)), and the numerical rank r_k of R_{Z_k}
  if $r_k = m_k$ **then**
    $Q_k := Q_{Z_k}$, $R_{kk} := R_{Z_k}$
  else
    Compute Q_k and R_{kk} via Equation (2.19) ▷ Q_k has r_k columns and R_{kk} is $r_k \times r_k$
    for $j = 1$ to $k - 1$ **do**
      Replace $R_{k,j}$ with the $r_j \times r_k$ zero matrix
    end for
    Replace V_k with Q_k
  end if
end for

Algorithm 15 CholeskyQR

Input: $m \times n$ matrix V with $m \gg n$
Output: QR factorization $V = QR$, where Q is $m \times n$ and R is $n \times n$

$G := V^* V$ ▷ The "Gram matrix"
Compute the Cholesky factorization $R^* R = G$
$Q := V R^{-1}$

Algorithm 16 M-CholeskyQR

Input: The SPD matrix M defines an inner product
Input: $n \times s$ matrix V with full column rank
Input: $n \times s$ matrix W satisfying $W = MV$
Output: $n \times s$ matrix Q with $\mathcal{R}(V) = \mathcal{R}(Q)$ and $Q^*MQ = I$ (the columns are M-orthogonal)
Output: $s \times s$ nonsingular upper triangular matrix R such that $V = QR$
1: Compute $G := V^*W$ ▷ The "Gram matrix"
2: Compute $s \times s$ Cholesky factorization $R^*R = G$
3: Compute $Q := VR^{-1}$

Algorithm 17 M-CholBGS

Input: The SPD matrix M defines an inner product
Input: $n \times s_{k-1}$ matrix Q_{k-1} with $n \gg s_{k-1}$ and $Q_{k-1}^*MQ_{k-1} = I$
Input: $n \times s_k$ matrix V_k with $n \gg s_{k-1} + s_k$, such that $[Q_{k-1}, V_k]$ has full column rank
Input: $n \times s_k$ matrix W_k with $W_k = MV_k$
Output: $n \times s_k$ matrix Q_k such that $\mathcal{R}(Q_{k-1}) \oplus \mathcal{R}(Q_k) = \mathcal{R}(Q_{k-1}) \oplus \mathcal{R}(V_k)$ and $[Q_{k-1}, Q_k]^*M[Q_{k-1}, Q_k] = I$
Output: $s_k \times s_k$ nonsingular upper triangular matrix R_{kk} and $s_{k-1} \times s_k$ matrix $R_{k-1,k}$ such that $[Q_{k-1}, Q_k] \begin{pmatrix} I & R_{k-1,k} \\ 0 & R_{kk} \end{pmatrix} = [Q_{k-1}, V_k]$
1: Compute $G_{k-1,k} := Q_{k-1}^*W_k$ and $G_{kk} := V_k^*W_k$
2: Compute $s_k \times s_k$ Cholesky factorization $R_{kk}^*R_{kk} = G_{kk} - G_{k-1,k}^*G_{k-1,k}$
3: Compute $Q_k := (V_k - Q_{k-1}G_{k-1,k})\, R_{kk}^{-1}$
4: Compute $R_{k-1,k} := -R_{kk}^{-1}G_{k-1,k}$

Chapter 3

Communication-avoiding Arnoldi and GMRES

In this chapter, we develop communication-avoiding versions of Krylov subspace methods related to Arnoldi iteration (see Arnoldi [8] and Bai et al. [16]). Such methods include Arnoldi iteration itself for solving nonsymmetric eigenvalue problems, as well as GMRES, the "Generalized Minimum Residual" method of Saad and Schultz [209] for solving nonsymmetric linear systems. Our communication-avoiding algorithms, CA-Arnoldi and CA-GMRES, communicate a factor of $\Theta(s)$ fewer messages in parallel than their corresponding standard Krylov methods. They also read the sparse matrix and vectors a factor of $\Theta(s)$ fewer times, where s is the s-step basis length (see Chapter 7). They do so by means of communication-avoiding implementations of the kernels presented in Chapter 2: the matrix powers kernel (Sections 2.1 and 2.2), TSQR (Section 2.3), and BGS (Section 2.4). Our parallel shared-memory implementation of CA-GMRES achieves speedups of up to 4.3× over standard GMRES on an 8-core Intel Clovertown processor, and up to 4.1× on an 8-core Intel Nehalem processor.

CA-Arnoldi and CA-GMRES were inspired by previous work in s-step Krylov methods. However, they are the first to avoid communication both in the sparse matrix and the dense vector operators. They also offer improved numerical stability, due to

- better choices of s-step basis (see Section 7.3),
- the use of balancing or equilibration (as appropriate) to avoid bad scaling of the s-step basis vectors (see Section 7.5),
- the ability to choose the basis length s independently of the restart length (see Section 3.3), and
- the use of an more accurate orthogonalization method for the Krylov basis vectors (see Section 3.3), which is important when solving eigenvalue problems.

Furthermore, we are the first to show how to avoid communication when preconditioning (for CA-GMRES) and when solving generalized eigenvalue problems (for CA-Arnoldi).

We begin this chapter in Section 3.1 with a summary of standard Arnoldi iteration. The sections that follow introduce communication-avoiding versions of Arnoldi. First, we summarize the standard version of Arnoldi in Section 3.1. In Section 3.2, we show the

Arnoldi(s) algorithm, which can complete s steps of a single restart cycle with the same communication requirements as a single iteration of standard Arnoldi. Then, in Section 3.3, we present the CA-Arnoldi or "Arnoldi(s, t)" algorithm. It has similar performance benefits as Arnoldi(s), but does not need to restart after every s steps. In Section 3.4, we use CA-Arnoldi as a model to develop a communication-avoiding version of GMRES, which we call CA-GMRES. Finally, Sections 3.5 and 3.6 show numerical resp. performance experiments with CA-GMRES.

3.1 Arnoldi iteration

In this section, we summarize Arnoldi's iteration for solving sparse nonsymmetric eigenvalue problems [8, 16]. This summary will serve our development of communication-avoiding versions of Arnoldi iteration in Sections 3.2 and 3.3. We begin with Arnoldi because it offers a good "template" for the data dependencies between kernels in many different Krylov subspace methods, as we discussed in Section 1.1.3. Arnoldi closely resembles other KSMs for solving linear systems, such as the Generalized Minimum Residual (GMRES) method of Saad and Schultz [209] and the Full Orthogonalization Method (FOM) (see [208]). It also serves as an "inner loop" for more complex algorithms more commonly used in practice to solve eigenvalue problems, such as Implicitly Restarted Arnoldi (see e.g., Lehoucq et al. [164]). Furthermore, a whole class of iterative methods for symmetric[1] matrices, that include (symmetric) Lanczos iteration [161], the Method of Conjugate Gradients (CG) of Hestenes and Stiefel [131], and the Minimum Residual (MINRES) algorithm of Paige and Saunders [187], can be derived from Arnoldi iteration by assuming that the matrix A is symmetric. (We will discuss these classes of algorithms in Chapters 4 and 5.) The generality of Arnoldi iteration thus makes it a good template for deriving communication-avoiding Krylov subspace methods.

This section does *not* address algorithms such as nonsymmetric Lanczos iteration [161] or its relatives, which include the Method of Biconjugate Gradients (BiCG) of Fletcher [99], Conjugate Gradient Squared (CGS) of Sonneveld [214], the "Method of Biconjugate Gradients, Stabilized" (BiCGSTAB) of van der Vorst [226], or the Quasi-Minimal Residual (QMR) method of Freund and Nachtigal [103]. Their underlying iteration differs enough from Arnoldi's method that we cannot apply the approach described in this section directly.

3.1.1 Notation

We introduce some notation in this section to help us compare the usual version of Arnoldi iteration with our communication-avoiding versions. Arnoldi begins with an $n \times n$ matrix A and an $n \times 1$ vector v. If we run it for s steps and it does not break down, it constructs an $s + 1$ by s nonsingular upper Hessenberg matrix $\underline{H}$ such that

$$AQ = \underline{Q}\underline{H}, \tag{3.1}$$

[1] Here and elsewhere in this thesis, we label as "symmetric" both real symmetric matrices and complex Hermitian matrices. We write the algorithms assuming that all quantities are complex, for example, using A^* to indicate the adjoint of A. However, we use language suggesting real arithmetic (such as "symmetric" and "orthonormal") rather than language suggesting complex arithmetic (such as "Hermitian" and "unitary").

in which $\underline{Q}$ is an n by $s+1$ orthonormal matrix

$$\underline{Q} = [Q, q_{s+1}] = [q_1, \ldots, q_s, q_{s+1}]$$

(with $q_1 = v/\|v\|_2$) whose columns comprise a basis of the Krylov subspace

$$\mathcal{K}_{s+1}(A, v) = \text{span}\{v, Av, A^2v, \ldots, A^sv\}.$$

We do not indicate s when we write $\underline{Q}$ or $\underline{H}$, because s is usually understood from context. We write $h_{ij} = \underline{H}(i, j)$, and we write H for the principal $s \times s$ submatrix of $\underline{H}$, so that

$$\underline{H} = \begin{pmatrix} H \\ 0, \ldots, 0, h_{s+1,s} \end{pmatrix}.$$

Depending on the context, an underline under a letter representing matrix means either "add one more column to the right side" (e.g., $\underline{Q}$) or "add one more row to the bottom" (e.g., $\underline{H}$) of the matrix.

Algorithm 18 Arnoldi iteration

Input: $n \times n$ matrix A, and length n starting vector v

Output: Orthonormal n by $s+1$ matrix $\underline{Q} = [Q, q_{s+1}]$ whose columns are a basis for the Krylov subspace $\mathcal{K}_{s+1}(A, r)$, and a nonsingular $s+1$ by s upper Hessenberg matrix $\underline{H}$ such that $AQ = \underline{Q}\underline{H}$

1: $\beta := \|v\|_2$, $q_1 := v/\beta$
2: **for** $j = 1$ to s **do**
3: $\quad w_j := Aq_j$
4: $\quad$ **for** $i = 1$ to j **do** ▷ Use MGS to orthogonalize w_j against $q_1, \ldots, q_j$
5: $\quad\quad h_{ij} := \langle w, q_i \rangle$
6: $\quad\quad w_j := w_j - h_{ij} q_i$
7: $\quad$ **end for**
8: $\quad h_{j+1,j} := \|w_j\|_2$
9: $\quad$ **if** $h_{j+1,j} = 0$ **then**
10: $\quad\quad$ Exit due to lucky breakdown
11: $\quad$ **end if**
12: $\quad q_{j+1} := w_j / h_{j+1,j}$
13: **end for**

Algorithm 18 shows the usual formulation of Arnoldi iteration. It uses Modified Gram-Schmidt (MGS) to orthogonalize successive basis vectors. There are other forms of Arnoldi iteration, which use other orthogonalization methods. We discussed Arnoldi with Delayed Reorthogonalization (ADR) in Section 1.6.1, which uses Classical Gram-Schmidt (CGS) to reduce the number of synchronization points in a parallel implementation. We will discuss other forms of Arnoldi when we present the Arnoldi(s) algorithm in Section 3.2. However, MGS-based Arnoldi (Algorithm 18) is most often employed in practice, either by itself (usually with restarting – see Section 3.1.2) or as the inner loop of a method like Implicitly Restarted Arnoldi.

The key operations of Arnoldi iteration are the computation of the upper Hessenberg matrix $\underline{H}$ and the orthonormal basis vectors $q_1, \ldots, q_{s+1}$. (We use the letter "q," rather than the customary "v," because q suggests orthonormality by reminding one of the QR factorization.) The upper Hessenberg matrix H is the projection of the matrix A onto the subspace $\text{span}\{q_1, \ldots, q_s\}$. Various operations on $\underline{H}$ yield information for solving linear systems or eigenvalue problems involving A. For example, the Arnoldi method finds approximations for the eigenvalues of A using the eigenvalues of H (the *Ritz values*). GMRES solves $Ax = b$ approximately by starting with an initial guess x_0, performing Algorithm 18 with $r = b - Ax_0$, and then solving the least squares problem $\min_y \|\underline{H}y - \beta e_1\|_2$ to obtain coefficients y for the approximate solution x_s in terms of the basis vectors $q_1, \ldots, q_s$. FOM gets coefficients y for the approximate solution by solving the linear system $Hy = \beta e_1$.

Line 10 of Algorithm 18 refers to the possibility of Arnoldi breaking down. In exact arithmetic, Arnoldi only breaks down when the smallest j has been reached such that a degree $j - 1$ polynomial p_{j-1} exists with $p_{j-1}(A)v = 0$. The Krylov subspace basis cannot be made any larger, given the starting vector v. This is called a *lucky breakdown* for two reasons: when solving eigenvalue problems, the set of eigenvalues of H equals the set of eigenvalues of A, and when solving linear systems, the current approximate solution equals the exact solution. Lucky breakdowns are rarely encountered in practice, and so we do not consider them here.

3.1.2 Restarting

The Arnoldi process requires storing all the basis vectors and doing a full orthogonalization at each step. If s iterations are performed, then memory requirements scale as a multiple of s, and the computational expense scales quadratically with s. Letting s grow until the algorithm converges to the desired accuracy may therefore be impossible (insufficient memory) or impractical (too much computation). As a result, the Arnoldi process may be *restarted* by forgetting $\underline{H}$ and all basis vectors except q_{s+1}, making the last (or best, where "best" means the residual vector with the smallest norm) residual vector the new starting vector, and beginning the iteration over again. That would entail enclosing Algorithm 18 in an outer loop, a simple step which we do not show. We say then that the *restart length* is s, and that Algorithm 18 represents one *restart cycle*.

Restarting bounds the memory requirements by a multiple of s. However, it causes a loss of information stored in the discarded basis vectors and $\underline{H}$ matrix. This loss can adversely affect how fast the iteration converges, if the Arnoldi process is used to solve a linear system or an eigenvalue problem. Picking the right restart length involves a tradeoff between convergence rate and computational cost, and is constrained by memory capacity.

There are a number of techniques for retaining and exploiting some of this information after a restart; see e.g., [164, 72, 177]. These techniques all use several iterations of the standard Arnoldi process as an inner loop. As a result, we do not consider these techniques here, though speeding up the Arnoldi process can make them faster as well.

3.1.3 Avoiding communication in Arnoldi

In the sections that follow, we develop communication-avoiding versions of Arnoldi iteration. We begin in Section 3.2 by showing how to rearrange Arnoldi so as to complete s steps of a single restart cycle with the same communication requirements as a single step of Algorithm 18. We call the resulting algorithm Arnoldi(s). The basic idea is to use the matrix powers kernel (Section 2.1) to generate s basis vectors, and then orthogonalize them all at once using TSQR (Section 2.3). Then, in Section 3.3, we present the CA-Arnoldi or "Arnoldi(s, t)" algorithm, which has similar performance benefits, but does not need to restart after every s steps. CA-Arnoldi uses the matrix powers kernel to avoid communication in the sparse matrix operations, and a combination of TSQR and Block Gram-Schmidt (Section 2.4) to avoid communication in the dense vector operations. Finally, in Section 3.4, we use CA-Arnoldi as a model to develop a communication-avoiding version of GMRES, which we call CA-GMRES.

3.2 Arnoldi(s)

3.2.1 Ansatz

Walker [239] developed a version of GMRES which we consider a precursor to the Arnoldi(s) algorithm. The usual implementation of Arnoldi (Algorithm 18) generates the unitary basis vectors one at a time and the upper Hessenberg matrix one column at a time, using modified Gram-Schmidt orthogonalization. Walker wanted to use Householder QR instead to orthogonalize the basis vectors, since Householder QR produces more orthogonal vectors than MGS in finite-precision arithmetic (see Section 2.3.3).[2] However, Householder QR requires that all the vectors to orthogonalize be available at once. Algorithm 18 only generates one basis vector at a time, and cannot generate another until the current one has been orthogonalized against all the previous basis vectors. Walker dealt with this by first generating $s+1$ vectors which form a basis for the Krylov subspace, and then computing their QR factorization. From the resulting R factor and knowledge about the basis, he could reconstruct the upper Hessenberg matrix $\underline{H}$, and use that to solve the least squares problem underlying GMRES. This performed the work of s steps of GMRES. Then, he restarted GMRES and repeated the process until convergence. Walker did not use a matrix powers kernel to compute the basis vectors, nor did he have a communication-avoiding QR factorization. Bai et al. [19] based their Newton-basis GMRES on Walker's Householder GMRES, improving numerical stability by using a different s-step basis (see Chapter 7).

We summarize Walker's algorithm in our own, more general notation. Given a starting vector v, the algorithm first computes the vectors $v_1 = v$, $v_2 = Av$, $v_3 = A^2v$, $\ldots$, $v_{s+1} = A^sv$. We write

$$\underline{V} = [V, v_{s+1}] = [v_1, \ldots, v_s, v_{s+1}].$$

These vectors form a basis for the Krylov subspace $\mathcal{K}_{s+1}(A, r)$. More specifically, for $k =$

[2] This turns out to matter more for Arnoldi than for GMRES. The loss of orthogonality in finite-precision arithmetic due to using MGS does not adversely affect the accuracy of solving linear systems with GMRES (see Greenbaum et al. [121]). However, at the time of Walker's work this was not known.

$1, \ldots, s+1$,

$$\mathcal{R}\left(\underline{V}(:, 1:k)\right) = \text{span}\{v, Av, A^2v, \ldots, A^{k-1}v\}.$$

We call this particular basis the *monomial basis*, by analogy with polynomial interpolation. (The analogy will be clarified in Chapter 7.) In matrix notation, the $s+1$ by s *change of basis* matrix

$$\underline{B} = [e_2, e_3, \ldots, e_{s+1}] \tag{3.2}$$

satisfies

$$AV = \underline{VB}. \tag{3.3}$$

Note the similarity between Equation (3.3) and the Arnoldi relation (Equation (3.1)); we will use this below to derive the Arnoldi(s) algorithm. We denote by B the $s \times s$ principal submatrix of $\underline{B}$, so that

$$\underline{B} = \begin{pmatrix} B \\ 0, \ldots, 0, \underline{B}(s+1, s) \end{pmatrix}.$$

Let us assume for now that the QR factorization of $\underline{V}$ produces the same unitary vectors as would the usual Arnoldi process. We will address this issue in Section 3.2.4. We write the QR factorization of $\underline{V}$: $\underline{Q}\underline{R} = \underline{V}$ and $QR = V$. Here, we use the "thin" QR factorization in which $\underline{R}$ is $s+1$ by $s+1$ and R is the $s \times s$ principal submatrix of R. Then, we can use $\underline{R}$ and Equation (3.3) to reconstruct the Arnoldi relation (Equation (3.1)):

$$\begin{aligned} AQ &= \underline{Q}\underline{H}, \\ AQR &= \underline{Q}\underline{R}\underline{R}^{-1}\underline{H}R, \\ AV &= \underline{V}\underline{R}^{-1}\underline{H}R. \end{aligned}$$

This means that if $AV = \underline{VB}$ for some $s+1$ by s matrix $\underline{B}$, we have

$$\underline{H} = \underline{R}\underline{B}R^{-1}. \tag{3.4}$$

(For an analogous derivation in the case of Lanczos iteration, see the beginning of Meurant [176].) Naturally one can exploit the structures of $\underline{B}$, $\underline{R}$, and $\underline{R}^{-1}$ in order to simplify this expression. We will do this in Section 3.2.3.

3.2.2 A different basis

Many authors have observed that the monomial basis v, Av, A^2v, ..., A^sv becomes increasingly ill-conditioned in practice as s increases (see e.g., Chronopoulos and Gear [57], in the context of their s-step CG algorithm). In fact, the condition number of the matrix $\underline{V}$ whose columns are this basis grows exponentially in s (see Gautschi [110]). This is apparent because repeatedly applying a matrix to a vector is called the "power method," and under the right conditions it converges to the principal eigenvector of the matrix. Converging vectors obviously become more and more linearly dependent. This disadvantage of the monomial basis is also apparent by analogy with polynomial interpolation, where one generally avoids computations with the monomial basis. For a detailed discussion of these issues, see Chapter 7 (in particular Section 7.4).

Other authors recommend changing Walker's "Householder GMRES" to use a basis other than the monomial one. This gives us a different change of basis matrix $\underline{B}$ than the one shown in Equation (3.2). However, it still satisfies Equation (3.3), so that the derivation of the previous section and the formula $\underline{H} = \underline{RB}R^{-1}$ (Equation (3.4)) still hold. De Sturler [71] suggested a GMRES algorithm using Chebyshev polynomials as the basis. We discuss the advantages of Chebyshev polynomials in Section 7.3.3. Chebyshev polynomials are computed using a three-term recurrence, so the cost of computing the basis would still be dominated by the matrix operations. Joubert and Carey [144, 145] used a Chebyshev basis in their implementation of GMRES. The authors use a kind of "matrix powers kernel" which is a special case of ours (see Section 2.1) for regular 2-D meshes. Joubert and Carey's algorithm avoids communication in the orthogonalization phase by using an algorithm we call "CholeskyQR" (see Section 2.5). Joubert and Carey recognize that the Cholesky factorization in CholeskyQR may break down due to ill conditioning, so they compute a singular value decomposition of the Gram matrix instead of Cholesky. This does not break down in case the Gram matrix is numerically singular. They suggest switching to the standard version of GMRES if $\underline{V}$ has low numerical rank over several restart cycles and the algorithm therefore fails to make sufficient progress. The Chebyshev basis requires estimates of the field of values of the matrix, which they obtain from the Hessenberg matrix from previous restart cycles, in the same way that the Arnoldi eigenvalue solver would.

Our work in this chapter extends Joubert and Carey's algorithm in many respects. First, the authors do not address preconditioning in their GMRES algorithm, nor do they in their matrix powers kernel. We have done so in both cases. Also, our Arnoldi and GMRES algorithms use an accurate as well as communication-avoiding QR factorization. TSQR is unconditionally stable, which means that the iterative methods will always make progress as long as the computed basis is numerically full rank. Joubert and Carey use CholeskyQR, which requires factoring a matrix whose condition number is the square of the condition number of the basis. Finally, our iterative methods, unlike theirs, do not need to restart after each invocation of the matrix powers kernel. This increases the convergence rate, as well as the likelihood of success on very difficult problems.

Other authors used different bases to create similar versions of GMRES. Bai et al. created a Newton basis variant of GMRES, in which the basis was named by analogy with the basis used in Newton interpolation of polynomials.

$$\begin{aligned} v_1 &= v \\ v_2 &= (A - \lambda_1 I)v \\ v_3 &= (A - \lambda_2 I)(A - \lambda_1 I)v \\ &\vdots \\ v_{s+1} &= \prod_{i=1}^{s} (A - \lambda_i I)v \end{aligned}$$

Here the λ_i are shifts chosen to improve the condition number of the resulting basis. (See Section 7.3.2 for details on the Newton basis and a justification of its use.) Bai et al. compute the shifts from eigenvalue information gathered from previous restart cycles. The

change-of-basis matrix $\underline{B}$ for the Newton basis is

$$\underline{B} = \begin{pmatrix} \lambda_1 & & & & & \\ 1 & \lambda_2 & & & & \\ & \ddots & \ddots & & & \\ & & & \lambda_{s-1} & & \\ & & & 1 & \lambda_s & \\ & & & & 1 & \end{pmatrix}. \tag{3.5}$$

The Newton basis is slightly cheaper to compute than the Chebyshev basis, since it requires operating on only two vectors in fast memory at once, rather than three. Also, the Newton basis' condition number may be less sensitive to the quality of the eigenvalue estimates used to construct it. As a result, we use the Newton basis in our numerical experiments.

Bai et al. recognized that using a QR factorization other than modified Gram-Schmidt could improve the performance of the Arnoldi process, but they did not come up with new QR or matrix powers kernels. Erhel [94] used the same Newton-basis GMRES algorithm of Bai and his collaborators, but her implementation uses a new parallel QR factorization, called RODDEC, that communicates minimally for P parallel processors on a ring network. It requires $\Theta(P)$ messages, which is asymptotically more than TSQR requires (for parallel processors on a more general network).

3.2.3 The Arnoldi(s) algorithm

Algorithm 19 shows our Arnoldi(s) algorithm. It restarts after every s steps. In Line 6 of the algorithm, we can exploit the structure of $\underline{B}$ and $\underline{R}$ in order to reduce the cost of computing $\underline{H}$. If we break down $\underline{R}$ into

$$\underline{R} = \begin{pmatrix} R & z \\ 0_{s,s} & \rho \end{pmatrix}$$

where R is $s \times s$, z is $s \times 1$, and ρ is a scalar, and break down $\underline{B}$ into

$$\underline{B} = \begin{pmatrix} B \\ 0, \ldots, 0, b \end{pmatrix}$$

where B is $s \times s$ and $b = \underline{B}(s+1, s)$, then

$$\begin{aligned} \underline{H} = \underline{RB}R^{-1} &= \begin{pmatrix} R & z \\ 0_{s,s} & \rho \end{pmatrix} \begin{pmatrix} B \\ 0, \ldots, 0, b \end{pmatrix} R^{-1} \\ &= \begin{pmatrix} RBR^{-1} + b \cdot ze_s^T R^{-1} \\ \rho b \cdot e_s^T R^{-1} \end{pmatrix}. \end{aligned}$$

If we further let $\tilde{\rho} = R(s, s)$, then $e_s^T R^{-1} = \tilde{\rho}^{-1}$ and therefore

$$\underline{H} = \begin{pmatrix} RBR^{-1} + \tilde{\rho}^{-1} b \cdot ze_s^T \\ \tilde{\rho}^{-1} \rho b \cdot e_s^T \end{pmatrix} \equiv \begin{pmatrix} H \\ 0, \ldots, 0, h \end{pmatrix}. \tag{3.6}$$

This formula can be used to compute $\underline{H}$ in Line 6 of Algorithm 19.

Algorithm 19 Arnoldi(s)

Input: $n \times n$ matrix A and length n starting vector v

Output: An orthonormal n by $s+1$ matrix $\underline{Q} = [Q, q_{s+1}]$ whose columns are a basis for the Krylov subspace $\mathcal{K}_{s+1}(A, r)$, and a nonsingular $s+1$ by s upper Hessenberg matrix $\underline{H}$ such that $AQ = \underline{Q}\underline{H}$

1: $\beta := \|v\|_2$, $q_1 := v/\beta$
2: **while** we want to restart **do**
3: Fix the $s+1$ by s basis matrix $\underline{B}$
4: Compute $\underline{V}$ via matrix powers kernel
5: Compute the QR factorization $\underline{V} = \underline{Q}\underline{R}$ via TSQR
6: $\underline{H} := \underline{R}\underline{B}R^{-1}$ (Equation (3.6))
7: **if** we want to restart **then**
8: Set $v := q_{s+1}$ (the last column of $\underline{Q}$) and $\beta := 1$
9: Optionally change $\underline{B}$ based on eigenvalue approximations gathered from $\underline{H}$
10: **end if**
11: **end while**

3.2.4 Unitary scaling of the basis vectors

In Section 3.2.1, our development of the Arnoldi(s) algorithm depended on the QR factorization producing the same orthogonalized vectors (in exact arithmetic) as modified Gram-Schmidt in the standard Arnoldi process. Another way to say this is that the QR factorization should produce an R factor with nonnegative real diagonal entries. We show in Demmel et al. [76] how to modify the usual Householder QR factorization in a numerically stable way so that it produces an R factor with nonnegative real diagonal entries. The resulting sequential QR factorization can then be used as the local QR factorization in our TSQR algorithm, so that TSQR also produces an R factor with nonnegative diagonal entries.

Suppose, however, that we are forced to use a QR factorization which may produce an R factor with negative diagonal entries. We will show now that this has no effect on the approximate eigenvalues and eigenvectors computed by Arnoldi(s). However, if we use Arnoldi(s) instead like GMRES to solve linear systems, a little more computation and communication is necessary to compute the right approximate solution.

Suppose that the relation produced by the MGS Arnoldi process, applied to the starting vector v, is

$$A\hat{Q} = \underline{\hat{Q}}\underline{\hat{H}},$$

in which

$$\underline{\hat{Q}} = [\hat{Q}, \hat{q}_{s+1}] = [\hat{q}_1, \ldots, \hat{q}_s, \hat{q}_{s+1}]$$

is the n by $s+1$ matrix whose columns are an orthogonal basis for the Krylov subspace $\text{span}\{v, Av, A^2v, \ldots, A^sv\}$. Now suppose instead that we construct

$$\underline{V} = [V, v_{s+1}] = [v_1, v_2, \ldots, v_s, v_{s+1}]$$

with $v_k = A^{k-1}v$, and compute the QR factorization $\underline{V} = \underline{Q}\underline{R}$. The unitary matrices $\underline{\hat{Q}}$ and $\underline{Q}$ may differ by a unitary scaling, because any unitary scaling of a set of orthonormal

vectors results in a set of orthonormal vectors with the same span. The particular unitary scaling depends on the orthogonalization method. This is true in exact arithmetic as well as in finite-precision arithmetic; the phenomenon has nothing to do with rounding error. We write the resulting unitary scaling as

$$\underline{\Theta} = \begin{pmatrix} \Theta & 0_{s,1} \\ 0_{1,s} & \theta_{s+1} \end{pmatrix} = \text{diag}(\theta_1, \theta_2, \ldots, \theta_s, \theta_{s+1}),$$

with $|\theta_j| = 1$ for all j, where $\hat{\underline{Q}} = \underline{Q}\underline{\Theta}$. In that case we have a different QR factorization:

$$\hat{\underline{Q}}\hat{\underline{R}} = \underline{Q}\underline{\Theta}\underline{\Theta}^{-1}\underline{R}.$$

(Note that

$$\underline{\Theta}^{-1} = \underline{\Theta}^* = \text{diag}\left(\overline{\theta}_1, \ldots, \overline{\theta}_{s+1}\right),$$

since $\underline{\Theta}$ is unitary.) This also means that the $\hat{\underline{H}}$ resulting from the standard Arnoldi process may be different from $\underline{H} = \underline{R}\underline{B}R^{-1}$. In particular, we have:

$$\hat{\underline{H}} = \hat{\underline{Q}}^* A \hat{\underline{Q}} = \underline{\Theta}^* \underline{Q} A Q_s \Theta = \underline{\Theta}^* \underline{H} \Theta, \tag{3.7}$$

in which Θ is the $s \times s$ principal submatrix of $\underline{\Theta}$. Note that $\hat{H} = \Theta^* H \Theta$, which means that $\hat{H}$ and H have the same eigenvalues. This means that eigenvalue solvers based on Arnoldi(s) need not compute any of the θ_i in order to find the eigenvalues. This is especially helpful if we wish to use Ritz values to improve the condition number of the basis $\underline{V}$ in future restart cycles. Also, we will show in Section 3.4 that our version of GMRES based on Arnoldi(s) computes the same answer in exact arithmetic as standard GMRES, as long as $\theta_1 = 1$. That is, either the QR factorization used by Arnoldi(s) must not change the direction of the first column, or $\theta_1 = \langle v, q_1 \rangle$ must be computed via an inner product. The latter requires an additional communication step. Thus, it is advantageous to use a QR factorization that produces an R factor with nonnegative real diagonal entries.

3.2.5 Properties of basis conversion matrix

We now deduce that the structure of the change of basis matrix $\underline{B}$ must always be upper Hessenberg. For all the bases of interest, it is either tridiagonal or bidiagonal, so this argument is practically unnecessary. However, it is theoretically interesting to see how the Arnoldi(s) algorithm constrains the order of operations in the matrix powers kernel.

Using an analogous derivation to that from which Equation (3.4) came, we have

$$\underline{R}^{-1} = \begin{pmatrix} R^{-1} & -\rho^{-1} R^{-1} z \\ 0_{1,s} & \rho^{-1} \end{pmatrix},$$

and therefore

$$\begin{aligned} \underline{B} &= \underline{R}^{-1} \underline{H} R \\ &= \begin{pmatrix} R^{-1} & -\rho^{-1} R^{-1} z \\ 0_{1,s} & \rho^{-1} \end{pmatrix} \begin{pmatrix} H \\ 0, \ldots, 0, h \end{pmatrix} R \\ &= \begin{pmatrix} R^{-1} H R - \rho^{-1} \tilde{\rho} h \cdot R^{-1} z e_s^T \\ \rho^{-1} \tilde{\rho} h \cdot e_s^T \end{pmatrix}, \end{aligned}$$

in which $\tilde{\rho} = R(s, s)$.

The term $\rho^{-1}\tilde{\rho}h \cdot e_s^T$ in the last line of the above expression is the last row of the matrix $\underline{B}$. We see that all elements except the last of that row are zero. Furthermore, $R^{-1}HR$ is structurally upper Hessenberg, and $\rho^{-1}\tilde{\rho}hR^{-1}ze_s^T$ is an $s \times s$ matrix which is zero in all columns except the rightmost. Therefore, the matrix $\underline{B}$ must be upper Hessenberg, with $\underline{B}(s+1, s) = \rho^{-1}\tilde{\rho}h$.

Note that in the above, we neglect the row and column scalings represented by $\underline{\Theta}$ that differentiate the upper Hessenberg matrix $\underline{H}$ computed by Arnoldi(s) from the upper Hessenberg matrix $\underline{\hat{H}}$ computed by standard MGS Arnoldi iteration. We do this for simplicity, as the scalings do not affect the structure of $\underline{B}$.

The requirement that $\underline{B}$ be upper Hessenberg restricts the method used to construct the basis matrix $\underline{V}$. In particular, each basis vector v_{i+1} is some linear combination of $A^i v_1$ and $v_1, \ldots, v_i$:

$$v_{i+1} \in \text{span}\{v_1, \ldots, v_i\} \oplus \text{span}\{A^i v_1\}.$$

Applying induction, we see that

$$v_i \in \text{span}\{v_1, Av_1, A^2 v_1, \ldots, A^{i-1} v_1\}$$

for $i \in \{1, 2, \ldots, s+1\}$. This seems reasonable for a matrix powers kernel: each new vector in the basis is generated in turn from the previous basis vectors and the product of A and the current basis vectors. All the bases we consider in this work follow this pattern.

3.3 CA-Arnoldi or "Arnoldi(s, t)"

In this section, we we develop the Communication-Avoiding Arnoldi (CA-Arnoldi) algorithm. We also call it "Arnoldi(s, t)" to distinguish it from Arnoldi(s) (Algorithm 19 of Section 3.2). Arnoldi(s) must restart after each time it generates and orthogonalizes s basis vectors. In contrast, the s-step basis length s in CA-Arnoldi can be shorter than the restart length. The restart length is $s \cdot t$, where t refers to the number of "outer iterations" (times that the matrix powers kernel is invoked and its output vectors are made orthogonal to previous vectors and among themselves). Thus, Arnoldi(s) is the same as Arnoldi(s, $t = 1$).

We demonstrate in our numerical and performance experiments with CA-GMRES in Section 3.5 that being able to choose s shorter than the restart length has at least two advantages:

- s-step methods are not numerically stable if s is too large, but if the restart length r is too short, the Krylov method may converge slowly or not at all ("stagnation"). In that case, CA-Arnoldi can use a known stable value of s, while still choosing a restart length $r = s \cdot t$ for which Arnoldi is known to converge sufficiently fast.

- The best value of s for performance may be much shorter than typical restart lengths. We show in Section 3.5 for CA-GMRES that $s = 5$ may give the best performance, among all possible integer $s \geq 2$ with $s \cdot t = 60$.

Most previous s-step Arnoldi or GMRES algorithms had the same limitation as Arnoldi(s). This includes Walker's Householder GMRES [239] and the Newton-basis GMRES algorithm

of Bai et al. [17, 19]. The s-step Arnoldi method of Kim and Chronopoulos [154] did not have this limitation: it could choose s shorter than the restart length r. Our CA-Arnoldi improves on their work in the following ways.

- CA-Arnoldi can use any s-step basis (see Chapter 7 for definitions and examples), which improves numerical stability. Kim and Chronopoulos' algorithm can only use the monomial basis.
- While Kim and Chronopoulos recognize the value of improving locality and avoiding memory traffic, and point out that their algorithm exposes the potential for increased locality, their s-step Arnoldi does not asymptotically reduce the amount of memory traffic. The authors present no communication-avoiding matrix powers kernel algorithms, and they use a column-by-column orthogonalization procedure for the Arnoldi basis vectors, rather than TSQR (Section 2.3) and Block Gram-Schmidt (Section 2.4).
- The Krylov basis vectors in Kim and Chronopoulos' algorithm are only made orthogonal in a block sense: blocks V_i and V_j of s vectors each satisfy $V_j^* V_i = 0$ when $i \neq j$, but do not necessarily satisfy $V_i^* V_i = I_{s \times s}$. CA-Arnoldi also makes the columns of each block orthogonal, using a numerically stable combination of TSQR (Section 2.3) and BGS (Section 2.4). This also makes it easier for CA-Arnoldi to perform reorthogonalization, as is often done when solving eigenvalue problems.

Like Arnoldi(s), CA-Arnoldi communicates a factor of s less than standard Arnoldi with the same restart length $r = s \cdot t$. CA-Arnoldi requires the same storage as standard Arnoldi. It may require a lower-order term more floating-point arithmetic operations than standard Arnoldi, but this is due only to the matrix powers kernel (see Section 2.1 for details).

In this section, we will derive CA-Arnoldi (also called Arnoldi(s, t)) step-by-step from Arnoldi(st). Each step clarifies how we save arithmetic operations over a naïve implementation. We begin in Section 3.3.1 by outlining the derivation approach. Section 3.3.2 introduces notation. In Section 3.3.3, we show that each outer iteration of CA-Arnoldi amounts to updating a QR factorization of a tall skinny matrix, by adding s more columns to its right side. We avoid communication when computing this update by using Block Gram-Schmidt (see Section 2.4). Section 3.3.4 shows how we use the results of that QR factorization update to perform a similar update of the Arnoldi upper Hessenberg matrix. Section 3.3.5 combines the previous sections' derivations in order to describe the CA-Arnoldi algorithm (shown as Algorithm 22). Finally, we suggest applications of CA-Arnoldi for solving generalized eigenvalue problems in Section 3.3.6.

3.3.1 Ansatz

In this section, we show how one begins deriving CA-Arnoldi. We introduce notation here which we will define formally only later. Suppose that we perform s steps of Arnoldi, either standard Arnoldi or the s-step variant, using the $n \times n$ matrix A and the starting vector q_1. Assuming that the iteration does not break down, this results in an n by $s+1$ matrix of basis vectors $\underline{Q}_0$, with

$$\underline{Q}_0 = [Q_0, q_{s+1}] = [q_1, q_2, \ldots, q_s, q_{s+1}] .$$

Its columns are an orthogonal basis for the Krylov subspace span$\{q_1, Aq_1, \ldots, A^s q_1\}$. Arnoldi also produces an $s+1$ by s nonsingular upper Hessenberg matrix $\underline{H}_0$, with

$$\underline{H}_0 = \begin{pmatrix} H_0 \\ h_0 e_s^T \end{pmatrix}.$$

The upper Hessenberg matrix $\underline{H}_0$ satisfies

$$AQ_0 = \underline{Q}_0 \underline{H}_0 = Q_0 H_0 + h_0 \cdot q_{s+1} e_s^T. \tag{3.8}$$

In Section 3.2, we showed how to compute $\underline{Q}_0$ and $\underline{H}_0$ using Arnoldi(s) instead of standard Arnoldi. We assume, as in the previous section, that the QR factorization used in Arnoldi(s) preserves the unitary scaling of the starting vector. However we choose to compute $\underline{Q}_0$ and $\underline{H}_0$, Equation (3.8) still holds in exact arithmetic.[3]

If this were Arnoldi(s), we would have to restart now. Instead, we take the last basis vector q_{s+1}, and make it the input of the matrix powers kernel. We set $v_{s+1} := q_{s+1}$, and the matrix powers kernel produces an n by $s+1$ matrix of basis vectors $\underline{V}_1$ with

$$\underline{V}_1 = [q_{s+1} = v_{s+1}, v_{s+2}, \ldots, v_{2s+1}].$$

We write

$$\begin{aligned} V_1 &= [v_{s+1}, v_{s+2}, \ldots, v_{2s}], \\ \underline{V}_1 &= [V_1, v_{2s+1}], \\ \acute{V}_1 &= [v_{s+2}, \ldots, v_{2s}], \\ \underline{\acute{V}}_1 &= [\acute{V}_1, v_{2s+1}]. \end{aligned}$$

The underline (as before) indicates "one more at the end," and the acute accent (pointing up and to the right) indicates "one less at the beginning." The vector omitted "at the beginning" is q_{s+1}, which is already orthogonal to the previous basis vectors; we neither need nor want to change it anymore. The columns of $\underline{\acute{V}}_1$ have not yet been orthogonalized against the columns of $\underline{Q}_0$.

We set

$$\begin{aligned} \mathfrak{Q}_0 &= Q_0, \\ \underline{\mathfrak{Q}}_0 &= \underline{Q}_0. \end{aligned}$$

Orthogonalizing the columns of $\underline{\acute{V}}_1$ against the previous basis vectors (here the columns of $\underline{\mathfrak{Q}}_0$) is equivalent to updating a QR factorization by adding the s columns of $\underline{\acute{V}}_1$ to the right of $\underline{\mathfrak{Q}}_0$. This results in an efficiently computed QR factorization of $[\underline{\mathfrak{Q}}_0, \underline{\acute{V}}_1]$:

$$[\underline{\mathfrak{Q}}_0, \underline{\acute{V}}_1] = [\underline{\mathfrak{Q}}_0, \underline{\acute{Q}}_1] \cdot \begin{pmatrix} I_{s+1,s+1} & \underline{\acute{\mathfrak{R}}}_{0,1} \\ 0 & \underline{\acute{R}}_1 \end{pmatrix}. \tag{3.9}$$

We perform this update in two steps. First, we use a block Gram-Schmidt operation to orthogonalize the columns of $\underline{\acute{V}}_1$ against the columns of $\underline{\mathfrak{Q}}_0$:

[3] In practice, we may want to begin a run of CA-Arnoldi with s iterations of standard Arnoldi first in order to compute Ritz values, which are used in successive outer iterations to compute a good basis. This is why we allow the first outer iteration to be computed either by standard Arnoldi or by Arnoldi(s).

- $\underline{\acute{\mathfrak{R}}}_{0,1} := \underline{\mathfrak{Q}}_0^* \underline{\acute{V}}_1$
- $\underline{\acute{V}}_1' := \underline{\acute{V}}_1 - \underline{\mathfrak{Q}}_0 \cdot \underline{\acute{\mathfrak{R}}}_{0,1}$

Then, we compute the QR factorization of $\underline{\acute{V}}_1'$, which makes the new basis vectors mutually orthogonal as well as orthogonal to the previous basis vectors:

$$\underline{\acute{V}}_1' = \underline{\acute{Q}}_1 \underline{\acute{R}}_1.$$

Now that we have the desired QR factorization update and a new set of s orthogonalized Arnoldi basis vectors, we can update the upper Hessenberg matrix as well. Just as in Arnoldi(s), the basis vectors $\underline{V}_1$ satisfy $AV_1 = \underline{V}_1 \underline{B}_1$ for the nonsingular upper Hessenberg (at least) matrix $\underline{B}_1$, which is defined by the choice of basis. That means

$$A[\mathfrak{Q}_0, V_1] = [\mathfrak{Q}_0, \underline{V}_1]\underline{\mathfrak{B}}_1, \tag{3.10}$$

where the $2s+1$ by $2s$ matrix $\underline{\mathfrak{B}}_1$ satisfies

$$\underline{\mathfrak{B}}_1 = \begin{pmatrix} H_0 & 0_{s,s} \\ h_0 e_1 e_s^T & \underline{B}_1 \end{pmatrix} = \begin{pmatrix} \mathfrak{B}_1 \\ b_1 e_{2s}^T \end{pmatrix}. \tag{3.11}$$

Combining Equation (3.10) with the QR factorization update (Equation (3.9)) and the definition of $\underline{\mathfrak{B}}_1$ (Equation (3.11)), we get

$$A[\mathfrak{Q}_0, Q_1] \begin{pmatrix} I & \mathfrak{R}_{0,1} \\ 0 & R_1 \end{pmatrix} = [\mathfrak{Q}_0, \underline{V}_1] \begin{pmatrix} I & \underline{\mathfrak{R}}_{0,1} \\ 0 & \underline{R}_1 \end{pmatrix} \begin{pmatrix} H_0 & 0_{s,s} \\ h_0 e_1 e_s^T & \underline{B}_1 \end{pmatrix}, \tag{3.12}$$

where

$$\mathfrak{R}_{0,1} = \mathfrak{Q}_0^* V_1 = [e_1, \mathfrak{Q}_0^* \acute{V}_1],$$
$$\underline{\mathfrak{R}}_{0,1} = \mathfrak{Q}_0^* \underline{V}_1 = [e_1, \mathfrak{Q}_0^* \underline{\acute{V}}_1],$$

and R_1 and $\underline{R}_1$ are the R factors in the QR factorization of V_1 resp. $\underline{V}_1$. (Similarly, $R_1(1,1) = \underline{R}_1(1,1) = 1$.) This is just a rearrangement of the updated R factor in Equation (3.9) and involves no additional computation. We set $\mathfrak{H}_0 := H_0$ and $\underline{\mathfrak{H}}_0 := \underline{H}_0$ also.

Equation (3.12) lets us recover the $2s+1$ by $2s$ upper Hessenberg matrix $\underline{\mathfrak{H}}_1$ which is the same as would be computed by $2s$ steps of standard Arnoldi iteration. We have

$$A[\mathfrak{Q}_0, Q_1] = [\mathfrak{Q}_0, \underline{Q}_1]\underline{\mathfrak{H}}_1$$

and therefore

$$\underline{\mathfrak{H}}_1 = \begin{pmatrix} I & \underline{\mathfrak{R}}_{0,1} \\ 0 & \underline{R}_1 \end{pmatrix} \begin{pmatrix} \mathfrak{H}_0 & 0_{s,s} \\ h_0 e_1 e_s^T & \underline{B}_1 \end{pmatrix} \begin{pmatrix} I & \mathfrak{R}_{0,1} \\ 0 & R_1 \end{pmatrix}^{-1}. \tag{3.13}$$

We will show later how to simplify the formula for $\underline{\mathfrak{H}}_1$ in Equation (3.13), so that $\underline{\mathfrak{H}}_1$ can be computed as an update, without needing to change $\underline{\mathfrak{H}}_0$ part of the matrix.

Applying the update technique in this section $t-1$ times results in a total of t "outer iterations" of CA-Arnoldi. The Arnoldi relation that results is

$$\begin{aligned} A\mathfrak{Q}_{t-1} &= \mathfrak{Q}_{t-1}\mathfrak{H}_{t-1} + h_{t-1} q_{st+1} e_{st}^T \\ &= \underline{\mathfrak{Q}}_{t-1}\underline{\mathfrak{H}}_{t-1}, \end{aligned} \tag{3.14}$$

where the outputs of CA-Arnoldi are the n by $st+1$ matrix of basis vectors $\underline{\mathfrak{Q}}_{t-1}$, and the $st+1$ by st upper Hessenberg matrix $\underline{\mathfrak{H}}_{t-1}$. Here,

$$\begin{aligned}\mathfrak{Q}_{t-1} &= [Q_0, \ldots, Q_{t-1}], \\ \underline{\mathfrak{Q}}_{t-1} &= [Q_1, \ldots, Q_{t-2}, \underline{Q}_{t-1}] = [\mathfrak{Q}_{t-1}, q_{st+1}],\end{aligned}$$

and

$$\underline{\mathfrak{H}}_{t-1} = \begin{pmatrix} H_0 & H_{0,1} & \ldots & H_{0,t-1} \\ h_0 e_1 e_s^T & H_1 & \ddots & H_{1,t-1} \\ & h_1 e_1 e_s^T & \ddots & \vdots \\ & & \ddots & H_{t-1} \\ & & & h_{t-1} e_s^T \end{pmatrix} = \begin{pmatrix} \mathfrak{H}_{t-1} \\ h_{t-1} e_{st}^T \end{pmatrix}.$$

In the above, each $H_{i,j}$ is a general $s \times s$ matrix, H_k is $s \times s$ upper Hessenberg, and $h_k = \underline{H}_k(s+1, s)$. We will explain our notation in more detail in Section 3.3.2, and sketch out the algorithm in the subsections that follow.

3.3.2 Notation

In this subsection, we explain the notation used in our Arnoldi(s, t) algorithm. The admittedly complicated notation helps clarify the structure of the algorithm.

CA-Arnoldi has three levels of notation. For more outer levels, we use taller and more elaborate typefaces. For example, v_{sk+j} (lowercase, Roman typeface) is a single basis vector in outer iteration k, V_k (uppercase, Roman) an $n \times s$ matrix whose columns are s basis vectors in a group for outer iteration k, and $\mathfrak{Q}_k$ (uppercase, Black letter) a collection of $s(k+1)$ groups of orthogonal basis vectors at outer iteration k. For example,

$$V_k = [v_{sk+1}, v_{sk+2}, \ldots, v_{sk+s}]$$

is an $n \times s$ matrix whose columns are s basis vectors in the k^{th} such group. At the outermost level, the subscripts refer to the current outer iteration. For example,

$$\mathfrak{Q}_k = [Q_0, Q_1, \ldots, Q_{k-1}]$$

is an $n \times s(k+1)$ matrix comprised of k groups of basis vectors, each group containing s basis vectors.

3.3.3 QR factorization update

The CA-Arnoldi algorithm follows directly from the QR factorization update outlined in Section 3.3.1. The update produces the same results (in exact arithmetic) as would a full QR factorization of all the basis vectors, but saves work. The update also preserves the directions of previous basis vectors, which a full QR factorization might not do.

There are many choices for computing the QR factorization update. First of all, in outer iteration k we may choose to work with $\underline{\acute{V}}_k$ (the *overlapping approach*) or with $\underline{V}_k$ (the *nonoverlapping approach*). In either case, we may represent the previously orthogonalized

basis vectors explicitly (as columns of a matrix), or implicitly (using the TSQR representation of the Q factor as a tree of small Q factors). Also, we may choose to perform the block Gram-Schmidt operation either in a classical Gram-Schmidt or modified Gram-Schmidt way. Each of these three choices may affect performance, and the type of block Gram-Schmidt used may also affect the orthogonality of the computed basis vectors in finite-precision arithmetic. We discuss the effects of all three choices on performance and accuracy in this section.

The QR update task

Suppose we have performed $k-1$ outer iterations of CA-Arnoldi. This produces $sk+1$ orthonormal resp. unitary basis vectors $q_1, \ldots, q_{sk}, q_{sk+1}$. These vectors form the first $sk+1$ columns of $\underline{\mathfrak{Q}}_{k-1}$, which can be considered the Q factor of a QR factorization with the identity as the R factor. For now, we gloss over the representation of $\mathfrak{Q}_{k-1}$. Outer iteration k of CA-Arnoldi then sets $v_{sk+1} := q_{sk+1}$ and produces s more basis vectors $v_{sk+2}, \ldots, v_{s(k+1)+1}$ using the matrix powers kernel. We write

$$\begin{aligned}
\acute{V}_k &= [v_{sk+2}, \ldots, v_{s(k+1)}], \\
\underline{\acute{V}}_k &= [\acute{V}_k, v_{s(k+1)+1}], \\
V_k &= [v_{sk+1}, \acute{V}_k], \\
\underline{V}_k &= [v_{sk+1}, \underline{\acute{V}}_k].
\end{aligned}$$

Our goal is to orthogonalize the new basis vectors v_{sk+2}, v_{sk+3}, $\ldots$, $v_{s(k+1)+1}$ against the previous basis vectors q_1, q_2, $\ldots$, q_{sk+1}. We can express this as updating a QR factorization by adding columns to the right of a matrix. There are two different ways to formulate the update, depending on how one distinguishes "previous" vectors and "new" vectors:

1. *Nonoverlapping approach*: update a QR factorization by adding the columns of $\underline{V}_k$ to the right of $\mathfrak{Q}_{k-1}$
2. *Overlapping approach*: update a QR factorization by adding the columns of $\underline{\acute{V}}_k$ to the right of $\underline{\mathfrak{Q}}_{k-1}$

Both approaches compute the same QR factorization of $[\underline{\mathfrak{Q}}_{k-1}, \underline{\acute{V}}_k] = [\mathfrak{Q}_{k-1}, \underline{V}_k]$ in exact arithmetic. The reason for distinguishing the two approaches is that the first basis vector of the k^{th} outer iteration, v_{sk+1} (which is also q_{sk+1}), is also the last basis vector of the $(k-1)^{\text{th}}$ outer iteration. The vector v_{sk+1} has already been orthogonalized against the previous basis vectors. The nonoverlapping approach chooses to orthogonalize v_{sk+1} redundantly against the previous basis vectors, while the overlapping approach avoids this extra work. We will compare both approaches in the sections that follow.

Nonoverlapping approach The nonoverlapping approach to the QR update proceeds as follows. We begin with the previously computed matrix of sk orthogonal basis vectors $\mathfrak{Q}_{k-1}$, and then add the $s+1$ columns of $\underline{V}_k$ to the right of it to form

$$\underline{\mathfrak{V}}_k = [\mathfrak{Q}_{k-1}, \underline{V}_k].$$

Here, the first column $v_{sk+1} = q_{sk+1}$ of $\underline{V}_k$ is the last column of $\underline{\mathfrak{Q}}_k$. Then, we find the QR factorization of $\underline{\mathfrak{V}}_k$ via an update. We call this approach "nonoverlapping" because we think of q_{sk+1} as belonging to and being processed by outer iteration k, even though it was generated by outer iteration $k-1$ (or was the starting vector, if $k = 0$). This was the approach we implemented in [79].

The nonoverlapping update proceeds as follows:

1: $\mathfrak{R}_{k-1,k} := \mathfrak{Q}_{k-1}^* \underline{V}_k$ ▷ $s(k-1)$ by $s+1$ matrix
2: $\underline{V}'_k := \underline{V}_k - \mathfrak{Q}_{k-1}\mathfrak{R}_{k-1,k}$
3: Compute QR factorization $\underline{Q}_k \underline{R}_k = \underline{V}'_k$

The resulting new QR factorization $\underline{\mathfrak{Q}}_k \underline{\mathfrak{R}}_k = \underline{\mathfrak{V}}_k$ has the form

$$\underline{\mathfrak{Q}}_k = [\mathfrak{Q}_{k-1}, \underline{Q}_k],$$
$$\underline{\mathfrak{R}}_k = \begin{pmatrix} I_{s(k-1),s(k-1)} & \underline{\mathfrak{R}}_{k-1,k} \\ 0 \cdot e_1 e_{s(k-1)}^T & \underline{R}_k \end{pmatrix}.$$

Obtaining $\mathfrak{Q}_{k-1}$ from $\underline{\mathfrak{Q}}_{k-1}$ is a special case of *downdating* the factorization (removing the effects of a column). For a QR factorization, downdating the last column requires no additional work. In a traditional Householder QR factorization, it involves ignoring the Householder reflector corresponding to the last column, when applying the Q factor. In TSQR (see Section 2.3), it involves ignoring the Householder reflector corresponding to the last column, for all of the blocks comprising the factorization. Similarly, $\mathfrak{R}_{k-1}$ is just the upper left $s(k-1)$ by $s(k-1)$ submatrix of $\underline{\mathfrak{R}}_{k-1}$.

Overlapping approach The overlapping approach to the QR update proceeds as follows. We begin with the previously computed matrix of $sk+1$ orthogonal basis vectors $\underline{\mathfrak{Q}}_{k-1}$, and then add the s columns of $\underline{\acute{V}}_k$ to the right of it to form

$$\underline{\mathfrak{V}}_k = [\underline{\mathfrak{Q}}_{k-1}, \underline{\acute{V}}_k].$$

Here, the last column $q_{sk+1} = v_{sk+1}$ of $\underline{\mathfrak{Q}}_{k-1}$ is the first column of $\underline{V}_k$. Then, we find the QR factorization of $\underline{\mathfrak{V}}_k$ via an update. We call this approach "overlapping" because q_{sk+1} is processed by outer iteration $k-1$, even though we think of it as "belonging" to outer iteration k.

The overlapping update proceeds as follows:

1: $\underline{\acute{\mathfrak{R}}}_{k-1,k} := \underline{\mathfrak{Q}}_{k-1}^* \underline{\acute{V}}_k$
2: $\underline{\acute{V}}'_k := \underline{\acute{V}}_k - \underline{\mathfrak{Q}}_{k-1}\underline{\acute{\mathfrak{R}}}_{k-1,k}$
3: Compute QR factorization $\underline{\acute{V}}'_k = \underline{\acute{Q}}_k \underline{\acute{R}}_k$

The resulting new QR factorization $\underline{\mathfrak{Q}}_k \underline{\mathfrak{R}}_k = \underline{\mathfrak{V}}_k$ has the following form:

$$\underline{\mathfrak{Q}}_k = [\underline{\mathfrak{Q}}_{k-1}, \underline{\acute{Q}}_k],$$
$$\underline{\mathfrak{R}}_k = \begin{pmatrix} I_{s(k-1)+1,s(k-1)+1} & \underline{\acute{\mathfrak{R}}}_{k-1,k} \\ 0 \cdot e_1 e_{s(k-1)+1}^T & \underline{\acute{R}}_k \end{pmatrix}.$$

This approach requires no downdating.

Block CGS or block MGS

Both the overlapping and nonoverlapping approaches begin with a step of Block Gram-Schmidt (see Section 2.4), to orthogonalize the the "new" set of basis vectors against the previously orthogonalized basis vectors. This may be done for all $sk + 1$ resp. sk basis vectors at once, which we call the *Block Classical Gram-Schmidt* or Block CGS approach. Alternately, it may be done $k - 1$ times for each group of s previous basis vectors, which we call the *Block Modified Gram-Schmidt* or Block MGS approach. We discuss both approaches in general in Section 2.4.3 and show their performance models in Section 2.4.5. Here, we show them in the context of CA-Arnoldi. When explaining Block CGS and Block MGS in this section, we assume the overlapping update approach without loss of generality. It is easy to extend the arguments here to the nonoverlapping update approach.

Algorithm 20 Block CGS update in outer iteration k of Arnoldi(s,t)

Require: $k > 0$

Input: $\underline{\mathfrak{Q}}_{k-1}$: $sk + 1$ already orthogonalized Arnoldi basis vectors

Input: $\underline{\acute{V}}_k$: s more basis vectors, not orthogonalized, which are the output of the matrix powers kernel

1: $\underline{\acute{\mathfrak{R}}}_{k-1,k} := \underline{\mathfrak{Q}}_{k-1}^* \underline{\acute{V}}_k$

2: $\underline{\acute{V}}_k' := \underline{\acute{V}}_k - \underline{\mathfrak{Q}}_{k-1} \underline{\acute{\mathfrak{R}}}_{k-1,k}$

Output: $\underline{\acute{V}}_k'$: s basis vectors, orthogonalized against $\underline{\mathfrak{Q}}_{k-1}$

Output: $\underline{\acute{\mathfrak{R}}}_{k-1}$: $sk + 1$ by s matrix

Algorithm 21 Block MGS update in outer iteration k of Arnoldi(s,t)

Require: $k > 0$

Input: $\underline{\mathfrak{Q}}_{k-1}$: $sk + 1$ already orthogonalized Arnoldi basis vectors

Input: $\underline{\acute{V}}_k$: s more basis vectors, not orthogonalized, which are the output of the matrix powers kernel

1: $\underline{\acute{R}}_{0,k} := \underline{Q}_0^* \underline{\acute{V}}_k$

2: $\underline{\acute{V}}_k := \underline{\acute{V}}_k - \underline{Q}_0 \acute{R}_{0,k}$

3: **for** $j = 1$ to $k - 1$ **do**

4: $\quad \underline{\acute{R}}_{j,k} := \underline{\acute{Q}}_j^* \underline{\acute{V}}_k$

5: $\quad \underline{\acute{V}}_k := \underline{\acute{V}}_k - \underline{\acute{Q}}_j \underline{\acute{R}}_{j,k}$

6: **end for**

Output: $\underline{\acute{V}}_k'$: s basis vectors, orthogonalized against $\underline{\mathfrak{Q}}_{k-1}$

Output: Collection of k matrices $\underline{\acute{R}}_{0,k}$, ..., $\underline{\acute{R}}_{k-1,k}$. The first is $s + 1$ by s and the rest are $s \times s$.

We show Block CGS at outer iteration k of CA-Arnoldi as Algorithm 20. It requires only two steps. Block MGS at outer iteration k of CA-Arnoldi is shown as Algorithm 21. It

requires $2k$ steps. In Block MGS, we assemble $\underline{\acute{\mathfrak{R}}}_{k-1,k}$ via

$$\underline{\acute{\mathfrak{R}}}_{k-1,k} = \begin{pmatrix} \underline{\acute{R}}_{0,k} \\ \vdots \\ \underline{\acute{R}}_{k-1,k} \end{pmatrix}.$$

In exact arithmetic, both Block CGS and Block MGS compute the same updated basis vectors $\underline{\acute{V}}_k'$ and R factor component $\underline{\acute{\mathfrak{R}}}_{k-1,k}$. The differences are in performance and accuracy. Since each "previous" block $\underline{\acute{Q}}_j$ already has unitary columns, the accuracy of Block MGS is comparable to that of orthogonalizing a vector against k unit-length orthogonal vectors using standard MGS. Similarly, the accuracy of Block CGS is comparable to that of orthogonalizing a vector against k unit-length orthogonal vectors with standard CGS.

Explicit or implicit Q factor

The standard version of Arnoldi stores all of the entries of the orthogonalized basis vectors explicitly. However, the basis vectors make up the columns of a Q factor, and there are implicit ways to represent and work with a Q factor. For example, Householder QR represents the Q factor as a sequence of Householder reflectors, which are stored in the bottom triangle of a matrix. TSQR (Section 2.3) computes the Q factor in a tree form, whose nodes are themselves Q factors. Converting the resulting implicitly stored Q factor into an explicitly stored matrix is another kernel, which requires as much communication (and about as much arithmetic) as TSQR. Furthermore, the key operations in the CA-Arnoldi QR factorization update do not require knowing the entries of the Q factor. They only require applying the Q factor (or its (conjugate) transpose) to a matrix, and the TSQR implicit representation suffice to compute that. Nevertheless, there are advantages to computing the explicit version of the Q factor. When Q is stored as an explicit matrix, the Block Gram-Schmidt update steps can use matrix-matrix multiplication, which is typically more optimized than the local QR-like computations required to apply an implicitly stored Q factor. We chose to store the Q factor produced by TSQR explicitly, in our CA-GMRES (see Section 3.4) implementation discussed in Section 3.5. CA-GMRES is still faster than standard GMRES in our experiments, despite the extra communication and computation required to compute the explicit form of the Q factors.

Repartitioning the R factor

When showing how to compute the Arnoldi upper Hessenberg matrix from the R factor in the QR factorization update (Section 3.3.3), it simplifies the notation if we first "repartition" the block matrix $\underline{\mathfrak{R}}_k$. This is purely notational; repartitioning requires no computation and insignificant data movement.

To this end, we first define four variations on the block inner product:

$$\begin{aligned} \mathfrak{R}_{k-1,k} &= \mathfrak{Q}_{k-1}^* V_k, \\ \underline{\mathfrak{R}}_{k-1,k} &= \mathfrak{Q}_{k-1}^* \underline{V}_k, \\ \acute{\mathfrak{R}}_{k-1,k} &= \underline{\mathfrak{Q}}_{k-1}^* \acute{V}_k, \\ \underline{\acute{\mathfrak{R}}}_{k-1,k} &= \underline{\mathfrak{Q}}_{k-1}^* \underline{\acute{V}}_k. \end{aligned} \tag{3.15}$$

The notation is complicated but the pattern is this: the acute accent means $\acute{V}_k$ is involved, and the underline means $\underline{V}_k$ is involved. Both an acute accent and an underline means that $\underline{\acute{V}}_k$ is involved. Given this notation, we can partition $\mathfrak{R}_k$ and $\underline{\mathfrak{R}}_k$ in two different ways:

$$\begin{aligned}
\mathfrak{R}_k &= \begin{pmatrix} I_{s(k-1)+1,s(k-1)+1} & \acute{\mathfrak{R}}_{k-1,k} \\ 0_{s,s(k-1)+1} & \acute{R}_k \end{pmatrix} = \begin{pmatrix} I_{s(k-1),s(k-1)} & \mathfrak{R}_{k-1,k} \\ 0_{s,s(k-1)} & R_k \end{pmatrix}, \text{ and} \\
\underline{\mathfrak{R}}_k &= \begin{pmatrix} I_{s(k-1)+1,s(k-1)+1} & \acute{\mathfrak{R}}_{k-1,k} \\ 0_{s,s(k-1)+1} & \underline{\acute{R}}_k \end{pmatrix} = \begin{pmatrix} I_{s(k-1),s(k-1)} & \underline{\mathfrak{R}}_{k-1,k} \\ 0_{s,s(k-1)} & \underline{R}_k \end{pmatrix}.
\end{aligned} \tag{3.16}$$

3.3.4 Updating the upper Hessenberg matrix

In this section, we show how to compute the upper Hessenberg matrix $\underline{\mathfrak{H}}_k$ efficiently, using the structure of the QR factorization update at outer iteration k. Recall that if CA-Arnoldi performs all its iterations without breaking down, it produces the following matrix relation:

$$A\mathfrak{Q}_k = \underline{\mathfrak{Q}}_k \underline{\mathfrak{H}}_k,$$

where

$$\begin{aligned}
\mathfrak{Q}_k &= [Q_0, Q_1, \ldots, Q_k], \\
\underline{\mathfrak{Q}}_k &= [\mathfrak{Q}_k, q_{s(k+1)+1}],
\end{aligned}$$

$$\mathfrak{H}_k = \begin{cases} H_0 & \text{for } k = 0, \\ \begin{pmatrix} \mathfrak{H}_{k-1} & \mathfrak{H}_{k-1,k} \\ h_{k-1} e_1 e_{s(k-1)}^T & H_k \end{pmatrix} & \text{for } k > 0, \end{cases}$$

and

$$\underline{\mathfrak{H}}_k = \begin{pmatrix} \mathfrak{H}_k \\ h_k e_{s(k+1)}^T \end{pmatrix}.$$

That is the end result, but how do we update the upper Hessenberg matrix $\underline{\mathfrak{H}}_k$? Suppose that we have already computed $\underline{\mathfrak{H}}_{k-1}$ and the new basis $\underline{V}_k$. Then, $A\mathfrak{Q}_{k-1} = \underline{\mathfrak{Q}}_{k-1}\underline{\mathfrak{H}}_{k-1}$ and thus

$$A[\mathfrak{Q}_{k-1}, V_k] = [\mathfrak{Q}_k, \underline{V}_k]\underline{\mathfrak{B}}_k, \tag{3.17}$$

where

$$\begin{aligned}
\mathfrak{B}_k &= \begin{pmatrix} \mathfrak{H}_{k-1} & 0_{s(k-1),s} \\ h_{k-1} e_1 e_{s(k-1)}^T & B_k \end{pmatrix}, \text{ and} \\
\underline{\mathfrak{B}}_k &= \begin{pmatrix} \mathfrak{H}_{k-1} & 0_{s(k-1),s} \\ h_{k-1} e_1 e_{s(k-1)}^T & \underline{B}_k \end{pmatrix}.
\end{aligned}$$

Now, we combine the QR factorization update in Section 3.3.3 with the Arnoldi relation $A\mathfrak{Q}_k = \underline{\mathfrak{Q}}_k \underline{\mathfrak{H}}_k$:

$$\begin{aligned}
A[\mathfrak{Q}_{k-1}, Q_k] \begin{pmatrix} I_{s(k-1),s(k-1)} & \mathfrak{R}_{k-1,k} \\ 0_{s,s(k-1)} & R_k \end{pmatrix} = \\
[\mathfrak{Q}_{k-1}, \underline{Q}_k] \begin{pmatrix} I_{s(k-1),s(k-1)} & \underline{\mathfrak{R}}_{k-1,k} \\ 0_{s,s(k-1)} & \underline{R}_k \end{pmatrix} \begin{pmatrix} \mathfrak{H}_{k-1} & 0_{s(k-1),s} \\ h_{k-1} e_1 e_{s(k-1)}^T & \underline{B}_k \end{pmatrix}.
\end{aligned}$$

Here, we use the repartitioning of the R factors described in Section 3.3.3 so that the sub-blocks of $\mathfrak{R}$, $\underline{\mathfrak{R}}$, and $\underline{\mathfrak{B}}_k$ line up for the multiplication. Assuming that the Arnoldi recurrence hasn't broken down, we then have

$$\begin{aligned}\underline{\mathfrak{H}}_k = \underline{\mathfrak{R}}_k \underline{\mathfrak{B}}_k \mathfrak{R}_k^{-1} = \\ \begin{pmatrix} I_{s(k-1),s(k-1)} & \underline{\mathfrak{R}}_{k-1,k} \\ 0 & \underline{R}_k \end{pmatrix} \begin{pmatrix} \underline{\mathfrak{H}}_{k-1} & 0_{s(k-1),s} \\ h_{k-1} e_1 e_{s(k-1)}^T & \underline{B}_k \end{pmatrix} \begin{pmatrix} I_{s(k-1),s(k-1)} & -\mathfrak{R}_{k-1,k} R_k^{-1} \\ 0 & \underline{R}_k^{-1} \end{pmatrix} = \\ \begin{pmatrix} \mathfrak{H}_{k-1} + h_{k-1}(\underline{\mathfrak{R}}_{k-1,k} e_1) e_{s(k-1)}^T & \underline{\mathfrak{R}}_{k-1,k} \underline{B}_k \\ h_{k-1}(\underline{R}_k e_1) e_{s(k-1)}^T & \underline{R}_k \underline{B}_k \end{pmatrix} \begin{pmatrix} I & -\mathfrak{R}_{k-1,k} R_k^{-1} \\ 0 & \underline{R}_k^{-1} \end{pmatrix}. \end{aligned} \quad (3.18)$$

We show in parentheses those terms in the last line of Equation (3.18) above which can be simplified. Two simplifications follow from the properties of the CA-Arnoldi recurrence:

- $\underline{R}_k e_1 = 1$ and $R_k e_1 = 1$, since at each outer iteration k, the first basis vector $v_{sk+1} = q_{sk+1}$ always has unit length by construction.
- $\underline{\mathfrak{R}}_{k-1,k} e_1 = \underline{\mathfrak{Q}}_{k-1}^* \underline{V}_k e_1 = \underline{\mathfrak{Q}}_{k-1}^* q_{sk+1} = 0_{s(k-1),1}$, and $\mathfrak{R}_{k-1,k} e_1 = 0_{s(k-1),1}$ also. This makes only a mild assumption on the orthogonality of the Arnoldi basis vectors, namely that the first basis vector q_{sk+1} of the $s+1$ basis vectors at outer iteration k are orthogonal to the previous group of s basis vectors.

These two simplifications will ensure that the matrix $\underline{\mathfrak{H}}_k$ has an upper Hessenberg structure for all k. We now apply the simplifications to Equation (3.18) to obtain

$$\begin{aligned}\underline{\mathfrak{H}}_k = \begin{pmatrix} \mathfrak{H}_{k-1} + h_{k-1}(\underline{\mathfrak{R}}_{k-1,k} e_1) e_{s(k-1)}^T & \underline{\mathfrak{R}}_{k-1,k} \underline{B}_k \\ h_{k-1}(\underline{R}_k e_1) e_{s(k-1)}^T & \underline{R}_k \underline{B}_k \end{pmatrix} \begin{pmatrix} I & -\mathfrak{R}_{k-1,k} R_k^{-1} \\ 0 & \underline{R}_k^{-1} \end{pmatrix} = \\ \begin{pmatrix} \mathfrak{H}_{k-1} & \underline{\mathfrak{R}}_{k-1,k} \underline{B}_k \\ h_{k-1} e_1 e_{s(k-1)}^T & \underline{R}_k \underline{B}_k \end{pmatrix} \begin{pmatrix} I & -\mathfrak{R}_{k-1,k} R_k^{-1} \\ 0 & \underline{R}_k^{-1} \end{pmatrix} = \\ \begin{pmatrix} \mathfrak{H}_{k-1} & -\mathfrak{H}_{k-1} \mathfrak{R}_{k-1,k} R_k^{-1} + \underline{\mathfrak{R}}_{k-1,k} \underline{B}_k R_k^{-1} \\ h_{k-1} e_1 e_{s(k-1)}^T & \underline{RB}_k R_k^{-1} - h_{k-1} e_1 e_{s(k-1)}^T \mathfrak{R}_{k-1,k} R_k^{-1} \end{pmatrix}. \end{aligned}$$

All these simplifications and calculations give us

$$\underline{\mathfrak{H}}_k = \begin{pmatrix} \mathfrak{H}_{k-1} & -\mathfrak{H}_{k-1} \mathfrak{R}_{k-1,k} R_k^{-1} + \underline{\mathfrak{R}}_{k-1,k} \underline{B}_k R_k^{-1} \\ h_{k-1} e_1 e_{s(k-1)}^T & \underline{RB}_k R_k^{-1} - h_{k-1} e_1 e_{s(k-1)}^T \mathfrak{R}_{k-1,k} R_k^{-1} \end{pmatrix}. \quad (3.19)$$

Let $\underline{H}_k$ be the lower left $s+1$ by s submatrix of $\underline{\mathfrak{H}}_k$. We can further simplify this as follows:

$$\underline{R}_k \underline{B}_k R_k^{-1} = \begin{pmatrix} R_k & z_k \\ 0_{1,k} & \rho_k \end{pmatrix} \begin{pmatrix} B_k \\ b_k e_s^T \end{pmatrix} R_k = \begin{pmatrix} R_k B_k R_k^{-1} + \tilde{\rho}_k^{-1} b_k z_k e_s^T \\ \tilde{\rho}_k^{-1} \rho_k b_k e_s^T \end{pmatrix},$$

where we write $\tilde{\rho}_k = R_k(s,s)$. As long as B_k is upper Hessenberg, $\underline{R}_k \underline{B} R_k^{-1}$ is upper Hessenberg. From this, we have

$$\underline{H}_k = \begin{pmatrix} R_k B_k R_k^{-1} + \tilde{\rho}_k^{-1} b_k z_k e_s^T - h_{k-1} e_1 e_{s(k-1)}^T \mathfrak{R}_{k-1,k} R_k^{-1} \\ \tilde{\rho}_k^{-1} \rho_k b_k e_s^T \end{pmatrix}, \quad (3.20)$$

which is clearly upper Hessenberg. This gives us

$$H_k = R_k B_k R_k^{-1} + \tilde{\rho}_k^{-1} b_k z_k e_s^T - h_{k-1} e_1 e_{s(k-1)}^T \mathfrak{R}_{k-1,k} R_k^{-1} \tag{3.21}$$

and

$$h_k = \tilde{\rho}_k^{-1} \rho_k b_k. \tag{3.22}$$

All these equations give us an update procedure for $\underline{\mathfrak{H}}_k$:

1: Compute $\underline{\mathfrak{H}}_{k-1,k} := -\mathfrak{H}_{k-1} \mathfrak{R}_{k-1,k} R_k^{-1} + \underline{\mathfrak{R}}_{k-1,k} \underline{B}_k R_k^{-1}$
2: Compute H_k via Equation (3.21)
3: Compute $h_k := \tilde{\rho}_k^{-1} \rho_k b_k$
4: $\underline{\mathfrak{H}}_k := \begin{pmatrix} \mathfrak{H}_{k-1} & \underline{\mathfrak{H}}_{k-1,k} \\ h_{k-1} e_1 e_{s(k-1)}^T & H_k \\ 0_{1,s(k-1)} & h_k e_s^T \end{pmatrix}$

The total cost in floating-point operations of the $\underline{\mathfrak{H}}_k$ update, for $k > 0$, is $4s^3(k + 1/4) + O(s^2 k)$ flops. In parallel, this requires no communication, as long as it is computed redundantly on all processors. In the sequential case, this can be computed in fast memory as long as s and k are of the small sizes expected in CA-Arnoldi. If we were to compute $\underline{\mathfrak{H}}_k$ as $\underline{\mathfrak{R}}_k \underline{\mathfrak{B}}_k \mathfrak{R}_k^{-1}$, assuming only that the R factors are upper triangular and that $\underline{\mathfrak{B}}_k$ is upper Hessenberg, this would cost $2s^3k^3/3 + O(s^2k^2)$ flops.

Computational notes

Note that in CA-Arnoldi, it is not strictly necessary to compute $\underline{\mathfrak{H}}_k$ before starting the next outer iteration. The h_k quantity tells us the magnitude of the last basis vector before normalization, which gives us an idea of the convergence of Arnoldi. This can be computed as soon as $\underline{R}_k$ is available from the QR factorization update. In fact, if we expect to run CA-Arnoldi for several outer iterations before converging, we can delay computing $\underline{\mathfrak{H}}_k$ for as long as we wish. Alternately, we can overlap the $\underline{\mathfrak{H}}_k$ update (mainly the computation of $\underline{\mathfrak{H}}_{k-1,k}$ and H_k) with the next outer iteration.

One application of the Arnoldi process is the GMRES algorithm of Saad and Schultz [209] for solving $Ax = b$. Saad and Schultz use a Givens rotation at every iteration of GMRES to maintain a QR factorization of the Arnoldi upper Hessenberg matrix. This efficiently computes the residual $\|b - Ax_k\|_2$ for the current approximate solution x_k, without needing to compute x_k itself. It would not be difficult to modify CA-Arnoldi to use Givens rotations in a similar way, to maintain a QR factorization of $\underline{\mathfrak{H}}_k$. We outline how to do this in Section 3.4.

3.3.5 CA-Arnoldi algorithm

CA-Arnoldi is equivalent to standard Arnoldi iteration in exact arithmetic. The integer parameter s (Arnoldi(s, t)) refers to the number of steps in each inner iteration and is the parameter in the matrix powers kernel optimization which we described in previous sections. The integer parameter t refers to the number of steps in each outer iteration, that is, the number of times an s-step basis is generated before the Arnoldi method restarts. Thus, CA-Arnoldi is equivalent to Arnoldi(st) in exact arithmetic. Algorithm 22 shows the Block CGS update version of CA-Arnoldi.

Algorithm 22 CA-Arnoldi

Require: $n \times n$ matrix A and $n \times 1$ starting vector v

1: $h_0 := \|r\|_2$, $q_1 := v/h_0$
2: **for** $k = 0$ to $t - 1$ **do**
3: Fix basis conversion matrix $\underline{B}_k$
4: Compute $\underline{\acute{V}}_k$ using matrix powers kernel
5: **if** $k = 0$ **then**
6: Compute QR factorization $\underline{V}_0 = \underline{Q}_0 \underline{R}$
7: $\underline{\mathfrak{Q}}_0 := \underline{Q}_0$
8: $\underline{\mathfrak{H}}_0 := \underline{R}_0 \underline{B}_0 R_0^{-1}$
9: **else**
10: $\underline{\acute{\mathfrak{R}}}_{k-1,k} := \underline{\mathfrak{Q}}_{k-1}^* \underline{\acute{V}}_k$
11: $\underline{\acute{V}}_k^{'} := \underline{\acute{V}}_k - \underline{\mathfrak{Q}}_{k-1} \underline{\acute{\mathfrak{R}}}_{k-1,k}$
12: Compute QR factorization $\underline{\acute{V}}_k^{'} = \underline{\acute{Q}}_k \underline{\acute{R}}_k$
13: Compute $\underline{\mathfrak{H}}_{k-1,k} := -\mathfrak{H}_{k-1} \mathfrak{R}_{k-1,k} R_k^{-1} + \underline{\mathfrak{R}}_{k-1,k} \underline{B}_k R_k^{-1}$
14: Compute H_k via Equation (3.21)
15: Compute h_k via Equation (3.22)
16: $\underline{\mathfrak{H}}_k := \begin{pmatrix} \mathfrak{H}_{k-1} & \underline{\mathfrak{H}}_{k-1,k} \\ h_{k-1} e_1 e_{s(k-1)}^T & H_k \\ 0_{1,s(k-1)} & h_k e_s^T \end{pmatrix}$
17: **end if**
18: **end for**

3.3.6 Generalized eigenvalue problems

CA-Arnoldi may also be used to solve generalized eigenvalue problems of the form

$$Ax = \lambda M x, \tag{3.23}$$

where A and M are both nonsingular $n \times n$ matrices. In that case, we may convert Equation (3.23) into an equivalent form, such as

$$M^{-1} A x = \lambda x. \tag{3.24}$$

See Section 4.3 for details and a list of different equivalent forms. The result could be called a "preconditioned eigenvalue problem," that is an eigenvalue problem with respect to a "preconditioned matrix." (which is $M^{-1}A$ in the above "left-preconditioned" example). Such problems may also be solved using CA-Arnoldi, if we change Line 4 of Algorithm 22 to compute a Krylov basis with respect to the preconditioned matrix, rather than to the original matrix A. For example, if the preconditioned matrix is $M^{-1}A$, then the columns of $\underline{V}_k$ must form a basis of

$$\mathcal{K}_{s+1}(M^{-1}A, v_{sk+1}) = \text{span}\{v_{sk+1}, M^{-1}Av_{sk+1}, (M^{-1}A)^2 v_{sk+1}, \ldots, (M^{-1}A)^s v_{sk+1}\}$$

at each outer iteration k of Algorithm 3.3. To compute this, we use the preconditioned version of the matrix powers kernel (Section 2.2. CA-Arnoldi requires no other changes in

order to solve generalized eigenvalue problems in a communication-avoiding way. (Some of our other communication-avoiding Krylov methods, such as CA-Lanczos, do require changes in this case. The algorithm and the matrix powers kernel both change. See Section 4.3 for details.)

3.3.7 Summary

In this section, we presented CA-Arnoldi (Algorithm 22). It communicates a factor s less than standard Arnoldi, both sequentially and in parallel. CA-Arnoldi uses a combination of the matrix powers kernel (Section 2.1), Block Gram-Schmidt (Section 2.4), and TSQR (Section 2.3). It is equivalent to standard Arnoldi in exact arithmetic. We will use CA-Arnoldi in Section 3.4 to derive CA-GMRES (Algorithm 23), which is a communication-avoiding version of GMRES.

3.4 CA-GMRES

The Generalized Minimal Residual method (GMRES) of Saad and Schultz [209] is a Krylov subspace method for solving nonsymmetric linear systems. It does so by choosing the solution update from the span of the Arnoldi basis vectors that minimizes the 2-norm residual error. This entails solving a least-squares problem with the Arnoldi upper Hessenberg matrix, which is $m+1$ by m at iteration m of GMRES. Saad and Schultz's formulation of GMRES combines (standard, MGS-based) Arnoldi with a Givens rotation scheme for maintaining a QR factorization of the upper Hessenberg matrix at each iteration. The Givens rotation scheme exposes the residual error of the least-squares problem at every iteration j, which (in exact arithmetic) is the GMRES residual error $\|b - Ax_j\|_2$ in iteration j. This means that GMRES need not compute the approximate solution x_j in order to estimate convergence.

We can compute exactly the same approximate solution as m iterations of Saad and Schultz's GMRES if we perform m iterations of Arnoldi(s, t), where s and t are chosen so that $m = s \cdot t$. This results in an $st+1$ by st upper Hessenberg matrix $\underline{\mathfrak{H}}_{t-1}$. We can even use Givens rotations to maintain its QR factorization and get the same convergence estimate as in Saad. The only difference is that we have to perform s Givens rotations at a time, since Arnoldi(s, t) adds s columns at a time to the upper Hessenberg matrix with every outer iteration. The Givens rotations technique has little effect on raw performance (one would normally solve the least-squares problem with a QR factorization and use Givens rotations to factor the upper Hessenberg matrix anyway), but it offers an inexpensive convergence test at every outer iteration. The resulting algorithm, based on Arnoldi(s, t) and equivalent to standard GMRES in exact arithmetic, we call *Communication-Avoiding GMRES* or CA-GMRES (Algorithm 23).

CA-GMRES includes all of Arnoldi(s, t) as shown in Algorithm 22, and also at every outer iteration k, solves the least-squares problem

$$y_k := \text{argmin}_y \|\underline{\mathfrak{H}}_k y - \beta e_1\|_2 \tag{3.25}$$

to compute a new approximate solution $x_{k+1} := x_0 + \mathfrak{Q}_k y_k$. Here, $\beta = \|b - Ax_0\|_2$, the 2-norm of the initial residual. Algorithm 23 shows how we update a QR factorization of

$\underline{\mathfrak{H}}_k$ by first applying all the previous $sk+1$ Givens rotations $G_1, \ldots, G_{sk+1}$ to the new s columns of $\underline{\mathfrak{H}}_k$, and then applying s more Givens rotations $G_{sk+2}, \ldots, G_{s(k+1)+1}$ to $\underline{H}_k$ to reduce it to upper triangular form. The Givens rotations are also applied to the right-hand side of the least-squares problem, resulting in what we denote at outer iteration k by ζ_k. The last component of ζ_k, in exact arithmetic, equals the two-norm of the absolute residual error $b - Ax_{k+1}$.

3.4.1 Scaling of the first basis vector

In Section 3.2.4, we discussed how some QR factorizations may give the basis vectors q_j computed by Arnoldi(s, t) a different unitary scaling than the basis vectors $\hat{q}_j$ computed by standard Arnoldi. Similarly, the upper Hessenberg matrices may differ by a unitary diagonal similarity transform. This does not affect Ritz value calculations, but it does affect the least-squares problem which GMRES solves in order to get the approximate solution to $Ax = b$. We will now show that for CA-GMRES, the only scaling factor that matters is that of the first basis vector q_1. Furthermore, if CA-GMRES uses a QR factorization under which the direction of the first column is invariant (i.e., where $\langle q_1, \hat{q}_1 \rangle = 1$ as long as $\|q_1\|_2 = 1$), then CA-GMRES computes the same approximate solution in exact arithmetic as standard GMRES at all iterations. Otherwise, one additional inner product will be necessary in order to compute the change in direction of q_1.

We review notation from Section 3.2.4 to describe the unitary scaling of the basis vectors. If we run standard MGS Arnoldi for st iterations (with s and $t \geq 1$) with the unit-length starting vector $\hat{q}_1$, it produces st more basis vectors $\hat{q}_2, \ldots, \hat{q}_{st+1}$ and an $st+1$ by st upper Hessenberg matrix $\underline{\hat{\mathfrak{H}}}$ such that

$$A\hat{\mathfrak{Q}} = \underline{\hat{\mathfrak{Q}}}\,\underline{\mathfrak{H}},$$

where $\hat{\mathfrak{Q}} = [q_1, \ldots, q_{st}]$ and $\underline{\hat{\mathfrak{Q}}} = [\hat{\mathfrak{Q}}, q_{st+1}]$. If we run Arnoldi($s$, t) with the same s and t as above, with the same starting vector $\hat{q}_1$, we obtain $st+1$ basis vectors q_1, q_2, $\ldots$, q_{st+1} and an $st+1$ by st upper Hessenberg matrix $\underline{\mathfrak{H}}_{t-1}$ such that

$$A\mathfrak{Q}_{t-1} = \underline{\mathfrak{Q}}_{t-1}\underline{\mathfrak{H}}_{t-1}.$$

The matrices $\underline{\mathfrak{Q}}_{t-1}$ and $\underline{\hat{\mathfrak{Q}}}$ may differ in exact arithmetic by a unitary scaling:

$$\underline{\hat{\mathfrak{Q}}} = \underline{\mathfrak{Q}}_{t-1}\underline{\Theta},$$

in which $\underline{\Theta}$ is an $st+1$ by $st+1$ diagonal matrix with elements $\theta_1, \theta_2, \ldots, \theta_{st+1}$. For all j, $|\theta_j| = 1$, and $\underline{\Theta}^{-1} = \underline{\Theta}^*$. We write Θ for the st by st upper left submatrix of $\underline{\Theta}$. Similarly, the relationship between the two upper Hessenberg matrices is

$$\underline{\hat{\mathfrak{H}}} = \underline{\Theta}^*\underline{\mathfrak{H}}\Theta. \tag{3.26}$$

Suppose now that iteration st of standard GMRES computes an approximate solution $\hat{x} = x_0 + \hat{\mathfrak{Q}}\hat{y}$, and our version of GMRES computes an approximate solution $x_t = x_0 + \mathfrak{Q}_{t-1}y_{t-1}$. We can use Equation (3.26) to compare these two solution updates $\hat{y}$ and y_{t-1}. Our version of GMRES obtains y_{t-1} as the solution of the least-squares problem

$$y_{t-1} = \text{argmin}_y \|\underline{\mathfrak{H}}_{t-1}y - \beta e_1\|_2,$$

where β is the initial residual norm $\|b - Ax_0\|_2$. Similarly, standard GMRES obtains $\hat{y}$ as the solution of

$$\hat{y} = \text{argmin}_y \|\hat{\underline{\mathfrak{H}}}_{t-1} y - \beta e_1\|_2.$$

The pseudoinverse formula for the least-squares problem gives us

$$y_{t-1} = \beta \left(\underline{\mathfrak{H}}_{t-1}^* \underline{\mathfrak{H}}_{t-1}\right)^{-1} \underline{\mathfrak{H}}_{t-1}^* e_1$$

and

$$\hat{y} = \beta \left(\hat{\underline{\mathfrak{H}}}^* \hat{\underline{\mathfrak{H}}}\right)^{-1} \hat{\underline{\mathfrak{H}}}^* e_1.$$

If we apply Equation (3.26), we get

$$\begin{aligned}
\hat{y} &= \beta \left(\Theta^* \underline{\mathfrak{H}}_{t-1}^* \underline{\Theta\Theta}^* \underline{\mathfrak{H}}_{t-1} \Theta\right)^{-1} \Theta^* \underline{\mathfrak{H}}^* \underline{\Theta} e_1 \\
&= \beta \left(\Theta^* \underline{\mathfrak{H}}_{t-1}^* \underline{\mathfrak{H}}_{t-1} \Theta\right)^{-1} \Theta^* \underline{\mathfrak{H}}^* \underline{\Theta} e_1 \\
&= \beta \Theta^* \left(\underline{\mathfrak{H}}_{t-1}^* \underline{\mathfrak{H}}_{t-1}\right)^{-1} \Theta\Theta^* \underline{\mathfrak{H}}^* \underline{\Theta} e_1 \\
&= \beta \Theta^* \left(\underline{\mathfrak{H}}_{t-1}^* \underline{\mathfrak{H}}_{t-1}\right)^{-1} \underline{\mathfrak{H}}^* \theta_1 e_1 \\
&= \beta \theta_1 \Theta^* \left(\underline{\mathfrak{H}}_{t-1}^* \underline{\mathfrak{H}}_{t-1}\right)^{-1} \underline{\mathfrak{H}}^* e_1.
\end{aligned}$$

Finally,

$$\begin{aligned}
\hat{x} &= x_0 + \hat{\Omega} \hat{y} \\
&= x_0 + \beta \theta_1 \Omega_{t-1} \Theta\Theta^* \left(\underline{\mathfrak{H}}_{t-1}^* \underline{\mathfrak{H}}_{t-1}\right)^{-1} \underline{\mathfrak{H}}^* e_1 \\
&= x_0 + \beta \theta_1 \Omega_{t-1} \left(\underline{\mathfrak{H}}_{t-1}^* \underline{\mathfrak{H}}_{t-1}\right)^{-1} \underline{\mathfrak{H}}^* e_1 \\
&= x_0 + \theta_1 \Omega_{t-1} y_{t-1}.
\end{aligned}$$

This differs from $x_{k+1} = x_0 + \Omega_{t-1} y_{t-1}$ by the direction θ_1 of the solution update. If $\theta_1 \neq 1$, the computed solution x_{k+1} will differ from the standard GMRES solution.

One way to fix this is to ensure that the QR factorization used in CA-GMRES always results in an R factor with a nonnegative diagonal (see Demmel et al. [76]). This will preserve the direction of the initial residual r_0, so that $q_1 = r_0/\beta$. Another way is to orthogonalize the first group of basis vectors $\underline{V}_0$ using MGS, instead of some other QR factorization. This happens naturally if the first outer iteration of CA-GMRES is performed using standard GMRES, which one might do in order to compute Ritz values to use in the matrix powers kernel basis. A third approach is to compute $\theta_1 = \langle r_0, q_1 \rangle / \beta$ via an inner product. This requires an additional communication step, but it need only be performed once, after the QR factorization of $\underline{V}_0$. One might be tempted to avoid the inner product in the parallel case by comparing corresponding components of r_0/β and q_1, since both r_0/β and q_1 are unit length and colinear. In finite-precision arithmetic, however, the processors may end up disagreeing on the value of θ_1. Furthermore, this communication-free procedure would fail in certain corner cases where one or more processor's components of the residual vector are all zero. If avoiding the latency of the extra inner product matters for performance, then one could package it up in the next outer iteration's communication steps.

3.4.2 Convergence metrics

CA-GMRES (Algorithm 23) computes a convergence metric at every outer iteration, namely the 2-norm absolute residual $\|r_{k+1}\|_2$. It does so inexpensively, in a way that usually requires little computation and no communication. Scaling this by β, the initial residual error, results in the relative residual $\|r_{k+1}\|_2/\|r_0\|_2$.

Many authors recommend other convergence metrics for iterative methods. See e.g., Barrett et al. [23, pp. 57–63]. This is because the relative residual error depends too strongly on the initial guess x_0. If $x_0 = 0$, then it may be hard to make the relative residual small if A is ill-conditioned and x_{k+1} lies nearly in the null space of A. In contrast, if $\|x_0\|$ is large, then $\beta = \|r_0\|_2$ will be large, which may cause the iteration to stop too early (as the relative residual will be artificially small).

Some of the recommended convergence metrics require computing the current approximate solution norm $\|x_{k+1}\|$. Others, that account for the sparsity structure of A, may even require a sparse matrix-vector product, as well as a norm. Norms involve summation or finding a maximum element in absolute value, which entails communication: sequentially, all the elements of a long vector must be read, and in parallel, the processors must perform a reduction. An SpMV operation will also involve communication. For example, in parallel, even if the structure of A completely decouples, some kind of synchronization is necessary so that the vector output can be reduced to a norm.

Whether users employ standard GMRES or our CA-GMRES, they must decide whether a good convergence metric is worth the cost. Since the two methods are equivalent in exact arithmetic and converge at the same rate in practice for most problems, one would use the same convergence metric for each. Thus, the convergence metric is just an additional cost atop the chosen iterative method. When we compare the performance and convergence of the two iterative methods, we use the already available relative residual metric. In practice, one could monitor the relative residual and compute a more expensive metric only occasionally.

Both CA-GMRES and standard GMRES have an inexpensive way to compute the two-norm of the current solution, if $x_0 = 0$. In that case, the current solution is a linear combination of unit-length orthogonal vectors (in exact arithmetic), and so the two-norm of the coefficients gives the two-norm of the current solution.

3.4.3 Preconditioning

To get preconditioned CA-GMRES, replace the matrix A in Algorithm 23 with the preconditioned matrix: $L^{-1}AL^{-T}$ for a split preconditioner LL^T, $M^{-1}A$ for a left preconditioner M, or AM^{-1} for a right preconditioner M. The CA-GMRES algorithm does not otherwise change, but the matrix powers kernel must change, as discussed in Section 2.2. One must use a preconditioner compatible with the matrix powers kernel. The performance of preconditioned CA-GMRES depends on the preconditioner. In some cases, it may be preferable to compute the same output as the matrix powers kernel using standard sparse matrix-vector multiplications and preconditioner applications. This is especially useful for longer restart lengths, where orthogonalizing the Arnoldi basis vectors dominates the runtime, as TSQR and block Gram-Schmidt are usually much faster than standard MGS.

3.5 CA-GMRES numerical experiments

In this section, we describe numerical experiments that compare standard GMRES with our CA-GMRES, as well as with earlier versions of s-step GMRES. The experiments show that CA-GMRES (with the correct choice of basis and restart length $r = s \cdot t$) converges in no more iterations than standard GMRES (with the same restart length r) for almost all practical and contrived examples. They demonstrate that our (novel) ability to choose the s-step basis length smaller than the restart length improves convergence. (In Section 3.6, we will also show that this ability improves performance.) In addition, the experiments show that in many cases, equilibration (see Section 7.5.3) and / or choosing a basis other than the monomial basis (see Section 7.3) are required for numerical stability. Equilibration may slow convergence of CA-GMRES relative to standard GMRES for some extremely poorly scaled matrices, but equilibration also allows the use of longer s-step basis lengths in such cases.

We begin this section with a key explaining the notation of the convergence plots, in Section 3.5.1. This section contains four different types of numerical experiments. Details on the matrix, right-hand side, and initial guess used for each experiment are found in Appendix B.

1. Positive diagonal matrices with varying condition numbers, which are described in more detail in Appendix B.1. The results of these experiments are shown in Section 3.5.2.

2. Discretizations of 2-D convection-diffusion PDEs, with physically relevant parameters that allow us to control the nonsymmetry of the matrix. The PDE and its discretization are described in Appendix B.4.2 resp. B.4.3. The results of these experiments are shown here in Section 3.5.3.

3. A variety of sparse matrices from different applications, for which we first presented CA-GMRES performance results in Demmel et al. [79]. These matrices are described in Appendix B.5. We show the results of numerical experiments with these matrices here in Section 3.5.4. Performance experiments for this set of matrices will be shown in Section 3.6.2.

4. WATT1: A challenging nonsymmetric matrix from petroleum engineering. We describe this matrix in detail in Appendix B.6, and present numerical experiments here in Section 3.5.5. Performance experiments for the WATT1 matrix will be in Section 3.6.3.

Diagonal matrices help us analyze numerical stability of the s-step basis, as a function of matrix condition number and distribution of the eigenvalues. Results for these matrices should apply to all iterative methods in this thesis, regardless of whether or not they assume the matrix is symmetric. The second class of problems, which are nonsymmetric and come from previous work on s-step versions of GMRES, exercise the GMRES part of CA-GMRES. The third class confirms that CA-GMRES obtains good approximate solutions on matrices from applications, on which it also performs well. The fourth problem poses special challenges to s-step basis stability. Previous work has studied the convergence behavior of the third problem in enough detail for us to make a fair comparison with CA-GMRES. Performance

results for the fourth problem, which we will show in Section 3.5.5, will illustrate the value of our novel ability to choose the basis length shorter than the restart length. We conclude with a summary of our results in Section 3.5.6.

3.5.1 Key for convergence plots

The x axis of all the convergence plots in this thesis shows the number of iterations of the Krylov methods tested. These are iterations in the sense of a standard Krylov method (such as standard GMRES); s of these iterations correspond to one "outer iteration" of a communication-avoiding Krylov method (such as CA-GMRES) with basis length s. The CA-GMRES convergence metric is evaluated and shown only once per outer iteration. The convergence metric for standard GMRES is shown as a continuous line, to suggest that it is evaluated more frequently than the CA-GMRES metric.

The y axis of the convergence plots shows the 2-norm of the relative residual vector: when solving $Ax = b$ with initial guess x_0, the 2-norm of the relative residual at iteration k is $\|b - Ax_k\|_2 / \|b - Ax_0\|_2$. There are arguments for using other convergence metrics. See e.g., Liesen [169] and Barrett et al. [23, "Stopping Criteria"]. Some of these alternate convergence metrics require additional communication, such as computing $\|x_k\|$. We chose the relative residual both for its familiarity, and because it requires no additional communication (it comes "for free" from GMRES itself). Furthermore, since the Krylov methods we compare start with the same initial guess for a given linear system, and since we run the Krylov methods for many iterations, the residual scaling factor does not matter.

Each convergence plot's legend shows data for three kinds of methods:

GMRES(r) Standard GMRES with restart length r. Its convergence metric is always shown as a black continuous line.

Monomial-GMRES(s,t) CA-GMRES with basis length s, restart length $r = s \cdot t$, and the monomial basis. The convergence metric is plotted as a circle, with a different color for each s value shown in the plot.

Newton-GMRES(s,t) CA-GMRES with basis length s, restart length $r = s \cdot t$, and the Newton basis. The convergence metric is plotted as an upward-pointing triangle, with a different color for each s value shown in the plot.

The legend includes other data for the CA-GMRES methods shown in the plot. That data shows the quality of the s-step basis, over each outer iteration k. In CA-GMRES, we write the $s + 1$ vectors in the s-step basis as $\underline{V}_k = [v_{sk+1}, v_{sk+2}, \ldots, v_{sk+s+1}]$ (see Algorithm 23 in Section 3.4). The first basis quality metric shown in the legend is the "min,max basis cond #," i.e., $\min_k \kappa_2(\underline{V}_k)$ and $\max_k \kappa_2(\underline{V}_k)$, the minimum and maximum 2-norm condition number of the s-step basis over all the outer iterations. When $\epsilon \cdot \kappa_2(\underline{V}_k) > 1$ (where ϵ is machine epsilon), the s-step basis is no longer numerically full rank. This often predicts the lack of convergence of the method.

The second s-step basis quality metric shown in the legend is the "min,max basis scaling." In Section 7.5, we explain that if the norm of the matrix A is significantly larger or smaller than 1, then the norms of the successive s-step basis vectors can increase resp. decrease

Figure #	Cond #	Restart length
3.1	10^5	60
3.2	10^{10}	60
3.3	10^{15}	60
3.4	10^5	20
3.5	10^{10}	20
3.6	10^{15}	20

Table 3.1: Figure numbers of convergence plots for diagonal test matrices in this section, corresponding to different test matrices and GMRES restart lengths. Each matrix is identified by it 2-norm condition number ("Cond #").

rapidly. This can lead to overflow resp. underflow. As we discussed in that section, previous authors controlled this process by scaling each basis vector in turn by its norm, before generating the next one. Each norm computation requires communication, and scaling each basis vector by its norm sets up a data dependency that prevents hiding this communication cost. (See Sections 1.1.3 and 1.3.) Our approach, which we discuss in Section 7.5, avoids communication by scaling the rows and columns of the matrix A, before starting the iterative method. Not all matrices A require this. Whether or not it is required, we measure the increase or decrease of the norms of the s-step basis vectors using the "min,max basis scaling" metric. For each outer iteration k, we compute the *average basis scaling*

$$\begin{aligned} \text{AverageBasisScaling}_k &\equiv \|v_{sk+s+1}\|_2^{1/s} \\ &= \left(\frac{\|v_{sk+2}\|_2}{\|v_{sk+1}\|_2} \cdot \frac{\|v_{sk+3}\|_2}{\|v_{sk+2}\|_2} \cdot \cdots \cdot \frac{\|v_{sk+s+1}\|_2}{\|v_{sk+s}\|_2} \right)^{1/s} . \end{aligned} \tag{3.27}$$

The average basis scaling is the geometric mean of the ratios between consecutive s-step basis vectors. Note that $v_{sk+1} = 1$ in CA-GMRES; the input vector of the matrix powers kernel is scaled to have unit length. As with the basis condition number metric, we show the minimum and maximum of the average basis scaling over all outer iterations k.

3.5.2 Diagonal matrices

We first test CA-GMRES with 10000×10000 positive diagonal matrices, with maximum element 1 and a specified 2-norm condition number. See Appendix B.3 for a detailed description of the diagonal test problems used in this thesis. Three different diagonal test matrices were tried, with condition numbers 10^5, 10^{10}, and 10^{15}, respectively. We tested two different restart lengths: 20, and 60. Table 3.1 shows which figure corresponds to which test matrix and restart length.

For restart length $r = 60$, we show CA-GMRES results for $s = 15$ and $s = 20$. These illustrate how CA-GMRES with the monomial basis fails to converge as s increases, whereas CA-GMRES with the Newton basis can still converge even with larger s. For restart length $r = 20$, we show CA-GMRES results for $s = 10$ and $s = 20$. (In the case of restart length 20, $s = 15$ does not divide 20 evenly, so we used $s = 10$ instead.) These illustrate the same phenomenon, except that the effects of a convergence failure are less dramatic when restarting

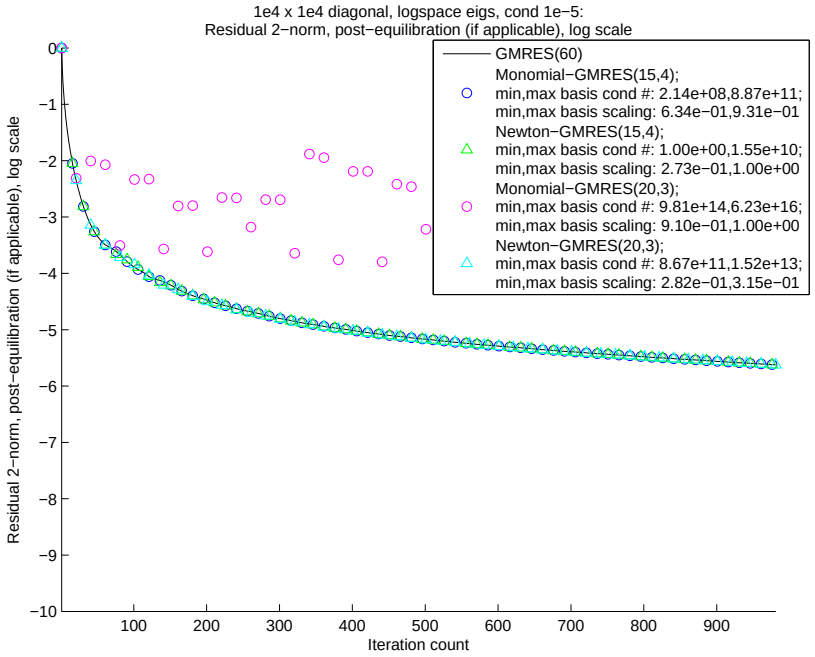


Figure 3.1: Convergence of standard GMRES and CA-GMRES, with restart length 60, on a 10000×10000 positive diagonal test matrix with 2-norm condition number 10^5.

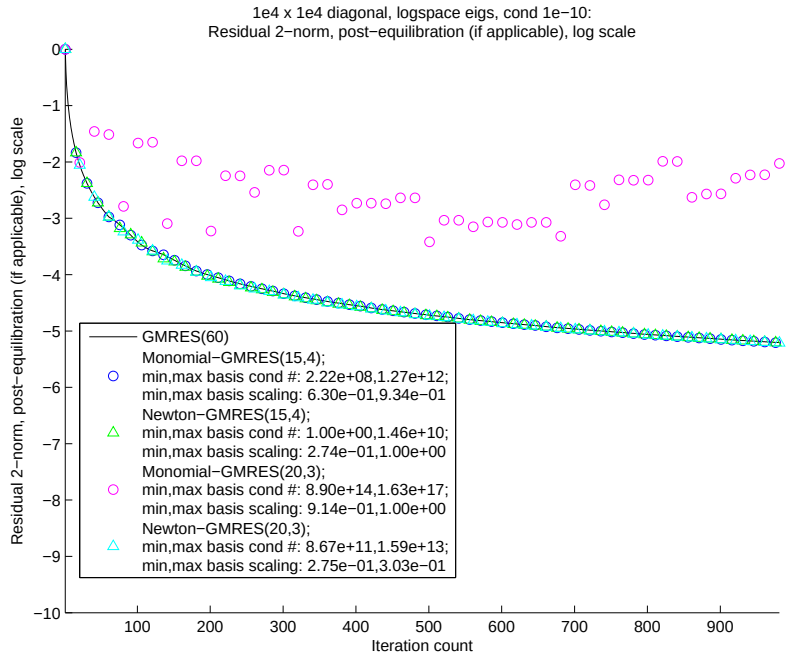


Figure 3.2: Convergence of standard GMRES and CA-GMRES, with restart length 60, on a 10000×10000 positive diagonal test matrix with 2-norm condition number 10^{10}.

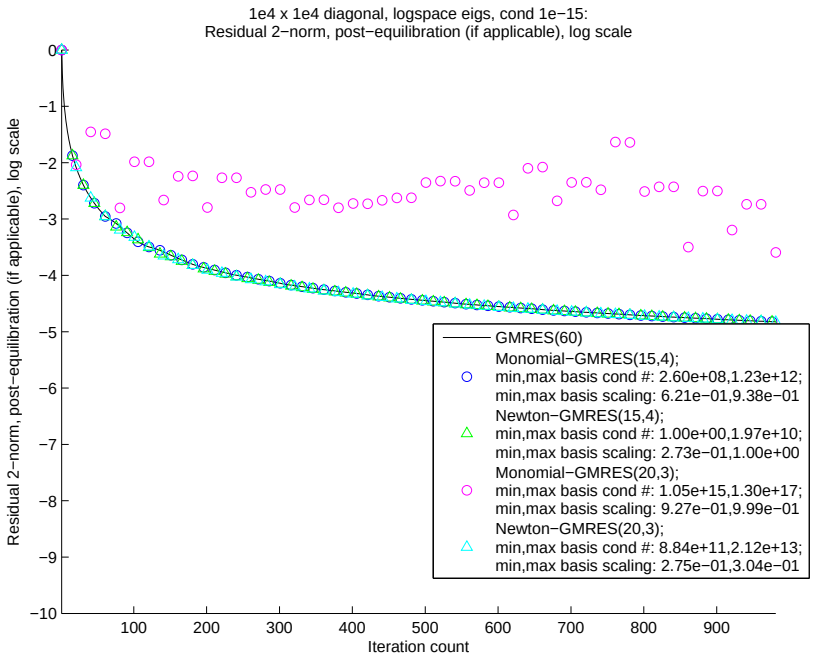


Figure 3.3: Convergence of standard GMRES and CA-GMRES, with restart length 60, on a 10000×10000 positive diagonal test matrix with 2-norm condition number 10^{15}.

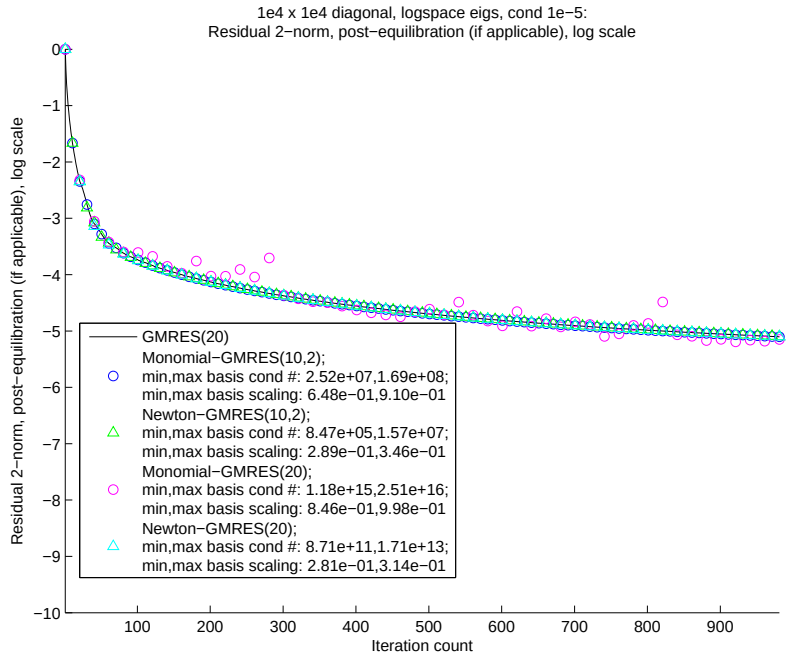


Figure 3.4: Convergence of standard GMRES and CA-GMRES, with restart length 20, on a 10000×10000 positive diagonal test matrix with 2-norm condition number 10^5.

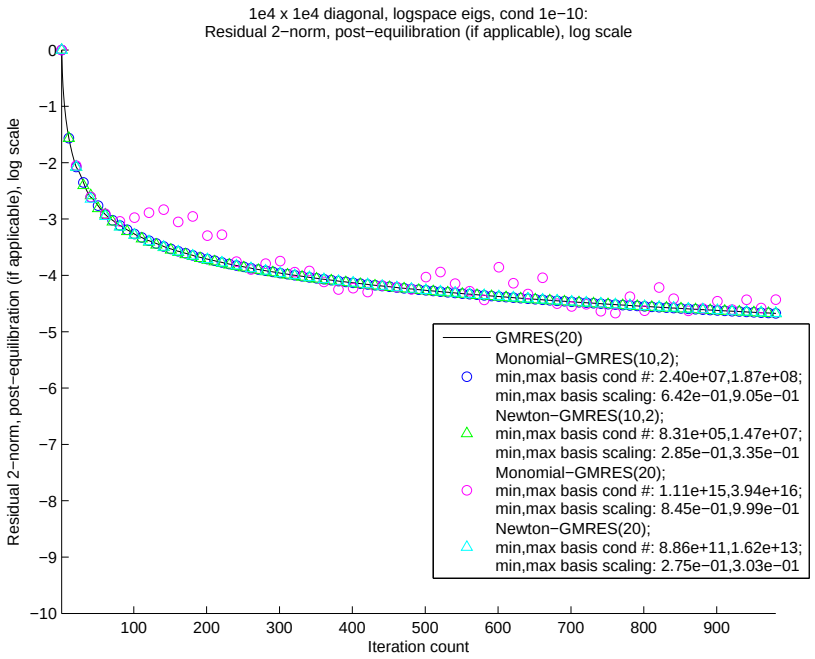


Figure 3.5: Convergence of standard GMRES and CA-GMRES, with restart length 20, on a 10000×10000 positive diagonal test matrix with 2-norm condition number 10^{10}.

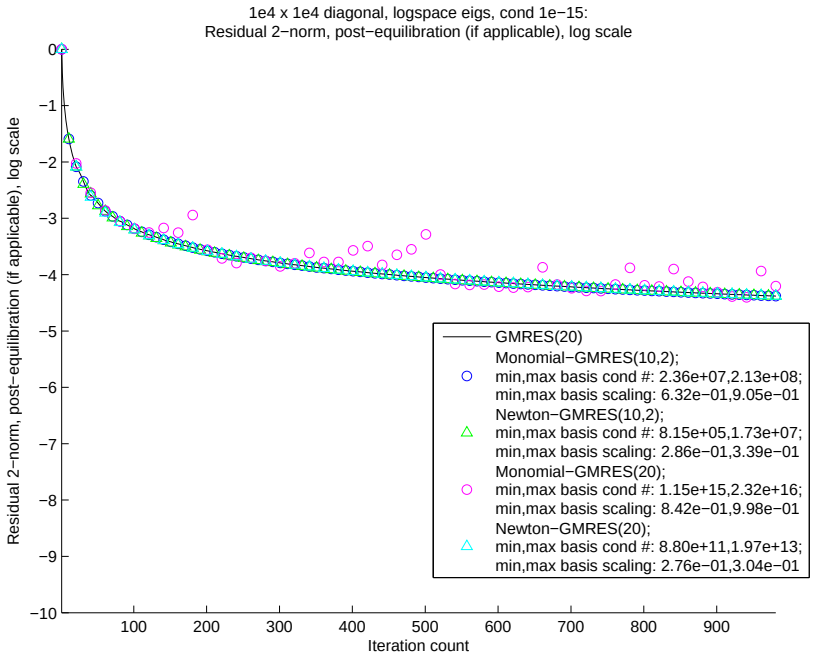


Figure 3.6: Convergence of standard GMRES and CA-GMRES, with restart length 20, on a 10000×10000 positive diagonal test matrix with 2-norm condition number 10^{15}.

Test #	$\|A\|_F$	`condest(A)`	Relative nonsymmetry	Relative departure from normality	Restart length
1	2.8103×10^2	3.4684×10^4	6.9497×10^{-3}	9.83×10^{-3}	20
2	2.8103×10^2	3.4684×10^4	6.9497×10^{-3}	9.83×10^{-3}	30
3	2.8095×10^2	1.4074×10^4	2.1983×10^{-2}	3.11×10^{-2}	25

Table 3.2: Test matrices in the three test problems of Bai et al. [19] (corresponding to Figures 1, 2, and 3 in that paper). Note that Test #1 uses the same matrix as Test #2; the two tests differ only in the GMRES restart length. The "relative nonsymmetry" of a matrix A is $\|(A - A^*)/2\|_F / \|A\|_F$, where $\|\cdot\|_F$ denotes the Frobenius (square root of sum of squares) norm. The "relative departure from normality" of the matrix is the Frobenius norm of the strict upper triangle (not including the diagonal) of the upper triangular Schur factor of A, divided by $\|A\|_F$. See Appendix D.1 for a detailed discussion of the departure from normality.

is more frequent. In all cases, however, CA-GMRES with the "right" basis converges at the same rate as standard GMRES with the same restart length, as it should in exact arithmetic.

3.5.3 Convection-diffusion PDE discretization

In this section, we describe the results of our numerical experiments using the three test problems of Bai et al. [19], as described in Appendix B.4. They come from the finite-difference discretization of a linear convection-diffusion partial differential equation on a 2-D regular mesh with Dirichlet boundary conditions. The resulting matrix is nonsymmetric. The discretization already scales the matrix and right-hand side so the norm of the matrix is close to one, so equilibration is not necessary. We show some details of each of the three test problems in Table 3.2, and show our convergence plots, one for each test problem, as Figures 3.7, 3.8, and 3.9.

In comparing the results of our experiments with those of Bai et al., we first point out that the authors chose the entries of the initial guess x_0 from a random uniform $[-1, 1]$ distribution, without giving details of the random number generator. This makes it hard for us to replicate their results. In fact, even when we tried two different implementations of standard GMRES (the one that comes standard with Matlab, as well as our own), we were unable to duplicate the convergence curve they observed for standard GMRES.[4] For example, compare Figure 1a in Bai et al. [19] with our Figure 3.7. The solid black line in Figure 3.7, representing the convergence of standard GMRES (with restart length 20), should coincide exactly with the solid black line in Figure 1a of Bai et al., since both the algorithms and the matrix are the same, but it does not. Furthermore, that figure shows the monomial basis failing to converge, but our version of monomial-basis CA-GMRES converges just as well as standard GMRES for that problem. These discrepancies could be due to any of the following reasons:

[4] Our version of standard GMRES has additional instrumentation and outputs so that we can use it to "start up" CA-GMRES, for example in computing Ritz values for the Newton basis to use as shifts. We have compared our version of GMRES and the version implemented by The MathWorks that is built in to Matlab, and found them to produce the same numerical results for various matrices.

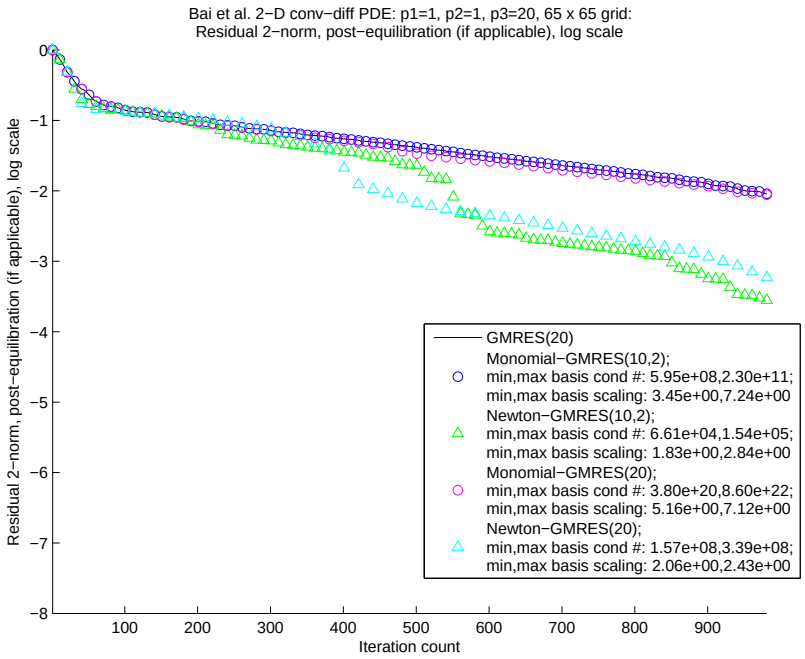


Figure 3.7: Convergence of standard GMRES and CA-GMRES on the first test problem of Bai et al.

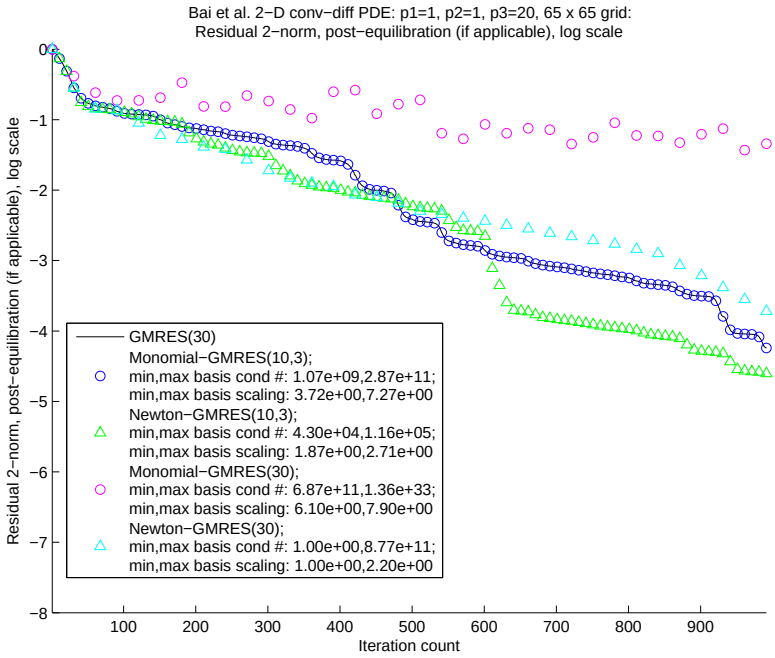


Figure 3.8: Convergence of standard GMRES and CA-GMRES on the second test problem of Bai et al.

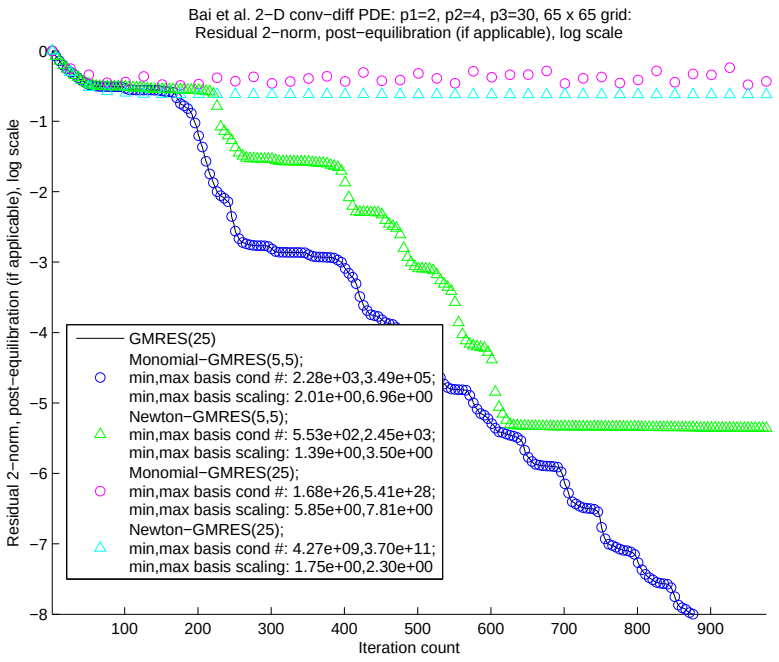


Figure 3.9: Convergence of standard GMRES and CA-GMRES on the third test problem of Bai et al.

- Our random initial guess x_0 is different than theirs
- Our PDE discretization differs in some way from theirs
- Some subtle difference in our implementation of s-step GMRES

This means that we cannot make the desired comparison between the numerical results of Bai et al., and our numerical results. Nevertheless, we can still compare standard GMRES and our CA-GMRES on these test problems.

Another (small) difference between our convergence plots and those of Bai et al., is that theirs show the 2-norm of the relative residual at each outer iteration of their Power-GMRES and Newton-GMRES algorithms, so that the x-axis of Figures 1, 2, and 3 in their paper is the outer iteration count. Our convergence plots show the "inner" iteration count, so one of their x-axis units corresponds to s of ours (where s in their case is the restart length, which is 20 for Figure 1, 30 for Figure 2, and 25 for Figure 3). This difference does not explain the discrepancy between our convergence results.

The major difference, in terms of numerical stability, between Bai et al.'s s-step GMRES algorithm and our CA-GMRES algorithm, is that CA-GMRES may choose the Krylov basis length s to be shorter than the restart length. The algorithm of Bai et al. cannot do this: their s-step basis length must be the same as the restart length. We simulate this restriction with each of the three test problems, by running CA-GMRES with two different values of s: first, with s equal to the restart length r for that test problem, and second, with s being the next smaller integer factor of r. (Our CA-GMRES does not require that s divide the restart length r evenly, but we impose this requirement in our experiments for simplicity.) We find

in all the experiments, that CA-GMRES' increased freedom to choose s independently of the restart length improves numerical stability.

We begin with the first experiment, whose convergence results are shown as Figure 3.7. The restart length is 20, and we compare standard GMRES, and CA-GMRES with both the monomial and Newton bases. For CA-GMRES, we show results for $s = 10$ and $s = 20$. We find, first, that for this problem, all of the bases and basis lengths tested converge at least as fast as standard GMRES (the continuous black line, which is covered by the blue and magenta circles representing monomial-basis CA-GMRES with $s = 10$ resp. $s = 20$). This is true even though the monomial basis for $s = 20$ is ill-conditioned: its 2-norm condition number ranges between 10^{20} and nearly 10^{23}. Consequently, this demonstrates that CA-GMRES can tolerate a severely ill-conditioned s-step basis, in some cases, without affecting convergence significantly. Second, CA-GMRES with the Newton basis actually converges faster than standard GMRES, after around 200 iterations. This result is surprising, since CA-GMRES is equivalent to standard GMRES in exact arithmetic. This suggests that CA-GMRES may be more tolerant of rounding error than standard GMRES. More experiments and analysis are needed to find the cause of this phenomenon.

We show the convergence results of the second experiment as Figure 3.8. The matrix is the same as in the first experiment, but the restart length is 30. We compare standard GMRES, and CA-GMRES with both the monomial and Newton bases. For CA-GMRES, we show results for $s = 10$ and $s = 30$. This plot shows how the condition number of the monomial basis grows with s, and that if the condition number becomes too large, then the convergence of CA-GMRES is affected. Here, the monomial basis with $s = 30$ produces ill-conditioned s-step bases. This results in CA-GMRES failing to converge, and displaying typical wandering behavior. However, the monomial basis with shorter basis length $s = 10$ produces s-step bases that are all numerically full rank, and CA-GMRES' relative residuals (the blue circles) track those of standard GMRES (the black line) exactly. Finally, Newton-basis CA-GMRES shows different behavior, depending on the basis length s, even though the s-step basis is always numerically full rank. Furthermore, Newton-basis CA-GMRES converges faster than standard GMRES for $s = 10$, and (generally) slower for $s = 30$. This illustrates that the type of basis affects convergence, even if the basis is numerically full rank.

We show the convergence results of the third experiment as Figure 3.9. Here, the matrix is different – as Table 3.2 shows, it is more nonnormal than the matrix used in the previous two experiments. The restart length is 25, and we compare standard GMRES, and CA-GMRES with both the monomial and Newton bases. For CA-GMRES, we show results for $s = 5$ (the largest integer factor of 25 smaller than 25) and $s = 25$. The third experiment affirms the observation of the previous paragraph, that whether the s-step basis is numerically full rank is not the only factor in its convergence. In fact, both for $s = 5$ and $s = 25$, the Newton basis is always numerically full rank, but for $s = 5$ CA-GMRES (the green triangles) converges slower than standard GMRES (the black line) and monomial-basis CA-GMRES (the blue circles), and and for $s = 25$, CA-GMRES (the cyan triangles) does not converge at all. It appears that this poor behavior of the Newton basis relates to the more severe nonnormality of this test problem, though more experiments are needed to verify this observation. We discuss this further in Appendix D.1.

Matrix name	Best s (Monomial basis)	Best s (Newton basis)
bmw7st_1	3	4
Cant_A	4	4
cfd2	3	4
pwtk	5	5
shipsec1	3	3
xenon2	3	3

Table 3.3: Test matrices in [79] for which numerical experiments were performed, and the basis lengths s resulting in the best matrix powers kernel performance on the Intel Clovertown platform (see Appendix A for a description of that test platform) for the monomial and Newton bases.

3.5.4 Sparse matrices from applications

In this section, we demonstrate that CA-GMRES converges for the s-step basis lengths and restart lengths used to obtain maximum performance, on a set of test matrices from applications. These are described in detail in Appendix B.5, and they are the same test problems we used in [79].

Even though our paper [79] concerns only variants of GMRES, an iterative method for nonsymmetric matrices, many (but not all) of the matrices we used there are symmetric positive definite. We did so at the time for reasons of performance and convenience. The matrix powers kernel requires a graph partitioner, and at the time, we had made an arbitrary choice of a graph partitioner which required symmetrization ($A \mapsto (A + A^T)/2$) for nonsymmetric matrices. This may have degraded performance for nonsymmetric matrices. (Future investigations of performance, not in this thesis, will include the use of partitioners more suited for nonsymmetric matrices, such as hypergraph partitioners like Zoltan [81].) Nevertheless, the test matrices are numerically interesting, at least for the matrix powers kernel alone. One of the matrices (bmw7st_1) is badly scaled, which makes equilibration challenging, and one of the matrices (xenon2) is nonsymmetric.

Table B.2 in Section B.5 lists the matrices in this collection for which we include numerical experiments in this thesis. Table 3.3 in this section shows for each matrix, the basis length s for each basis type (monomial or Newton) that resulted in the best matrix powers kernel performance. The value of s for each matrix suggests a reasonable range of basis lengths for which CA-GMRES must converge effectively on that matrix, in order for the method to have optimal performance. We do not include the matrices 1d3pt, 1d5pt, and 2d9pt from Demmel et al. [79], as they represent sparsity patterns from regular discretizations of 2-D PDEs on square domains. This is a set of problems for which we already run more interesting numerical experiments in this thesis (for example, in Section 3.5.3).

The main conclusion to draw from Table 3.3 is that the s-step basis length s need not be very large in order to get the best performance on matrices from applications, at least when running on a single parallel shared-memory node with a small number of processors. For all the experiments presented in this thesis, CA-GMRES usually converges for $s = 5$ or less, even for the monomial basis, and even when the matrix A is ill-conditioned. Thus, the

choice of basis may not matter so much, at least for this particular hardware platform. This conclusion is supported by the numerical results, which as we will discuss below, show that equilibrating the matrix A before running CA-GMRES is often better for numerical stability than the choice of s-step basis.

Many of the matrices in this collection are strongly affected by equilibration. It is important to note that equilibration is a special case of preconditioning, and thus affects the convergence rate of both standard GMRES and CA-GMRES. In exact arithmetic, it should affect the convergence rate in the same way, but this is not necessarily true when computing in finite-precision arithmetic. Regardless, one should not expect the convergence of a single method (such as standard GMRES) to be the same, with and without equilibration. This is true even though we are careful to measure the residual error with respect to the nonequilibrated problem, so that the norm in which we measure the residual error is the same with or without equilibration.

Figures 3.10–3.17 show results with restart length 20 and no equilibration. Figures 3.18–3.25 show results with restart length 20 and equilibration. Figures 3.26–3.33 show results with restart length 60 and no equilibration, and Figures 3.18–3.25 show results with restart length 60 and equilibration. For all the experiments, we show numerical results for basis lengths $s = 5$ and $s = 10$. For almost all of the test problems, $s = 5$ is closer to the basis length best for performance, which is why we show convergence for these relatively small s values. What we found is that for all matrices tested but `bmw7st_1`, the choice of basis does not affect convergence at all, and CA-GMRES converges at the same rate as standard GMRES (excluding a few "blips" for some matrices, such `xenon2`).

For the `bmw7st_1` matrix (see e.g., Figures 3.26 and 3.34 for restart length 60 without resp. with equilibration), we observe an unusual effect. Without equilibration, standard GMRES with the same restart length converges faster than CA-GMRES (with any basis). CA-GMRES with the monomial basis and $s = 10$ utterly fails to converge, and in general CA-GMRES converges better with the Newton basis. However, *with* equilibration, CA-GMRES (with either basis, and either $s = 5$ or $s = 10$) converges at the same rate as standard GMRES. Table B.4 in Appendix B.5 shows that the entries of `bmw7st_1` vary much more in magnitude than the entries of the other matrices, which could explain why this matrix is so much more sensitive to equilibration. This is true even though `bmw7st_1` is symmetric positive definite. Further investigation is needed to determine when equilibration is necessary, and to avoid reductions in the GMRES convergence rate when performing equilibration.

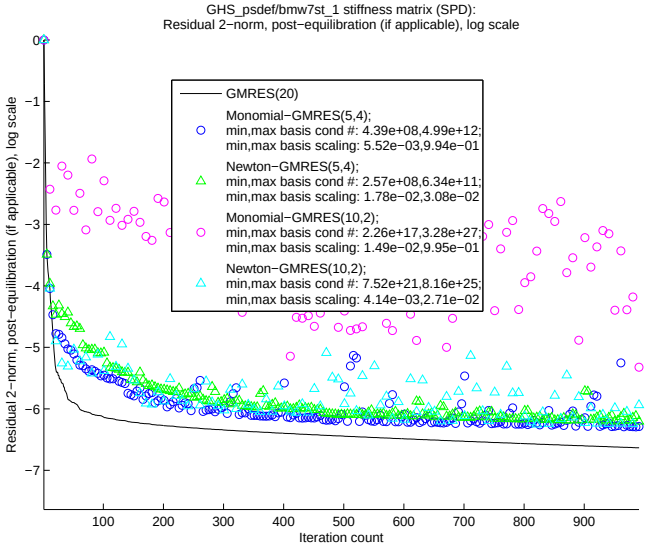


Figure 3.10: Convergence of standard GMRES and CA-GMRES, with restart length 20 and without equilibration, on the matrix "`bmw7st_1`" from applications.

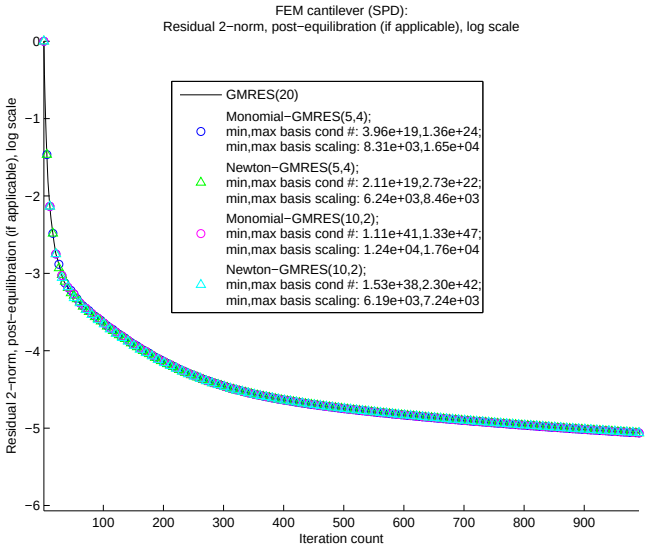


Figure 3.11: Convergence of standard GMRES and CA-GMRES, with restart length 20 and without equilibration, on the matrix "`Cant_A`" from applications.

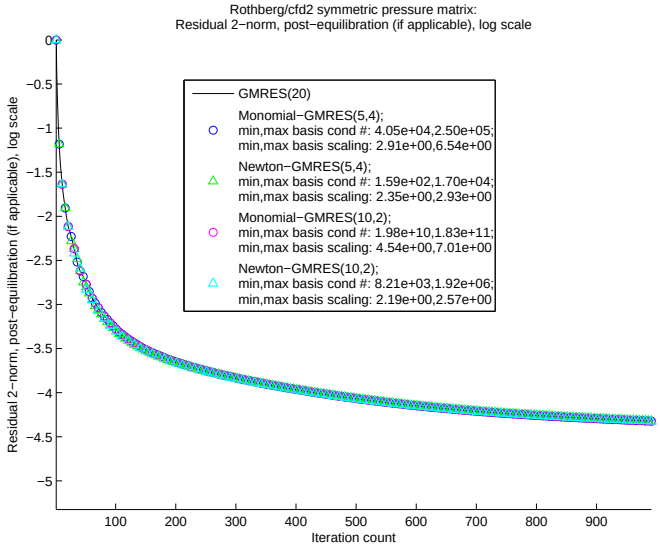


Figure 3.12: Convergence of standard GMRES and CA-GMRES, with restart length 20 and without equilibration, on the matrix "`cfd2`" from applications.

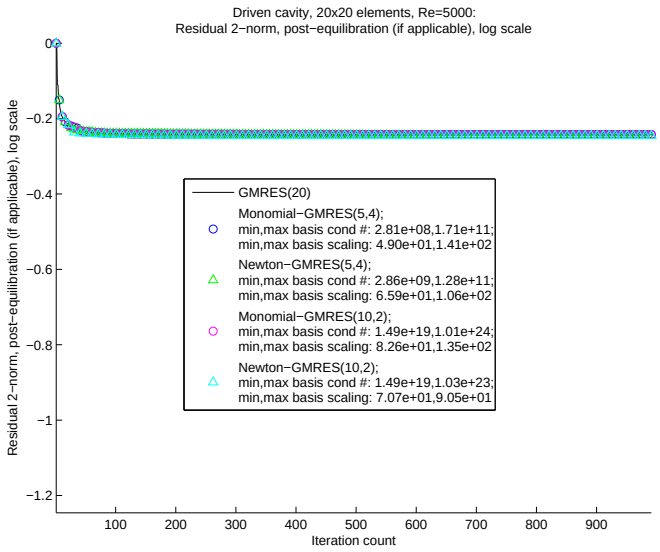


Figure 3.13: Convergence of standard GMRES and CA-GMRES, with restart length 20 and without equilibration, on the matrix "`e20r5000`" from applications.

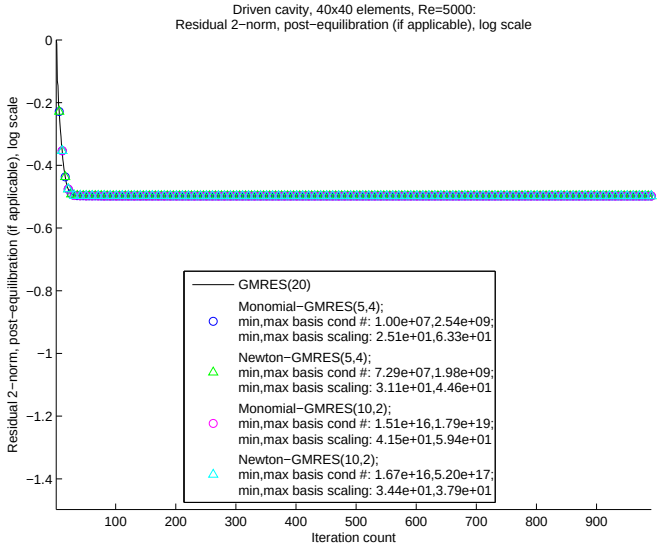


Figure 3.14: Convergence of standard GMRES and CA-GMRES, with restart length 20 and without equilibration, on the matrix "`e40r5000`" from applications.

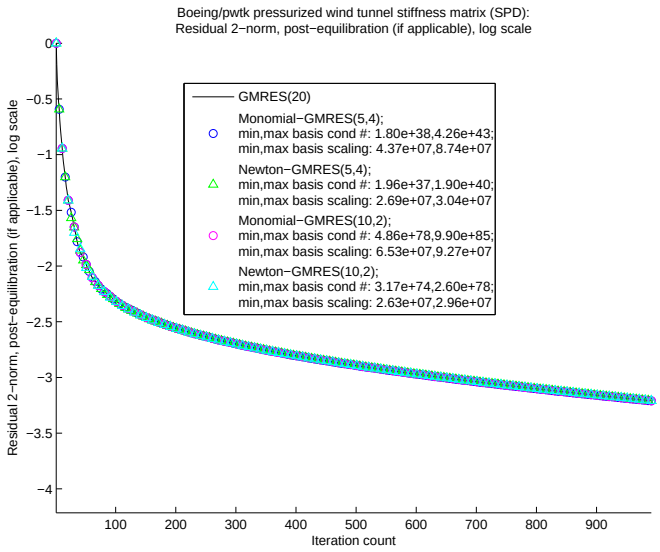


Figure 3.15: Convergence of standard GMRES and CA-GMRES, with restart length 20 and without equilibration, on the matrix "`pwtk`" from applications.

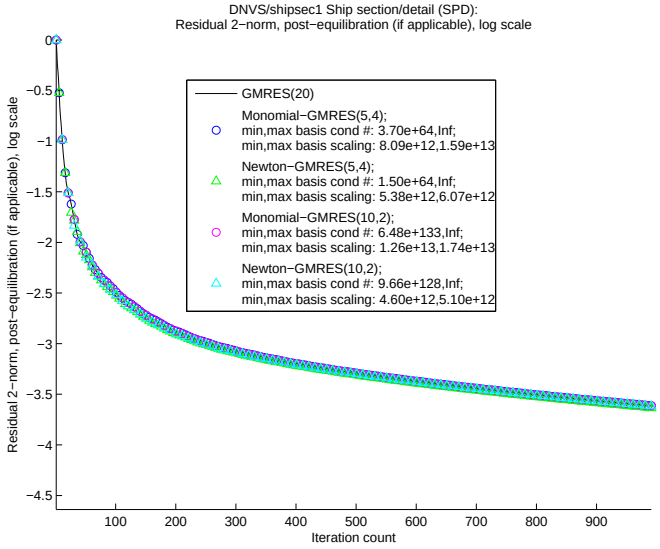


Figure 3.16: Convergence of standard GMRES and CA-GMRES, with restart length 20 and without equilibration, on the matrix "`shipsec1`" from applications.

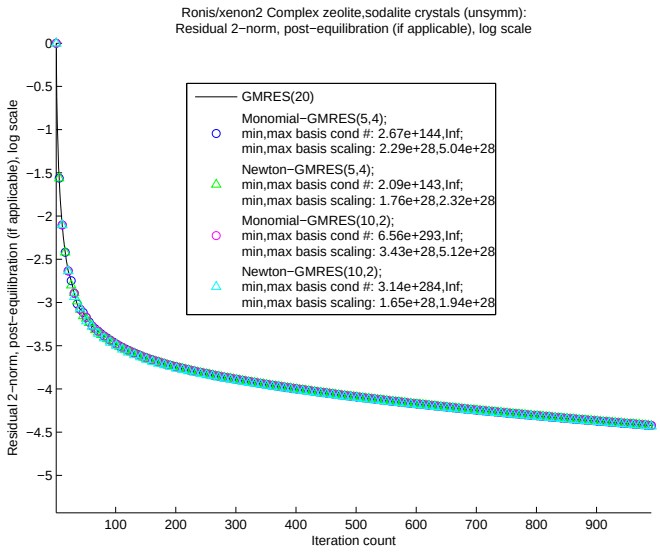


Figure 3.17: Convergence of standard GMRES and CA-GMRES, with restart length 20 and without equilibration, on the matrix "`xenon2`" from applications.

Algorithm 23 CA-GMRES

Require: $n \times n$ linear system $Ax = b$, and initial guess x_0

1: $r_0 := b - Ax_0$, $\beta := \|r_0\|_0$, $q_1 := r_0/\beta$
2: **for** $k = 0$ to $t - 1$ **do**
3: Fix basis conversion matrix $\underline{B}_k$
4: Compute $\underline{\acute{V}}_k$ using matrix powers kernel
5: **if** $k = 0$ **then**
6: Compute QR factorization $\underline{V}_0 = \underline{Q}_0 \underline{R}$
7: Set $\underline{\mathfrak{Q}}_0 := \underline{Q}_0$ and $\underline{\mathfrak{H}}_0 := \underline{R}_0 \underline{B}_0 R_0^{-1}$
8: Reduce $\underline{H_0}$ from upper Hessenberg to upper triangular form using s Givens rotations G_1, G_2, ..., G_s. Apply the same rotations in the same order to βe_1, resulting in the length $s + 1$ vector ζ_0.
9: **else**
10: $\underline{\acute{\mathfrak{R}}}_{k-1,k} := \underline{\mathfrak{Q}}_{k-1}^* \underline{\acute{V}}_k$
11: $\underline{\acute{V}}_k' := \underline{\acute{V}}_k - \underline{\mathfrak{Q}}_{k-1} \underline{\acute{\mathfrak{R}}}_{k-1,k}$
12: Compute QR factorization $\underline{\acute{V}}_k' = \underline{\acute{Q}}_k \underline{\acute{R}}_k$
13: Compute $\underline{\mathfrak{H}}_{k-1,1} := -\mathfrak{H}_{k-1} \mathfrak{R}_{k-1,k} R_k^{-1} + \underline{\mathfrak{R}}_{k-1,k} \underline{B}_k R_k^{-1}$
14: Compute H_k via Equation (3.21)
15: Compute h_k via Equation (3.22)
16: $\underline{\mathfrak{H}}_k := \begin{pmatrix} \mathfrak{H}_{k-1} & \underline{\mathfrak{H}}_{k-1,k} \\ h_{k-1} e_1 e_{s(k-1)}^T & H_k \\ 0_{1,s(k-1)} & h_k e_s^T \end{pmatrix}$
17: Apply Givens rotations G_1, ..., G_{sk} in order to $\begin{pmatrix} \underline{\mathfrak{H}}_{k-1,k} \\ \underline{H}_k \end{pmatrix}$.
18: Reduce $\underline{H}_k$ to upper triangular form using s Givens rotations G_{sk+1}, ..., $G_{s(k+1)}$. Apply the rotations in the same order to $\begin{pmatrix} \zeta_{k-1} \\ 0_s \end{pmatrix}$, resulting in the length $s(k + 1) + 1$ vector ζ_k.
19: **end if**
20: Element $s(k + 1) + 1$ of ζ_k is the 2-norm (in exact arithmetic) of the current residual $r_{k+1} = b - Ax_{k+1}$ of the current solution x_{k+1}.
21: **if** converged **then**
22: Use the above reduction of $\underline{\mathfrak{H}}_k$ to upper triangular form, and ζ_k, to solve $y_k := \operatorname{argmin}_y \|\underline{\mathfrak{H}}_k y - \beta e_1\|_2$
23: Set $x_k := x_0 + \mathfrak{Q}_k y_k$, and exit
24: **end if**
25: **end for**

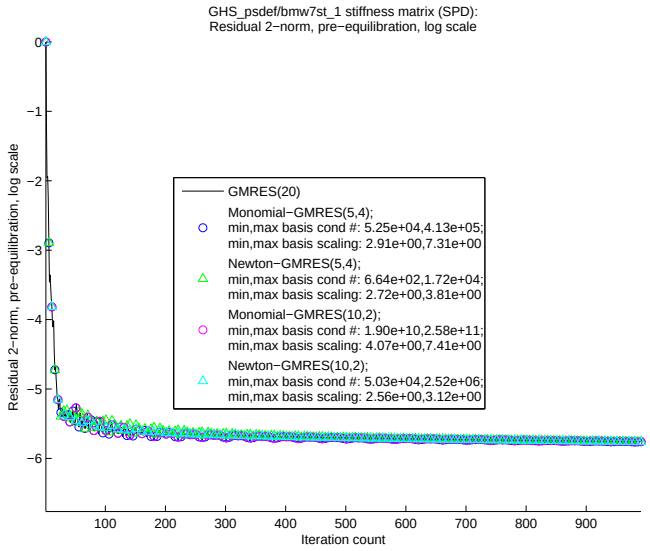


Figure 3.18: Convergence of standard GMRES and CA-GMRES, with restart length 20 and with equilibration, on the matrix "`bmw7st_1`" from applications.

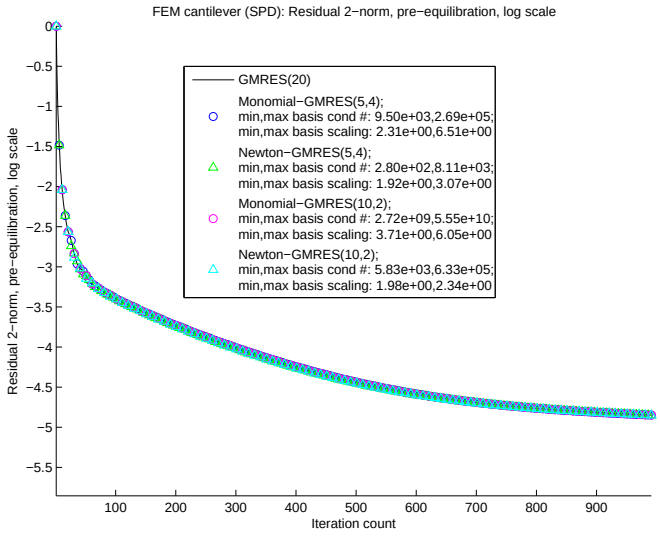


Figure 3.19: Convergence of standard GMRES and CA-GMRES, with restart length 20 and with equilibration, on the matrix "`Cant_A`" from applications.

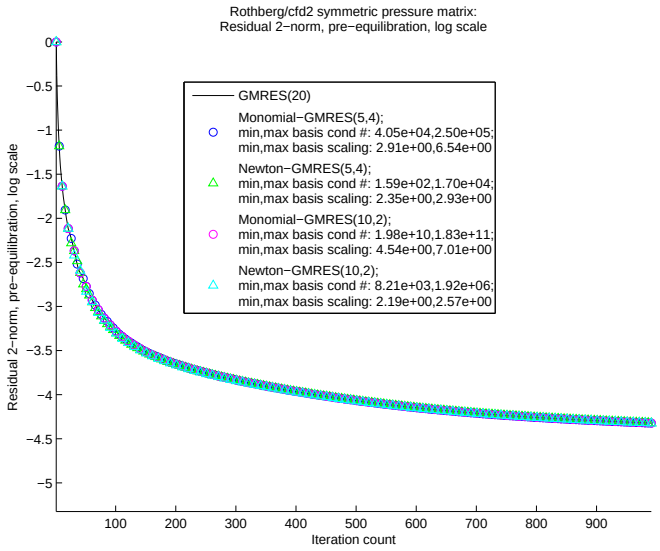


Figure 3.20: Convergence of standard GMRES and CA-GMRES, with restart length 20 and with equilibration, on the matrix "`cfd2`" from applications.

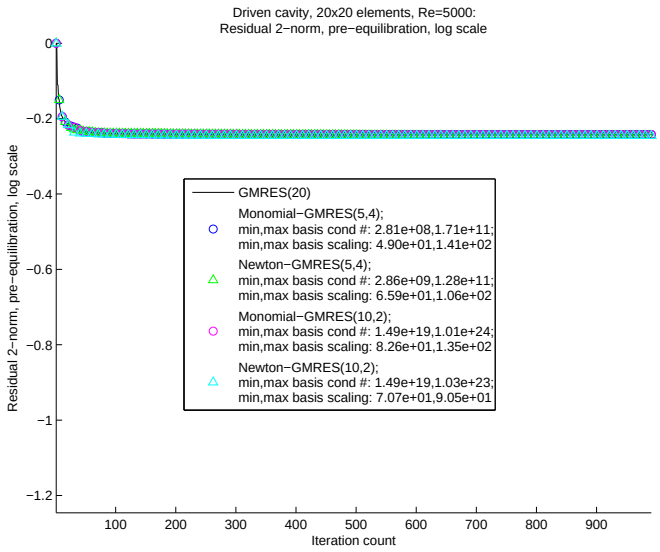


Figure 3.21: Convergence of standard GMRES and CA-GMRES, with restart length 20 and with equilibration, on the matrix "`e20r5000`" from applications.

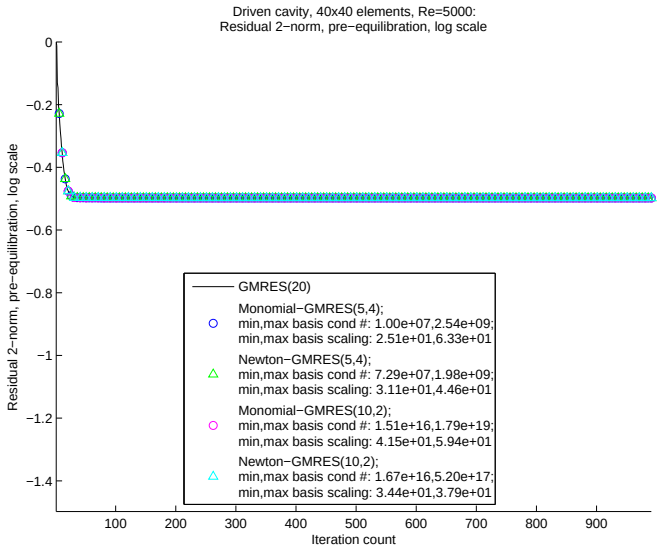


Figure 3.22: Convergence of standard GMRES and CA-GMRES, with restart length 20 and with equilibration, on the matrix "`e40r5000`" from applications.

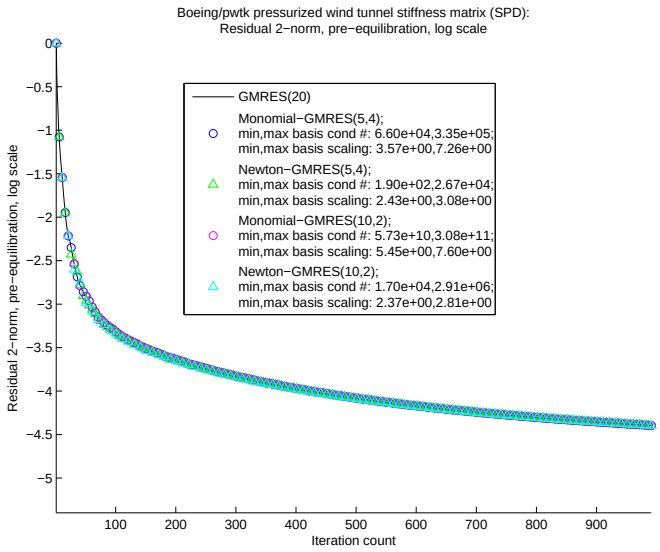


Figure 3.23: Convergence of standard GMRES and CA-GMRES, with restart length 20 and with equilibration, on the matrix "`pwtk`" from applications.

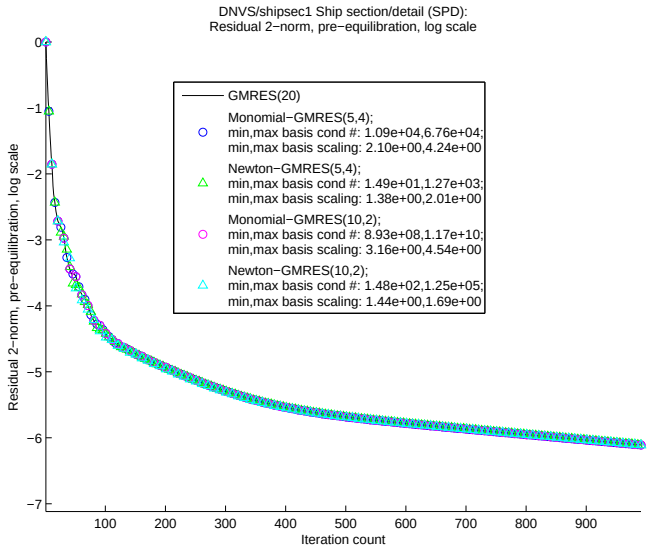


Figure 3.24: Convergence of standard GMRES and CA-GMRES, with restart length 20 and with equilibration, on the matrix "`shipsec1`" from applications.

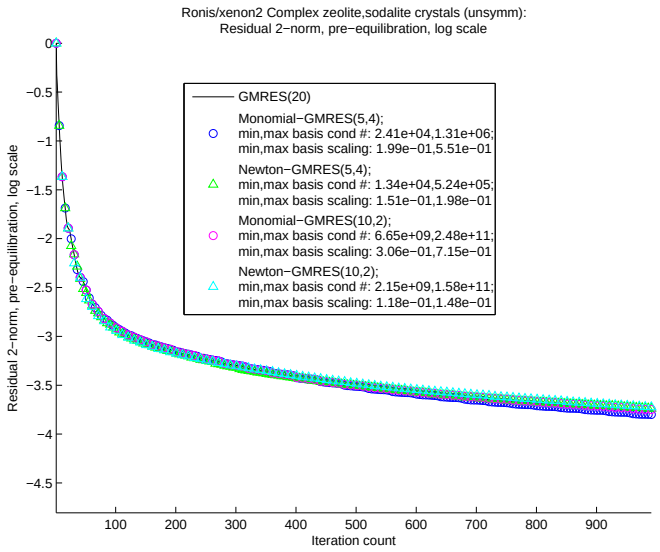


Figure 3.25: Convergence of standard GMRES and CA-GMRES, with restart length 20 and with equilibration, on the matrix "`xenon2`" from applications.

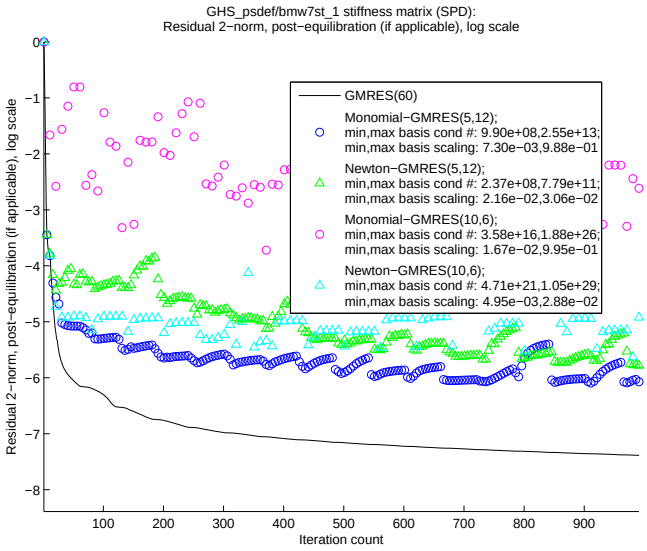


Figure 3.26: Convergence of standard GMRES and CA-GMRES, with restart length 60 and without equilibration, on the matrix "`bmw7st_1`" from applications.

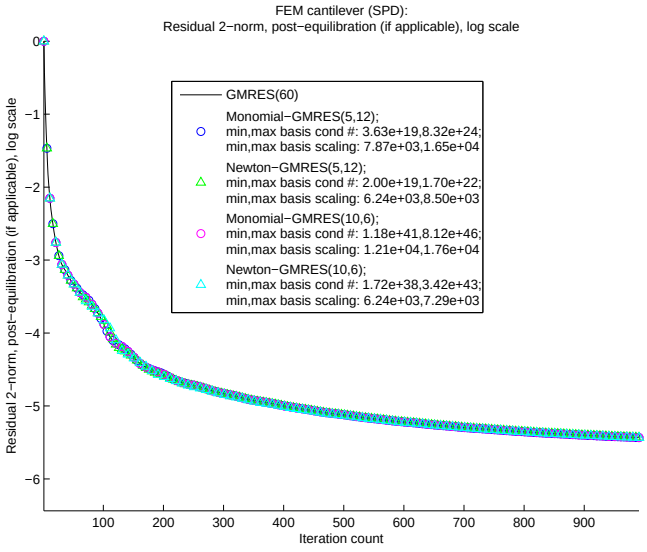


Figure 3.27: Convergence of standard GMRES and CA-GMRES, with restart length 60 and without equilibration, on the matrix "`Cant_A`" from applications.

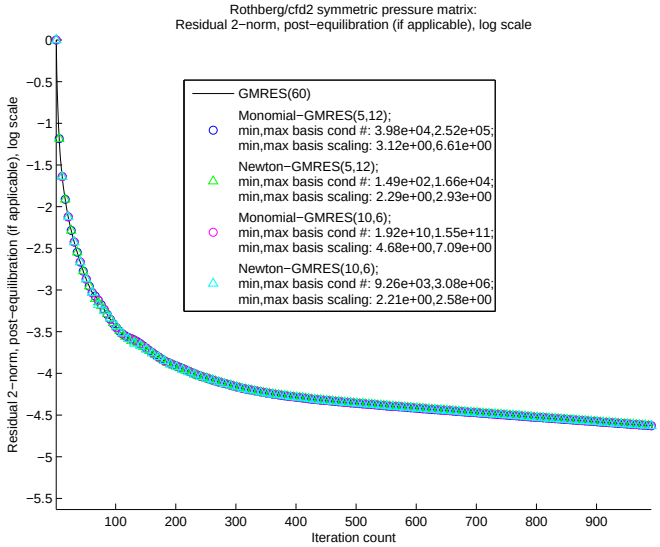


Figure 3.28: Convergence of standard GMRES and CA-GMRES, with restart length 60 and without equilibration, on the matrix "`cfd2`" from applications.

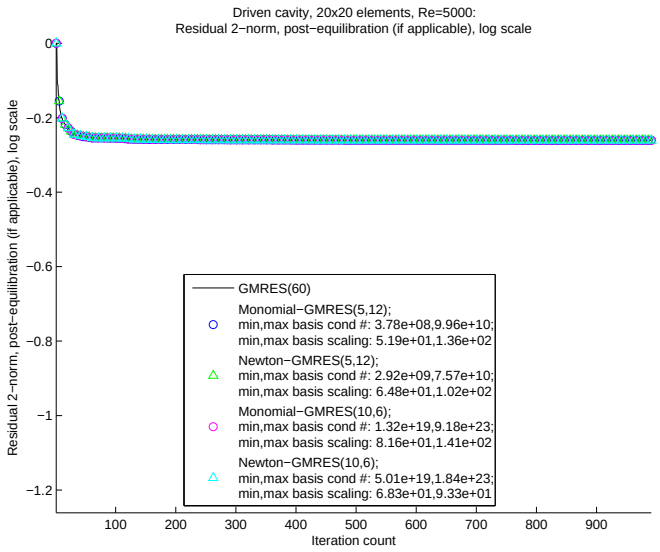


Figure 3.29: Convergence of standard GMRES and CA-GMRES, with restart length 60 and without equilibration, on the matrix "`e20r5000`" from applications.

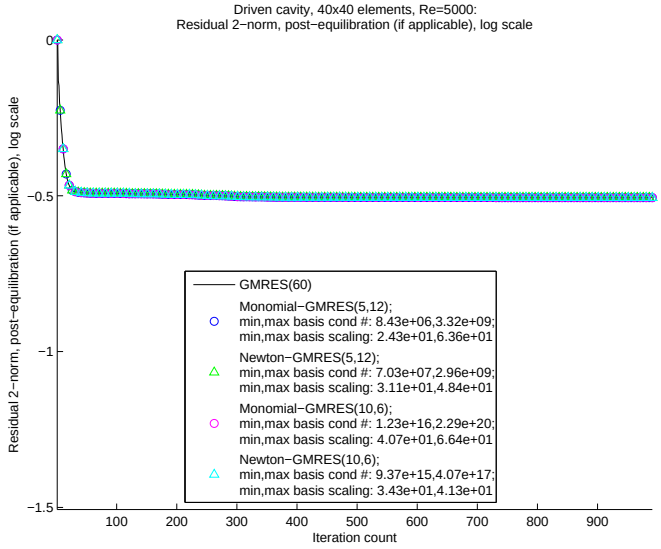


Figure 3.30: Convergence of standard GMRES and CA-GMRES, with restart length 60 and without equilibration, on the matrix "`e40r5000`" from applications.

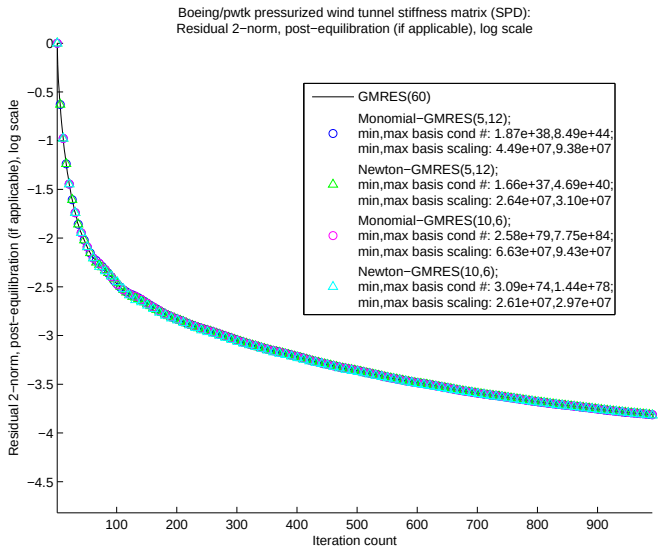


Figure 3.31: Convergence of standard GMRES and CA-GMRES, with restart length 60 and without equilibration, on the matrix "`pwtk`" from applications.

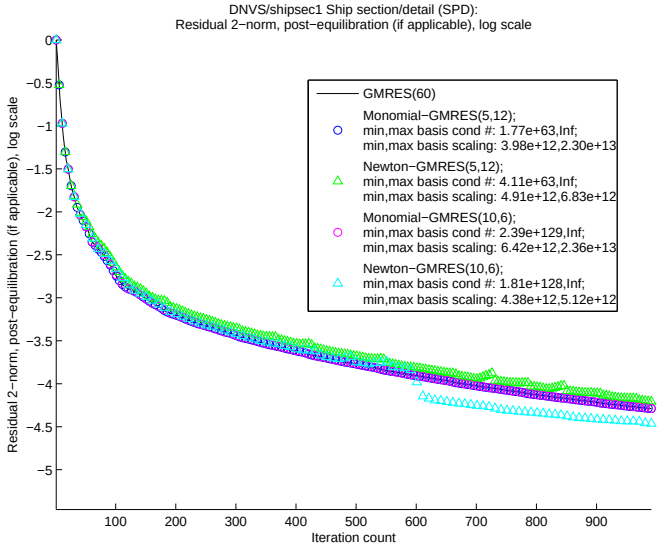


Figure 3.32: Convergence of standard GMRES and CA-GMRES, with restart length 60 and without equilibration, on the matrix "`shipsec1`" from applications.

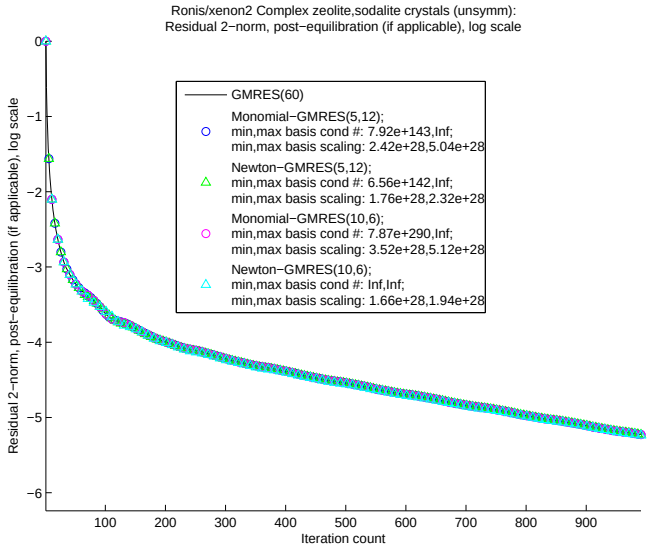


Figure 3.33: Convergence of standard GMRES and CA-GMRES, with restart length 60 and without equilibration, on the matrix "`xenon2`" from applications.

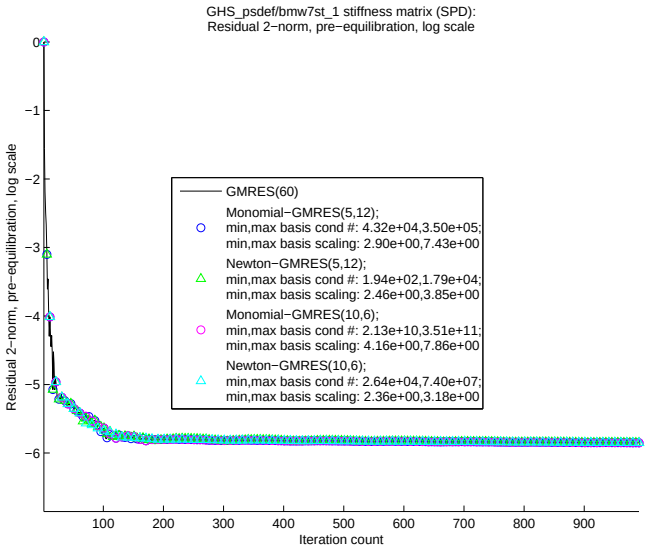


Figure 3.34: Convergence of standard GMRES and CA-GMRES, with restart length 60 and with equilibration, on the matrix "`bmw7st_1`" from applications.

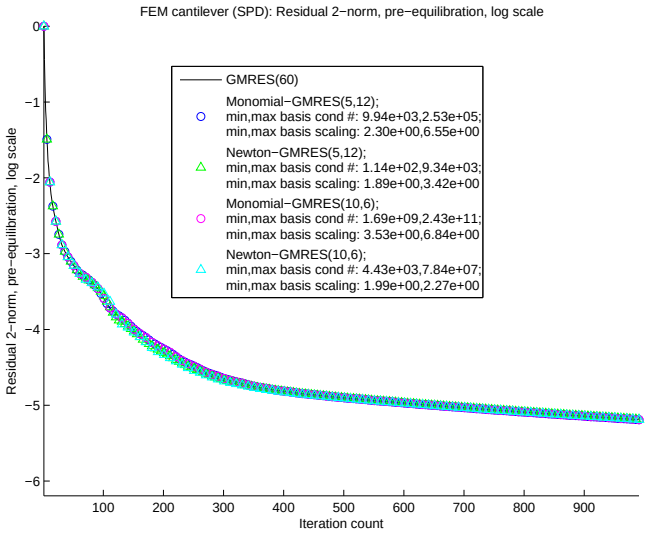


Figure 3.35: Convergence of standard GMRES and CA-GMRES, with restart length 60 and with equilibration, on the matrix "`Cant_A`" from applications.

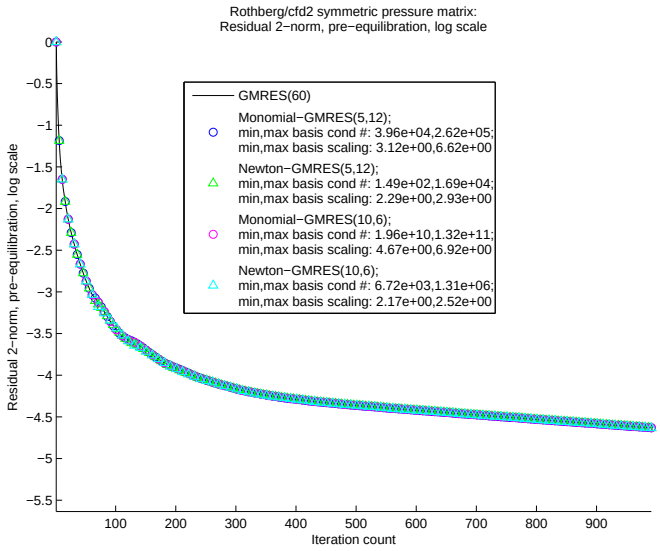


Figure 3.36: Convergence of standard GMRES and CA-GMRES, with restart length 60 and with equilibration, on the matrix "`cfd2`" from applications.

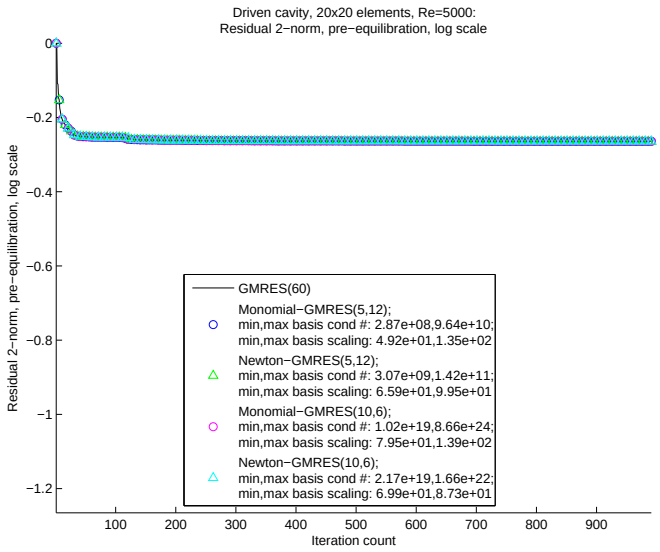


Figure 3.37: Convergence of standard GMRES and CA-GMRES, with restart length 60 and with equilibration, on the matrix "`e20r5000`" from applications.

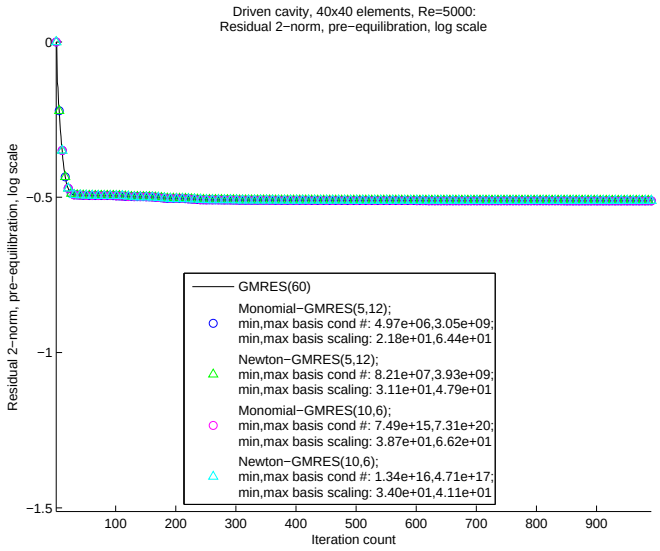


Figure 3.38: Convergence of standard GMRES and CA-GMRES, with restart length 60 and with equilibration, on the matrix "`e40r5000`" from applications.

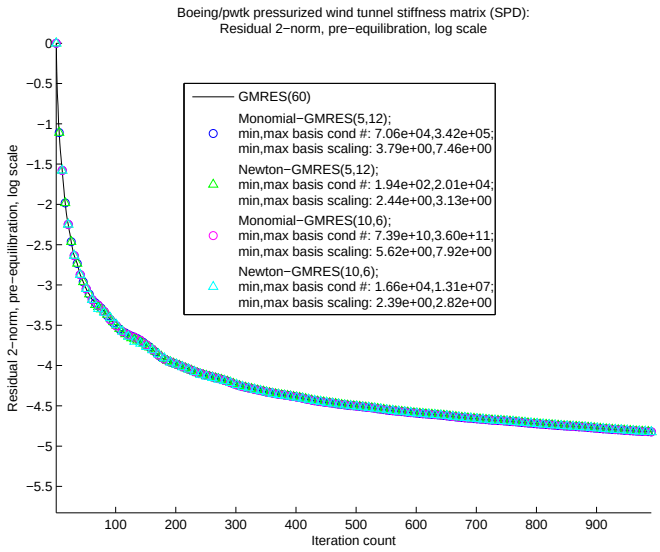


Figure 3.39: Convergence of standard GMRES and CA-GMRES, with restart length 60 and with equilibration, on the matrix "`pwtk`" from applications.

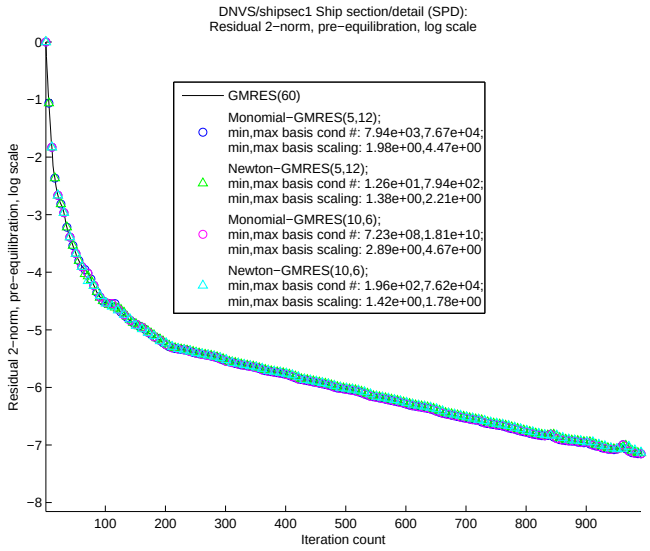


Figure 3.40: Convergence of standard GMRES and CA-GMRES, with restart length 60 and with equilibration, on the matrix "`shipsec1`" from applications.

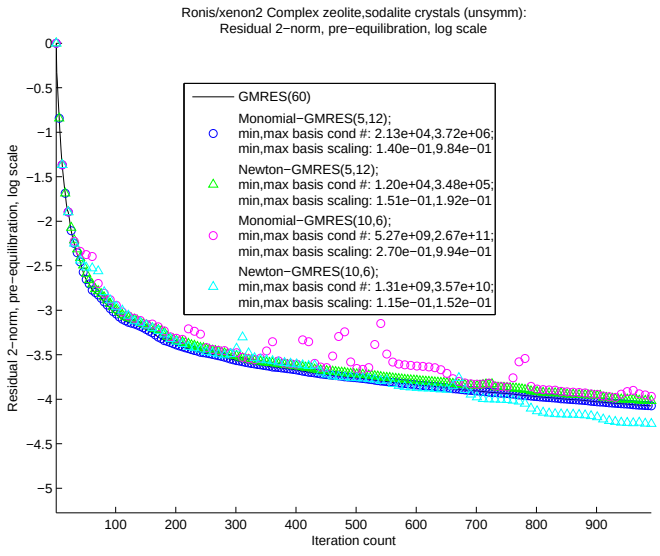


Figure 3.41: Convergence of standard GMRES and CA-GMRES, with restart length 60 and with equilibration, on the matrix "`xenon2`" from applications.

3.5.5 WATT1 test problem

In this section, we compare the convergence of our CA-GMRES on a particular nonsymmetric matrix, WATT1, with the convergence on the same matrix of the Newton-basis s-step GMRES algorithm by Bai et al. [19]. Erhel [94] used this test problem to test her implementation of the algorithm of Bai and his coauthors. (Erhel's implementation differs only in that she uses a more efficient, but equally accurate, parallel QR factorization.) The ability of our CA-GMRES to adjust the s-step basis length independently of the restart length helps us achieve a qualitatively similar convergence rate as Erhel observes for the algorithm of Bai et al. Erhel must compute $2s$ start-up iterations of standard GMRES in order to get s sufficiently good shifts for the Newton basis; we simply choose s smaller to avoid this problem. Performance results for this matrix in Section 3.6.3 show that being able to choose the s-step basis length to be smaller than the restart length improves performance. Of course, we do not want to be forced to choose s too small, as that restricts the potential performance improvements of the matrix powers kernel (see Section 2.1). Equilibration provides the necessary numerical stability improvement to increase the range of possible s values.

The origin and properties of the WATT1 matrix is described in detail in Appendix B.6. Erhel [94] used the first matrix WATT1 in the WATT set as a test problem for her implementation of the Newton-basis s-step GMRES of Bai et al. [19]. Her paper includes plots of the 2-norm of the residual error at each iteration of the method, for standard GMRES (with restart length s), monomial-basis s-step GMRES, and Newton-basis s-step GMRES. An "iteration" in her convergence plots refers to a single restart cycle, and in her case corresponds to s iterations of standard GMRES. Unfortunately, Erhel does not mention what right-hand side or initial guess she uses for her experiments, so we cannot compare her results directly with ours. However, we can make a qualitative comparison.

Erhel's implementation of GMRES uses different computational kernels than ours, as we have mentioned before. The numerical properties of her algorithm are also different, because the Newton-basis s-step GMRES algorithm of Bai et al. [19] that Erhel implements requires that the basis length s be equal to the restart length. Our CA-GMRES has restart length $s*t$, where the basis length s may in general be less than the restart length. This generalization is useful both for performance (as sometimes a smaller s performs better) and for accuracy (as large values of s often result in an ill-conditioned or numerically rank-deficient s-step basis).

Figures 3.42 and 3.42 show convergence results without resp. with equilibration, for standard GMRES and CA-GMRES with restart length $r = 30$ on the WATT1 matrix. For each plot, we show CA-GMRES with two values of s:

- the largest s value for which CA-GMRES has convergence behavior similar to standard GMRES for some basis, and
- the smallest s value for which CA-GMRES does not converge well for any basis.

The largest "good" s value differs, depending on whether we perform equilibration on the matrix. Without equilibration, this is $s = 3$, but with equilibration, this is $s = 5$. This shows that equilibration helps convergence with larger values of s, although the effect is not large. The smallest "bad" value also differs: with equilibration, the Newton basis can

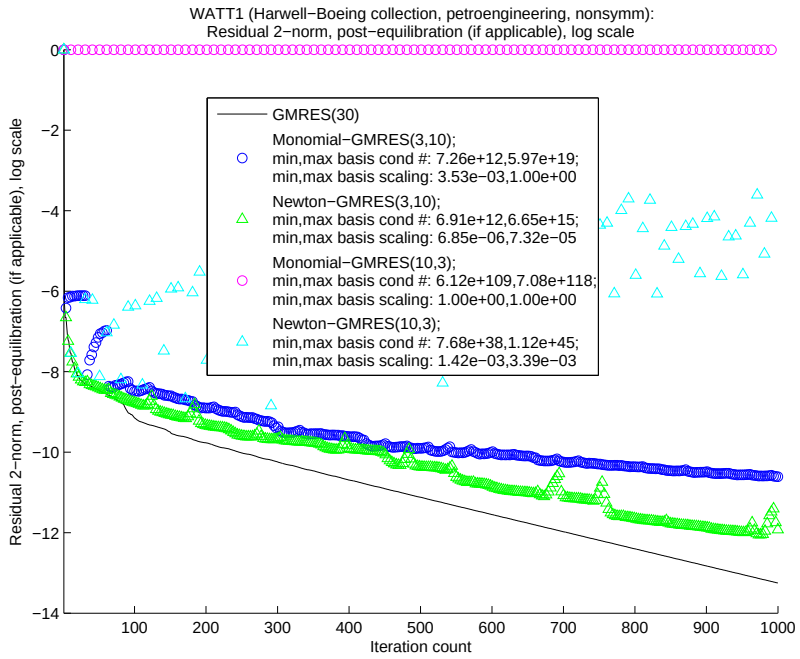


Figure 3.42: Convergence of standard GMRES and CA-GMRES, with restart length 30 and without equilibration, on the WATT1 matrix. Two s values shown: $s = 3$ (largest with some convergence success) and $s = 10$ (smallest with little or no convergence success).

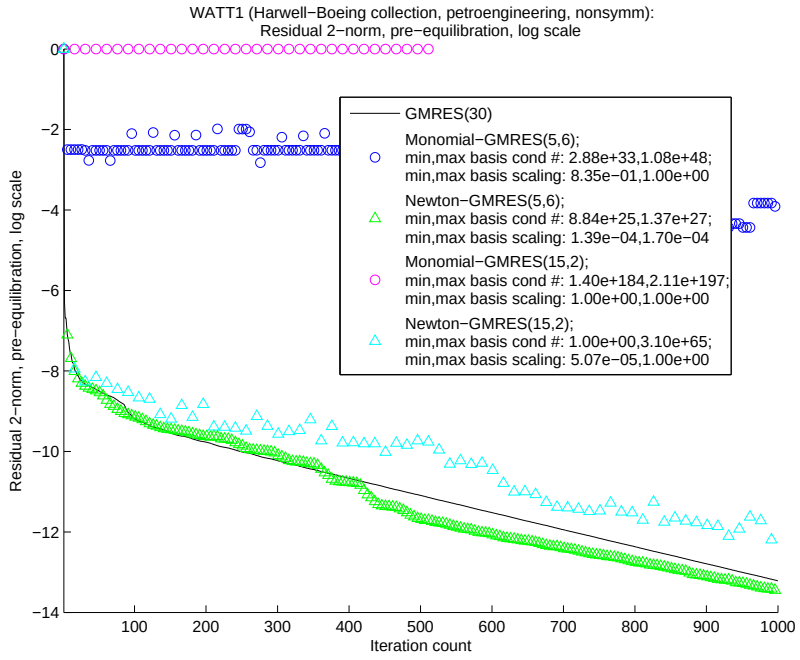


Figure 3.43: Convergence of standard GMRES and CA-GMRES, with restart length 30 and with equilibration, on the WATT1 matrix. Two s values shown: $s = 5$ (largest with some convergence success) and $s = 15$ (smallest with little or no convergence success).

converge similarly to standard GMRES with $s = 15$, but without equilibration, the Newton basis converges with s at most 10.

Erhel was aware that large basis lengths s could result in an ill-conditioned s-step basis. She dealt with this by improving the accuracy of the Ritz values used as shifts in the Newton basis. Rather than performing s iterations of standard GMRES as a preprocessing step to get s Ritz values, she performed $2s$ iterations of standard GMRES to get $2s$ Ritz values. The modified Leja ordering process then selects s of these as shifts for the Newton basis. For $s = 30$, the basis length that Erhel used in her experiments, collecting 60 Ritz values rather than 30 made Newton-basis GMRES converge with the WATT1 matrix, when otherwise neither the Newton basis nor the monomial basis resulted in convergence. This, however, increases the startup cost of the algorithm. We chose instead to solve this problem by reducing the size of s, which when combined with equilibration, gives us both good convergence and good performance (see Section 3.6.3).

We also found that subtle features of the modified Leja ordering computation (see Section 7.3.2 and Algorithm 40) were necessary to avoid failure of the ordering. In particular, for many values of s, both with and without equilibration, the product of absolute differences of shifts maximized in the modified Leja ordering algorithm (Line 16 of Algorithm 40) often was zero or very small. This triggered a cycle of random perturbations (see Line 18 of the algorithm). We set the perturbations so as to perturb each shift in a uniform random way by a relative perturbation of 10^{-3+k}, where k is the current random perturbation cycle number (starts at $k = 1$ and increases by one each time the product in Line 16 underflows). One perturbation cycle was generally enough to avoid underflow in the product to maximize. This solution to the underflow problem was used Bai et al. [19] in their Newton GMRES algorithm; Erhel does not mention it, although it seems necessary in the case of the WATT1 matrix. Note that the underflow problem is separate from the case of repeated shifts; we already have a method of dealing with those (which we found in Reichel [202] but not in Bai et al., who seem to rely only on random perturbations to deal with the problem).

3.5.6 Summary

In this section, we showed that CA-GMRES converges like standard GMRES for restart lengths of interest on many types of problems, including those with badly scaled and / or ill-conditioned nonsymmetric matrices. CA-GMRES can do so if we do the following:

- Pick the basis length s according to the difficulty of the problem (this might be done dynamically in a practical implementation)
- Use the right s-step basis for the problem (usually the Newton basis is the right choice)
- Equilibrate the problem first, to avoid a badly scaled basis

The ability of CA-GMRES to choose the basis length s shorter than the restart length $r = s \cdot t$ helps convergence in many cases. It also helps performance, as we will see in the next section.

3.6 CA-GMRES performance experiments

In this section, we describe performance experiments that compare our shared-memory parallel implementation of CA-GMRES (Section 3.4, Algorithm 23) with standard GMRES, on the experimental platforms described in Appendix A. CA-GMRES achieves a speedup of 1.31 to 4.3× over standard GMRES on the Intel Clovertown platform on the set of test problems in this section, and a speedup of 1.3 to 4.1× on the Intel Nehalem platform. Both CA-GMRES and standard GMRES in this case are running on 8 cores of the platform.

The main set of test problems in this section are sparse matrices from various applications, described in Appendix B.5, for which we showed in Section 3.5.4 that CA-GMRES converges just as well as standard GMRES. We first presented performance results for these problems in Demmel et al. [79]; there we only include results for the Intel Clovertown platform, whereas here we also include Intel Nehalem results. Details on the matrix, right-hand side, and initial guess used for each experiment are found in Appendix B. We also include performance results for the WATT1 matrix, which we describe in Appendix B.6 and for which we show numerical experiments in Section 3.5.5.

We begin in Section 3.6.1 by describing the details of our shared-memory parallel implementations of standard GMRES and CA-GMRES. Section 3.6.2 shows the results of our experiments for the sparse matrices from applications, and Section 3.6.3 the results for the WATT1 matrix. Finally, Section 3.6.5 discusses future work.

3.6.1 Implementation details

Parallel primitives

We implemented parallelism in both standard GMRES and CA-GMRES using the POSIX standard Pthreads library, with a custom low-overhead barrier implementation. We programmed in a single-program multiple-data style, in which each thread has a copy of its own "main" function, and synchronize with each other using barriers and collective operations. Threads were only created once, at the beginning of the benchmark, and destroyed once, at the end of the benchmark.

Our implementation of standard GMRES required parallel reductions to compute vector norms and inner products. We implemented reductions with a combination of the low-overhead barriers and an array of per-thread local reduction results state, which one thread was responsible for reducing to a single scalar. The array of shared state was padded to avoid false sharing.

Third-party software libraries

We build our benchmark using Goto's BLAS (version 1.26). See e.g., Goto and van de Geijn [118]. This version of the BLAS uses OpenMP (see [34]) internally for parallelism. We disabled this feature, however, when calling the BLAS from parallel TSQR, to avoid harmful interactions between the two levels of parallelism. (Future work may explore using two levels of shared-memory parallelism, so that parallel TSQR calls a parallel BLAS. For an example of a parallel runtime system that enables correct and efficient use of shared-memory parallel libraries calling other shared-memory parallel libraries, see the "Lithe" system of Pan et al.

[188]. Their benchmark application is a sparse QR factorization that exploits parallelism both in the sparse matrix's elimination, and in the dense updates (via a parallel BLAS).) Our benchmark requires at least version 3.2 of LAPACK, in order to ensure that `DGEQRF` produces an R factor with a nonnegative real diagonal (see Section 3.2.4 for a full discussion of this issue). For partitioning the sparse matrix for the matrix powers kernel, we used Metis (version 4.0) (see Karypis and Kumar [149]).

Thread and memory affinity

Our benchmark implementation pins threads via the GNU / Linux `sched_setaffinity` system call, in order to ensure that at most one thread is associated to each core. (We found no benefit from using the SMT feature of the Nehalem processor, so we do not show results from intentionally scheduling more than one thread per core.) Even though the Clovertown platform does not have NUMA memory, our implementation of TSQR and BGS uses the first-touch policy, which in GNU / Linux (and in some but not all operating systems) ensures that the memory in question is made local (in a NUMA sense) to the thread that initialized it. On the Nehalem platform, our implementation of the matrix powers kernel uses the `libnuma` library to set the affinity of sparse matrix data in a favorable way. (It is only an artifact of the implementation that different kernels set memory affinity in different ways.)

Compiler

We build our benchmark with the GNU C and Fortran compilers (version 4.3). That version or later was necessary, since only in version 4.3 was support for Fortran 2003's `ISO_C_BINDING` module introduced. Our benchmark uses the `ISO_C_BINDING` module to ease communication between Fortran and C code. We encountered some difficulties when using `ISO_C_BINDING` with the then-current version of the Intel Fortran Compiler. That module appeared to interact with multithreading in an unexpected way, but we did not have time to determine whether that was due to a compiler bug.

3.6.2 Results for various sparse matrices from applications

Table B.2 in Section B.5 lists the matrices in this collection for which we include numerical and performance experiments in this thesis. For our experiments, we ran both standard GMRES and CA-GMRES with restart length $r = 60$. Our CA-GMRES experiments searched over all basis lengths $s \geq 2$ that divide r evenly.[5] While the CA-GMRES algorithm itself does not require this (it can even use a different basis length in each outer iteration), our matrix powers kernel implementation did. In some cases, the matrix powers kernel was not able to process the matrix for s sufficiently large, so we excluded those cases. CA-GMRES performance is shown in all cases for the best choice of s, given a particular restart length r. We ran our experiments for thread counts 1, 2, 4, and 8.

Table 3.4 in this section shows for each of the test matrices in this section, the basis length s for each basis type (monomial or Newton) that resulted in the best matrix powers kernel performance. The value of s for each matrix suggests a reasonable range of basis lengths

[5] We chose $r = 60$ for the same reason that the Babylonians chose the hour to have 60 minutes.

Matrix name	Best s (Monomial basis)	Best s (Newton basis)
`1d3pt`	15	15
`bmw7st_1`	3	4
`Cant_A`	4	4
`cfd2`	3	4
`pwtk`	5	5
`shipsec1`	3	3
`xenon2`	3	3

Table 3.4: Test matrices from applications for which performance results are presented in this section, and the basis length s for each resulting in the best matrix powers kernel performance for the monomial and Newton bases.

for which CA-GMRES must converge effectively on that matrix, in order for the method to have optimal performance. We do not include the matrices `1d3pt`, `1d5pt`, and `2d9pt` from Demmel et al. [79], as they represent sparsity patterns from regular discretizations of 2-D PDEs on square domains. This is a set of problems for which we already run more interesting numerical experiments in this thesis (for example, in Section 3.5.3).

3.6.3 Results for WATT1 matrix

In Section 3.5.5, we showed the results of numerical experiments with the WATT1 sparse matrix from a petroleum engineering application. This matrix is described in Appendix B.6. We tested the performance of CA-GMRES on this matrix on the Intel Clovertown platform (see Appendix A), and achieved a speedup over standard GMRES of 1.31×. This was attained when running on two cores, since neither algorithm benefited from more parallelism than that on this platform. Furthermore, being able to choose the s-step basis length to be smaller than the restart length, which previous versions of s-step GMRES could not do, meant that CA-GMRES improved its performance by a factor of 2.23× when running on one core, and 1.66× on two cores. The best s value is also different for different numbers of processors, which illustrates the need for some combination of automatic tuning and performance models to choose the best value.

As we mention in Section 3.5.5 and Appendix B.6, Erhel [94] uses WATT1 as a test problem for her implementation of the Newton-basis GMRES algorithm of Bai et al. [19]. We cannot compare our implementation directly with Erhel's, because hers was intended to run on early 1990's distributed-memory parallel hardware, whereas ours targets the single-node shared-memory regime. For example, the parallel QR factorization discussed in her paper, `RODDEC`, optimizes the number of messages on a ring network. TSQR (Section 2.3), in contrast, minimizes the number of messages in parallel on a general network of processors, as well as minimizing the volume of data movement between levels of the memory hierarchy. Furthermore, Erhel did not show performance results for the WATT1 matrix in [94]. What we can show, however, is that our algorithmic innovations result in performance improvements.

We have performance results for the Intel Clovertown platform (described in Appendix A). We tuned over a variety of TSQR cache block sizes and matrix powers kernel options.

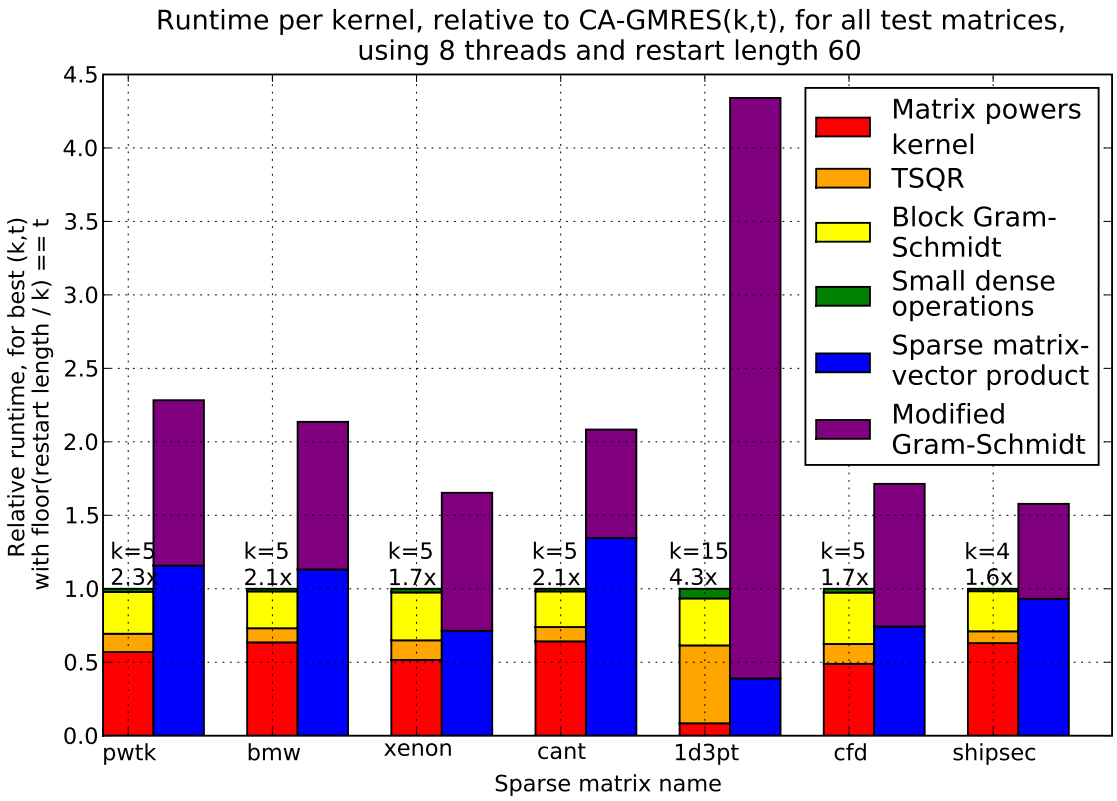


Figure 3.44: Run time in seconds of standard GMRES and CA-GMRES on 8 cores of an Intel Clovertown processor, with restart length 60, on the set of test problems from applications described in Appendix B.5.

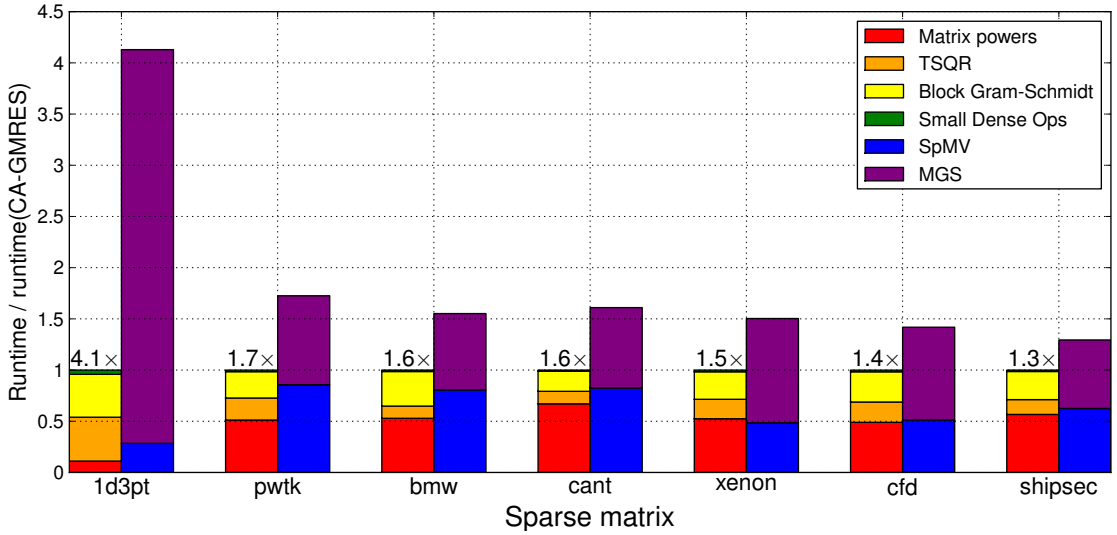


Figure 3.45: Run time in seconds of standard GMRES and CA-GMRES on 8 cores of an Intel Nehalem processor, with restart length 60, on the set of test problems from applications described in Appendix B.5.

For a given set of parameters, each timing run was repeated 100 times, to avoid variation in the reported run times. We fixed the restart length to be $r = 30$, as Erhel used in her experiments, and searched for the best CA-GMRES performance over all basis lengths $s \geq 2$ that divide r evenly. CA-GMRES performance is shown in all cases for the best choice of s, given a particular restart length r. We ran our experiments for thread counts 1, 2, 4, and 8.

Figure 3.46 compares the runtimes of CA-GMRES and standard GMRES on the WATT1 matrix, for different numbers of processors. The best speedup of CA-GMRES over standard GMRES is 1.31×, when running on 2 processor cores. In fact, neither algorithm obtains a total performance benefit from using more than two processor cores. Since the sparse matrix and vectors are small enough to fit in the L2 cache on this platform, the main reason why CA-GMRES is faster than standard GMRES is the use of TSQR and BGS instead of MGS orthogonalization. Our ability in CA-GMRES to choose the s-step basis length s to be less than the restart length $r = s \cdot t$, means that CA-GMRES executes a restart cycle 2.23× faster on one core and 1.66× faster on two cores, than the Newton-basis GMRES algorithm of Bai et al. which requires that s equal the restart length r. Furthermore, this happens for different values of s for different numbers of processors: $s = 5$ is the fastest choice when running on one core, whereas $s = 6$ is the fastest when running on two cores. Although neither CA-GMRES and standard GMRES benefit from more parallelism than this, it is interesting to note that $s = 15$ is the fastest choice of basis length for CA-GMRES on 4 and 8 processors.

These results suggest that for small sparse matrices that fit in cache, that implementing the matrix powers kernel straightforwardly, that is by using s invocations of SpMV, may be faster than using any of the matrix powers kernel optimizations we discuss in Section 2.1. This was the case in our WATT1 experiments on 4 and 8 processors, as indicated in the annotations of Figure 3.46. This is not surprising, because the primary motivation of previous work in s-step GMRES was to decrease the run time of the orthogonalization process (see e.g., Bai et al. [19]). In the standard MGS implementation of GMRES, orthogonalization may take the majority of the runtime, so optimizing it was a first priority of authors developing s-step versions of GMRES.

Note that in a deployment-quality implementation of the matrix powers kernel, one would add the possibility of implementing the matrix powers kernel straightforwardly (that is, by using s invocations of SpMV, without any matrix powers kernel optimizations) to the autotuning process. We did not do so in the above performance results. Instead, we simply substituted the time for s SpMV operations in standard GMRES for the matrix powers kernel runtime. This may even be an overestimate of the resulting matrix powers kernel runtime, due to reuse of the sparse matrix. Standard GMRES only performs one SpMV operation before doing several vector operations. However, if the matrix powers kernel is implemented straightforwardly, it may go faster than s times the runtime of a single SpMV, since the matrix powers kernel reuses the sparse matrix s times. This is especially true if the matrix is small enough to fit in cache.

3.6.4 Implementation challenges

We discovered painfully that our benchmark implementation approach of hand-coding all of the parallelism constructs took much longer than anticipated. One challenge was that

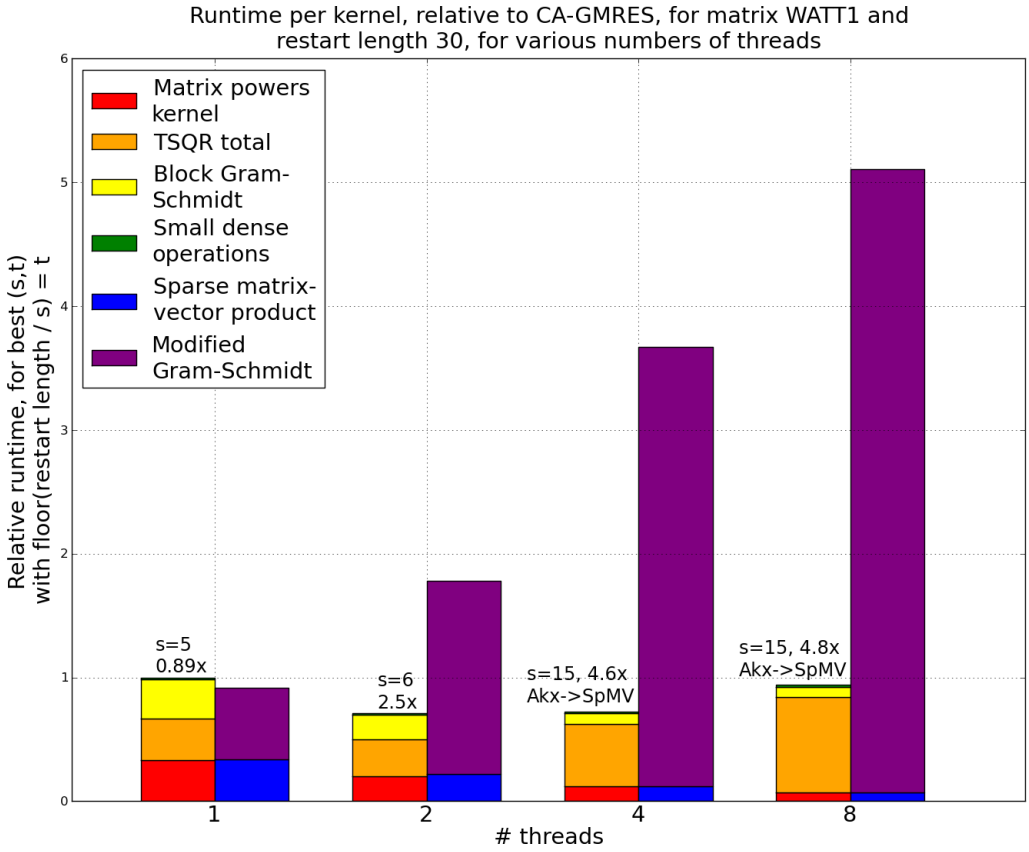


Figure 3.46: Relative run times of CA-GMRES and standard GMRES, with restart length 30, on the WATT1 matrix, on a single node of an 8-core Intel Clovertown. Each pair of bars corresponds to a different number of threads used: 1, 2, 4, or 8. In each pair, the left bar corresponds to CA-GMRES, and the right bar to standard GMRES. The top of each bar shows the total run time for that particular method (standard GMRES or CA-GMRES) and number of threads (1, 2, 4, or 8), relative to the total run time of CA-GMRES on one thread. That is, the total run time of CA-GMRES on one thread is scaled to be one. For each number of threads, the CA-GMRES run time shown is for the best (s, t) pair allowed by the matrix structure such that restart length$/s = t$. For simplicity, we only test s values that divide the restart length evenly, although the CA-GMRES algorithm does not require this. For both standard GMRES and CA-GMRES, its total run time shown is the sum of the total runtimes of the various kernels in that method. The height of each color within a bar represents the part of the total runtime taken by that particular kernel. The annotation above each pair of bars shows the best basis length s and the speedup *only for that number of threads* resulting from using CA-GMRES instead of standard GMRES (that is, the standard GMRES run time, divided by the CA-GMRES runtime). The actual speedup is 1.31×, which is the best possible CA-GMRES run time (in this case, on 2 threads) divided by the best possible standard GMRES run time (in this case, on 1 thread). If the annotation also contains "Akx→SpMV," then the matrix powers kernel would be faster (for that number of threads) if implemented by (parallel) SpMV.

we were limited to the low-level parallel primitives available in the Pthreads and system libraries. While our first pass at the benchmark used OpenMP, this did not give us control over memory affinity or over assignment of threads to cores. In particular, we needed to ensure both that only one thread would run on one core, and that benchmark runs with four or fewer threads would only use the four cores on one socket. (This was necessary for a fair comparison of the four-thread case with the eight-thread case, since threads running on one socket can share data via the shared cache.) Furthermore, the matrix powers kernel was written by a collaborator, Marghoob Mohiyuddin, using Pthreads, so that he could control affinity of data to threads in a NUMA implementation. Joining Pthreads code to OpenMP code correctly, without disturbing the careful association of data to threads, was too difficult for us to manage in the time allotted. This restricted us to a Pthreads-only implmentation.

Pthreads offers only limited collective operations, and its barrier implementation is known to be orders of magnitude slower than performance-oriented expert implementations (see Nishtala and Yelick [180]). This made us reach outside system libraries in search of implementations of parallel primitives, like barriers and reductions. We attempted to use the automatically tuned shared-memory collective operations by Nishtala and Yelick [180]. Their library chooses the best implementation of each collective among many possibilities, via runtime search at startup. While we had success with their barrier implementation, we found a bug in their reduction implementation (which the authors later fixed, but too late for us to use). Due to time constraints, we were forced to implement our own very simple reduction. While we found later that the threads spent very little time in the reduction and so our simple implementation did not significantly affect performance, it did increase programmer time and the potential for introducing additional bugs. This example illustrates the challenge of writing correct shared-memory parallel code using low-level primitives, as the primary implementer of Nishtala and Yelick's library is a highly skilled parallel systems programmer with experience in both the shared-memory and distributed-memory programming models.

We also encountered an interesting possible bug in the Intel Fortran Compiler's (version 10.1) implementation of the `ISO_C_BINDING` module. This module, defined in the Fortran 2003 standard, provides a set of constructs that make it easier for Fortran and C libraries to interact. We discovered that the `C_F_POINTER` subroutine seemed to be interacting oddly with multiple threads. Investigation with a tracing tool[6] showed that invocations of `C_F_POINTER` at the same part of a subroutine but in different threads were sometimes resulting in crossed pointer addresses, so that one thread could get another thread's data. We lacked the time to debug further or to provide a use case for verifying the bug, so to prevent the bug, we simply had to rewrite that part of the code in C. The code we had to rewrite was dense matrix manipulation code that is much easier to write in Fortran than C, which was a further hindrance.

3.6.5 Future work

In this section, we discuss future work, specifically for the parallel shared-memory implementation of CA-GMRES described here. Much of this will involve deploying CA-GMRES and

[6]Thanks to Heidi Pan, then a graduate student at MIT working in the UC Berkeley Parallel Computing Laboratory, for providing and helping with this tool.

the other Krylov methods described in this thesis for use in real applications. Other future work will include ways to improve performance by eliminating unnecessary synchronization between kernels.

Usability

Time constraints, due mainly to the challenges of writing and debugging correct low-level parallel code, meant that our benchmark was roughly written. This makes it hard to build and maintain, hard to port to different platforms, hard to optimize further, and impossible to expect nonexperts to use in realistic applications. A major near-term goal, therefore, is to make communication-avoiding implementations of the kernels of Chapter 2, and also entire solvers (such as CA-GMRES), available in an easy-to-use library. This requires solving two problems:

1. Making the kernels and solvers available for easy use in applications
2. Reducing the time required to do runtime search for kernel tuning parameters

We plan to solve the first problem by implementing the communication-avoiding kernels in the Trilinos [129] framework for solving "large-scale, complex multiphysics engineering and scientific problems" (Heroux et al. [130]). Trilinos provides sparse matrix primitives, such as data redistribution and repartitioning to minimize communication, that will make it easier to implement and experiment with the matrix powers kernel in the distributed-memory setting. This deserves more attention than we showed it in Demmel et al. [77], especially since distributed-memory programming models can be used to program shared-memory machines (e.g., one MPI process per socket, or one MPI process per compute accelerator such as a Graphics Processing Unit (GPU)). Trilinos also has hooks for experimentation with multiple levels of parallelism, which would expand our search space of kernel implementation possibilities.

The second problem mentioned above, the time required for runtime search of tuning parameters, is a problem often concealed by typical reporting of performance results. We did not show above the time required for runtime tuning of the CA-GMRES solver. In many cases, it took much longer than the actual benchmark. This was because our kernels were performing brute-force search over a large parameter space. In a real application, CA-GMRES will have to reduce this search time considerably. This can be helped in part by caching and reusing results of previous searches on the same or similar problems, as well as allowing more advanced users to give the kernels "hints" that can be used to prune the search space. These techniques are both described in Vuduc et al. [237]. We also plan to work with collaborators on improved search strategies for runtime tuning, such as machine learning. See Ganapathi et al. [107] for an example of how machine learning can improve both performance and total energy consumption of an automatically tuned scientific code.

Removing unnecessary interkernel synchronization

Dr. Sam Williams (Lawrence Berkeley National Laboratory) pointed out to us that in our CA-GMRES implementation, we were invoking our matrix powers, TSQR, and BGS kernels

in a bulk synchronous parallel way (see Valiant [224] for a description of the Bulk Synchronous Parallel (BSP) model of computation). Each kernel did its work on its own data in parallel, and all the threads had to wait at a barrier until the next kernel was allowed to start. In many cases, global synchronization between kernels is unnecessary. For example, once one processor finishes computing its matrix powers kernel output in CA-GMRES, that processor can begin the first step of parallel TSQR (the first local QR factorization) without communicating or synchronizing with the other processors. A problem with this approach is that it intertwines the kernel implementations, but this can be avoided by reimplementing the kernels as a collection of tasks connected by a task graph. Kernels then would execute in a dataflow fashion. Approaches like this have proven successful for one-sided dense matrix factorizations which combine multiple kernels. See e.g., Song et al. [212].

Chapter 4

Communication-avoiding symmetric Lanczos

In this chapter, we develop communication-avoiding versions of symmetric Lanczos iteration, a Krylov subspace method for solving symmetric positive definite eigenvalue problems. This chapter is organized as follows. Section 4.1 summarizes the standard symmetric Lanczos algorithm (Algorithm 24) for solving symmetric positive definite eigenvalue problems $Ax = \lambda x$. We show that standard Lanczos has communication-bound performance. Section 4.2 presents CA-Lanczos (Algorithm 25), a communication-avoiding version of symmetric Lanczos iteration for solving symmetric positive definite eigenvalue problems $Ax = \lambda x$. CA-Lanczos requires a factor of s times additional vector storage over standard Lanczos. However, in parallel, it saves a factor of s fewer messages in the dense vector operations over standard Lanczos. CA-Lanczos also communicates the sparse matrix a factor of s fewer times than standard Lanczos. While the CA-Lanczos algorithm is similar to an existing s-step version of Lanczos by Kim and Chronopoulos [151], it has the following advantages:

- Previous work did not include communication-avoiding kernels. CA-Lanczos uses the kernels of Chapter 2 to avoid communication in both the sparse matrix and the dense vector operations.
- Kim and Chronopoulos' s-step Lanczos was limited to using the monomial basis. Our CA-Lanczos algorithm can use any s-step basis, as well as the techniques discussed in Chapter 7 for improving numerical stability of the basis.
- CA-Lanczos can also avoid communication when performing reorthogonalization, and still reorthogonalize accurately. It does so using RR-TSQR-BGS (Algorithm 14 in Section 2.4.8). Communication-avoiding reorthogonalization is completely independent of the choice of Lanczos algorithm, so it may also be used with standard Lanczos. Previous work did not address the cost of reorthogonalization.

Section 4.3 develops two communication-avoiding versions of Lanczos (Algorithm 28 in Section 4.3.3 and Algorithm 29 in Section 4.3.4) for solving the generalized eigenvalue problem $Ax = \lambda Mx$, where A and M are both symmetric positive definite. We call them "left-preconditioned CA-Lanczos," because we use them in Chapter 5 as a model for developing

communication-avoiding versions of the Method of Conjugate Gradients (CG, for solving $Ax = b$). Algorithm 28, "Left-preconditioned CA-Lanczos without MGS," avoids communication in the sparse matrix operations. It is good for when the SpMV operations dominate the cost of standard Lanczos. Algorithm 29, "Left-preconditioned CA-Lanczos (with MGS)," avoids communication in both the sparse matrix operations, and the dense vector dot products. It is good for when the dot products in standard Lanczos have a significant cost. As far as we know, these two algorithms are completely novel.

4.1 Symmetric Lanczos

In this section, we discuss *symmetric Lanczos iteration*, a Krylov subspace method for solving eigenvalue problems $Ax = \lambda x$, where the matrix A is symmetric positive definite. We abbreviate symmetric Lanczos as "Lanczos" unless we have a need to distinguish it from the *non*symmetric Lanczos method, which is discussed in Chapter 6. Although Lanczos is used for solving eigenvalue problems, it is closely related to other Krylov methods for solving symmetric linear systems, such as the Method of Conjugate Gradients (CG – see Chapter 5) for SPD matrices, and the Minimum Residual method (MINRES) for symmetric indefinite matrices. We will use one of the communication-avoiding versions of Lanczos developed in Section 4.3 of this chapter as a model to derive communication-avoiding CG in Chapter 5.

This section is organized as follows. Section 4.1.1 summarizes the standard Lanczos algorithm. Section 4.1.2 explains the cost of communication in standard Lanczos, and shows that the communication-avoiding versions of Lanczos we develop later in this chapter will require a factor of s more storage and arithmetic operations in order to avoid communication. Finally, Section 4.1.3 demonstrates how to avoid communication when performing reorthogonalization. This can be done whether using standard Lanczos or one of the communication-avoiding versions of Lanczos we develop later in this chapter.

4.1.1 The standard Lanczos algorithm

Lanczos begins with an $n \times n$ SPD matrix A and a starting vector v. After s steps, if the method does not break down, it computes an $s+1$ by s tridiagonal matrix $\underline{T}$ (with principal $s \times s$ submatrix T) and an n by s orthonormal resp. unitary basis matrix $Q = [q_1, q_2, \ldots, q_s]$ such that

$$AQ = QT + \beta_{s+1} q_{s+1} e_s^T, \tag{4.1}$$

where q_{s+1} is unit-length and orthogonal to $q_1, \ldots, q_s$. The symmetric tridiagonal matrix T has the following form:

$$T = \begin{pmatrix} \alpha_1 & \beta_2 & & \ldots & 0 \\ \beta_2 & \alpha_2 & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & \beta_s \\ 0 & \ldots & & \beta_s & \alpha_s \end{pmatrix}$$

We refer to q_1, q_2, ..., q_{s+1} as the *Lanczos basis vectors*. They span the Krylov subspace $\text{span}\{v, Av, A^2v, \ldots, A^sv\}$. They also satisfy the recurrence

$$\beta_{j+1}q_{j+1} = Aq_j - \alpha_j q_j - \beta_j q_{j-1} \tag{4.2}$$

for each $j = 1, 2, \ldots, s$, where we set $q_0 = 0_n$. Lanczos computes $\alpha_j = \langle Aq_j, q_j \rangle$ and $\beta_{j+1} = \|Aq_j - \alpha_j q_j - \beta_j q_{j-1}\|_2$, each of which requires one inner product. Algorithm 24 shows the Lanczos method.

Line 8 of Algorithm 24 refers to the possibility of Lanczos breaking down. In exact arithmetic, Lanczos only breaks down when the smallest j has been reached such that a degree $j-1$ polynomial p_{j-1} exists with $p_{j-1}(A)r = 0$. This is called a *lucky breakdown* for two reasons: when solving eigenvalue problems, the set of eigenvalues of T equals the set of eigenvalues of A, and when solving linear systems, the current approximate solution equals the exact solution. Lucky breakdowns are rarely encountered in practice, and so we do not consider them here.

Algorithm 24 has many alternate formulations, which affect its performance characteristics as well as its accuracy in finite-precision arithmetic. See e.g., Cullum and Willoughby [65] for some alternate formulations. Regardless, ech iteration requires one SpMV, two dot products, and two AXPY operations. If only the tridiagonal matrix T is desired, only the two rightmost columns of the basis matrix Q need to be kept at any one time.

Algorithm 24 Symmetric Lanczos iteration

Require: r is the starting vector

1: $\beta_0 := 0$, $\beta_1 := \|r\|_2$, $q_0 := 0$, $q_1 := r/\beta_1$
2: **for** $k = 1$ to convergence **do**
3: $\quad w_j := Aq_k - \beta_k q_{k-1}$
4: $\quad \alpha_k := \langle w_j, q_k \rangle$
5: $\quad w_j := w_j - \alpha_k q_k$
6: $\quad \beta_{k+1} := \|w_j\|_2$
7: $\quad$ **if** $\beta_{k+1} = 0$ **then**
8: $\quad\quad$ Exit due to lucky breakdown
9: $\quad$ **end if**
10: $\quad q_{k+1} := w_j/\beta_{k+1}$
11: **end for**

Lanczos can be considered as a special case of Arnoldi iteration for when the matrix A is symmetric (in the real case) or Hermitian (in the complex case). In exact arithmetic, if $A = A^*$, then the Lanczos basis vectors are the same as the Arnoldi basis vectors, and $\underline{T} = \underline{H}$. In other words, the symmetry of the matrix A allows simplification of the Arnoldi orthogonalization procedure to a short recurrence. This simplification is the main reason for using Lanczos instead of Arnoldi, as it prevents the number of vector operations from increasing linearly with the current iteration number, rather than remaining constant. Also, for certain applications, Lanczos only needs to store three basis vectors at once,[1] unlike Arnoldi, which must store as many basis vectors as the restart length.

[1] Actually only two basis vectors are needed, if the sparse matrix-vector product kernel computes $u := u + A \cdot v$ rather than $u := A \cdot v$; see [189, p. 290].

4.1.2 Communication in Lanczos iteration

In the previous Chapter 3, we developed communication-avoiding versions of Arnoldi iteration. Arnoldi benefits significantly from communication-avoiding techniques, but Lanczos has different performance characteristics than Arnoldi. Depending on the restart length r and the structure of the sparse matrix, Arnoldi may spend significantly more time orthogonalizing its basis vectors than doing sparse matrix-vector products. For example, standard Arnoldi requires $r^2/2 + O(r)$ inner products and $r^2/2 + O(r)$ vector subtractions per restart cycle, and only r sparse matrix-vector products per restart cycle. The combination of TSQR and Block Gram-Schmidt performs much better than MGS orthogonalization for sufficiently long vectors, as the performance results (for CA-GMRES) in Section 3.6 show. Thus, CA-Arnoldi may perform much better than standard Arnoldi, even if the sparse matrix-vector products are not replaced with the matrix powers kernel.

In contrast, r iterations of symmetric Lanczos require only $2r$ inner products and $2r$ AXPY operations, but have the same number of sparse matrix-vector products as r iterations of Arnoldi. Even though Arnoldi and Lanczos are equivalent in exact arithmetic for a symmetric resp. Hermitian matrix A, Lanczos exploits that symmetry to avoid performing a full orthogonalization against all previous basis vectors at every step. Furthermore, we will show that communication-avoiding versions of Lanczos require a factor of $\Theta(s)$ more storage and arithmetic operations than standard Lanczos. This is also true for the method of conjugate gradients. As a result, communication-avoiding versions of Lanczos require larger communication costs relative to arithmetic costs than Arnoldi requires in order to pay off.

4.1.3 Communication-avoiding reorthogonalization

Reorthogonalization in standard Lanczos can be done in a communication-avoiding way, by using an algorithm analogous to the RR-TSQR-BGS orthogonalization method (Algorithm 14 of Section 2.4.8). This saves a factor of $\Theta(s)$ messages in parallel and a factor of $\Theta(s)$ words transferred between levels of the memory hierarchy, yet achieves comparable or better accuracy to full column-by-column MGS reorthogonalization. This works whether or not a communication-avoiding Lanczos algorithm is used.

What is reorthogonalization?

Standard Lanczos iteration often requires *reorthogonalizing* the Krylov basis vectors, in order to maintain accuracy of the computed eigenvalues and eigenvectors. This is because the vectors tend to lose their orthogonality in finite-precision arithmetic. Paige [185] found that the main cause of their loss of orthogonality is convergence of a particular Ritz value to an eigenvalue of the matrix A, so this is not merely an issue of slow degradation due to gradual accumulation of rounding errors. In particular, this occurs even for variants of Lanczos that converge in only a few iterations, such as shift and invert for eigenvalues in the interior of the spectrum, as well as for runs of many iterations. This makes reorthogonalization an important part of the Lanczos method.

Reorthogonalization schemes

Different reorthogonalization schemes are used in practice, depending on the problem size and the number of iterations required. *Full reorthogonalization* means making all the basis vectors orthogonal. Typically this is done using Modified Gram-Schmidt, or sometimes Classical Gram-Schmidt. Full reorthogonalization is an obvious candidate for RR-TSQR-BGS (Algorithm 14 of Section 2.4.8), which combines TSQR and Block Gram-Schmidt.

Selective reorthogonalization, as described in Bai et al. [16], only requires reorthogonalizing certain vectors, based on an inexpensively computed estimate of their loss of orthogonality. The point of selective reorthogonalization is to avoid the cost of a full reorthogonalization, but RR-TSQR-BGS may make full reorthogonalization inexpensive enough to outweigh the savings of the more complicated selective reorthogonalization procedures. Furthermore, future work may suggest ways to do selective reorthogonalization in a communication-avoiding way, for example, by grouping the selected vectors into blocks ad using

Local reorthogonalization is a last resort for when memory constraints forbid keeping all the previous basis vectors, or performance constraints forbid reorthogonalizing against all of them. Bai et al. [16] describe an approach in which at every iteration, the current basis vector is reorthogonalized against the previous two basis vectors. One of our CA-Lanczos implementations will use TSQR to orthogonalize groups of s basis vectors at once. We conjecture that this scheme is no less accurate, and perhaps more accurate, than local reorthogonalization against the previous s basis vectors. The "automatic local reorthogonalization" that our CA-Lanczos algorithm provides may delay the need for selective or full reorthogonalizations.

4.2 CA-Lanczos

In this section, we present CA-Lanczos (Algorithm 25), a communication-avoiding version of symmetric Lanczos iteration for solving SPD eigenvalue problems $Ax = \lambda x$. CA-Lanczos uses the matrix powers kernel (Section 2.1) to avoid a factor of s communication in the sparse matrix-vector products over standard Lanczos. It also uses the TSQR (Section 2.3) and Block Gram-Schmidt (Section 2.4) kernels to save a factor of $\Theta(s)$ fewer messages in parallel in the vector operations. The vector operations require no additional traffic between levels of the memory hierarchy than those of standard Lanczos, and also are implemented with BLAS 3 kernels instead of BLAS 1 or 2 kernels. Unlike CA-Arnoldi and CA-GMRES, however, CA-Lanczos requires a factor of $\Theta(s)$ more computation and a factor of s additional vector storage over standard Lanczos. This extra work and storage perform the same work as partial reorthogonalization. Partial reorthogonalization may improve convergence in finite-precision arithmetic (though not in exact arithmetic) when using Krylov methods to compute eigenvalues. Thus, the extra arithmetic operations mean that CA-Lanczos may be more accurate than standard Lanczos.

Just like CA-Arnoldi (Section 4.2) and CA-GMRES (Section 3.4), CA-Lanczos takes two parameters s and t. Similarly, we also call CA-Lanczos "Lanczos(s, t)." The s parameter is the s-step basis length, and the t parameter is the number of outer iterations before restart. The restart length is thus $r = s \cdot t$. Unlike CA-Arnoldi or CA-GMRES, the amount of storage, floating-point operations, and communication does not increase linearly with each outer

iteration; in CA-Lanczos, all of these are the same at each outer iteration. This means that restarting is not necessary as a cost-saving measure. However, restarting standard symmetric Lanczos is helpful in practice for maintaining orthogonality of the basis vectors (see e.g., [16, "Implicit Restart"]), so we expect this to be true also for CA-Lanczos. Restarted versions of Lanczos typically invoke the restart length number of iterations of standard Lanczos as as "inner loop," and we expect to be able to replace that with an invocation of CA-Lanczos without damaging convergence. Indeed, CA-Lanczos may be able to use a longer restart length, as we expect the basis vectors to lose their orthogonality a factor of s times slower than standard Lanczos.

The CA-Lanczos algorithm was developed independently, but it is similar to the "s-step Lanczos" algorithm of Kim and Chronopoulos [151]. CA-Lanczos has the following advantages over s-step Lanczos.

- Kim and Chronopoulos did not use communication-avoiding kernels. CA-Lanczos uses the kernels of Chapter 2 to avoid communication in both the sparse matrix and the dense vector operations.
- Kim and Chronopoulos algorithm was limited to using the monomial basis. Our CA-Lanczos algorithm can use any s-step basis, as well as the techniques discussed in Chapter 7 for improving numerical stability of the basis.
- CA-Lanczos can also avoid communication when performing reorthogonalization, and still reorthogonalize accurately. It does so using RR-TSQR-BGS (Algorithm 14 in Section 2.4.8). Communication-avoiding reorthogonalization is completely independent of the choice of Lanczos algorithm, so it may also be used with standard Lanczos. Previous work did not address the cost of reorthogonalization.

Symmetric Lanczos produces two different outputs: a collection of mutually orthogonal unit-length basis vectors, and an associated tridiagonal matrix which is the projection of the original matrix A onto that set of basis vectors. Our CA-Lanczos produces exactly the same output in exact arithmetic, except that it produces them in groups of s at a time, instead of one at a time. In Section 4.2.1, we explain how we how we orthogonalize each group of s new basis vectors against the previous basis vectors. Then, in Section 4.2.2, we show how to update the Lanczos tridiagonal matrix, using the matrix powers basis coefficients and the coefficients from the orthogonalization.

4.2.1 CA-Lanczos update formula

In this section, we present the CA-Lanczos update formula, which shows how we efficiently orthogonalize each outer iteration's group of s basis vectors against the previously orthogonalized basis vectors. Lanczos(s, t) is a special case of Arnoldi(s, t), so we model the presentation here from that in Section 3.3.3. The difference is that Lanczos assumes that the matrix A is symmetric, so the basis vectors satisfy a three-term recurrence which can be exploited to orthogonalize them efficiently. Arnoldi does not assume that A is symmetric, so each new basis vector must be orthogonalized against all the previous basis vectors. The usual implementation of Lanczos need only keep a "window" of three basis vectors at one time, as one can see from the inner loop:

1: $w_j := Aq_j - \beta_j q_{j-1}$
2: $\alpha_j := \langle w_j, q_j \rangle$
3: $w_j := w_j - \alpha_j q_j$
4: $\beta_{j+1} := \|w_j\|_2$
5: $q_{j+1} := w_j / \beta_{j+1}$

We will see that CA-Lanczos needs to keep a window of two outer iterations' worth ($2s$) of basis vectors at once. Also, each new basis vector may need to be orthogonalized against a different number of previous basis vectors (but no more than $2s$ of them). This means CA-Lanczos requires a factor of $\Theta(s)$ more storage and $\Theta(s)$ more computation than the usual implementation of Lanczos. However, even if we neglect the substantial benefits of the matrix powers kernel, just the orthogonalization part of CA-Lanczos requires no more words transferred between levels of the memory hierarchy than the orthogonalization operations in the equivalent s number of iterations of standard Lanczos. In parallel, Lanczos(s, t) saves a factor of $\Theta(s)$ messages over s iterations of standard Lanczos.

Notation

In this section, we recycle most of the notation in Section 3.3.3, where we present the Arnoldi(s, t) update procedure. For Lanczos, we find it easier to use the overlapping approach to orthogonalization. This motivates our use of $\underline{\acute{V}}_k$, the *right-shifted basis matrix* at outer iteration k:

$$\acute{V}_k := [v_{sk+2}, \ldots, v_{sk+s}],$$
$$\underline{\acute{V}}_k := [\acute{V}_k, v_{sk+s+1}].$$

The acute accent, which points up and to the right, hints at a shift of the "window of vectors" over to the right by one. Similarly, we define $\underline{\grave{V}}_k$, the *left-shifted basis matrix* at outer iteration k:

$$\grave{V}_k := \begin{cases} [0_{n,s}, V_0] & \text{for } k = 0, \\ [v_{sk}, V_k] & \text{for } k > 0; \end{cases}$$
$$\underline{\grave{V}}_k := [\grave{V}_k, v_{sk+s+1}].$$

The only time we will use $\underline{\grave{V}}_k$, though, is in the discussion in Section 4.2.1. For actual computations, we will use the right-shifted basis matrix $\underline{\acute{V}}_k$.

Updating the QR factorization

At outer iteration k, the matrix powers kernel takes the last basis vector $q_{sk+1} = v_{sk+1}$ from the previous outer iteration, and produces s more basis vectors v_{sk+2}, …, v_{sk+s+1} which satisfy

$$AV_k = \underline{V}_k \underline{B}_k$$

for the $s+1$ by s tridiagonal basis matrix $\underline{B}_k$. All this is the same as with Arnoldi(s, t). If we were doing Arnoldi with overlapping orthogonalization, we would then orthogonalize $\underline{\acute{V}}_k$ against the previous basis vectors via a QR factorization update:

1: $\underline{\acute{\mathfrak{R}}}_{k-1,k} := \underline{\mathfrak{Q}}_{k-1}^* \underline{\acute{V}}_k$
2: $\underline{\acute{V}}_k := \underline{\acute{V}}_k - \underline{\mathfrak{Q}}_{k-1} \underline{\acute{\mathfrak{R}}}_{k-1,k}$

3: Compute the n by s QR factorization $\acute{\underline{Q}}_k \cdot \acute{\underline{R}}_k = \acute{\underline{V}}_k$

4: $\underline{\mathfrak{Q}}_k := [\underline{\mathfrak{Q}}_{k-1}, \acute{\underline{Q}}_k]$

5: $\underline{\mathfrak{R}}_k := \begin{pmatrix} \underline{\mathfrak{R}}_{k-1} & \acute{\underline{\mathfrak{R}}}_{k-1,k} \\ 0_{s,s(k-1)+1} & \acute{\underline{R}}_k \end{pmatrix}$

Lines 4 and 5 involve no additional work, only notation. Lines 1 and 2 include all the work of orthogonalizing $\acute{\underline{V}}_k$ against the previous basis vectors. In the case of Lanczos, we can save most of the work in these lines, in that we only have to orthogonalize against $\underline{Q}_{k-1}$ (one previous group of basis vectors). The proof of this will fall out of the argument in Section 4.2.1. This optimization means that

$$\acute{\underline{\mathfrak{R}}}_{k-1,k} := \underline{\mathfrak{Q}}_{k-1}^* \acute{\underline{V}}_k = \begin{pmatrix} 0_{s(k-2),s} \\ \underline{Q}_{k-1}^* \acute{\underline{V}}_k \end{pmatrix}$$

This reduces line 1 above to

$$\acute{\underline{R}}_{k-1,k} := \underline{Q}_{k-1}^* \acute{\underline{V}}_k$$

and line 2 above to

$$\acute{\underline{V}}_k := \acute{\underline{V}}_k - \underline{Q}_{k-1} \acute{\underline{R}}_{k-1,k}.$$

That means in each outer iteration of CA-Lanczos, we have to keep around the previous outer iteration's group of orthogonalized basis vectors $\underline{Q}_{k-1}$.

Can we do better?

Section 4.2.1 suggests what we will show later, that CA-Lanczos costs a factor of $\Theta(s)$ more flops than standard Lanczos. Furthermore, $\Theta(s)$ times more basis vectors must be stored. One might wonder whether we could do better, perhaps by exploiting the structure of the basis matrix $\underline{B}_k$ or that the matrix A is symmetric. If a three-term recurrence works in the original Lanczos algorithm, why wouldn't it suffice to invoke TSQR once to orthogonalize $\grave{\underline{V}}_k$, as it contains the last two basis vectors from the previous outer iteration? In this section, we will show that in general, one invocation of TSQR on $\grave{\underline{V}}_k$ does not suffice to compute the correct Lanczos basis vectors. In fact, in exact arithmetic, we will show that it is necessary to keep all columns of $\underline{Q}_{k-1}$ but the first, $q_{s(k-1)+1}$.

In finite-precision arithmetic, the basis vectors in the usual implementation of Lanczos rapidly lose orthogonality without reorthogonalization. CA-Lanczos uses a more accurate orthogonalization scheme which should require reorthogonalization less often. Reorthogonalization can be expensive both in terms of communication and in terms of computation, and the extra work performed by CA-Lanczos in each outer iteration may save extra reorthogonalization work later. How much work is saved over standard Lanczos depends on the specific problem as well as the reorthogonalization techniques used.

To show that CA-Lanczos requires all the previous outer iterations' basis vectors, we begin by defining

$$\acute{\underline{R}}_{k-1,k} := \underline{Q}_{k-1}^* \acute{\underline{V}}_k.$$

The i, j entry of $\acute{\underline{R}}_{k-1,k}$ is given by

$$\acute{\underline{R}}_{k-1,k}(i,j) = q_{s(k-1)+i}^* v_{sk+j+1} = \langle v_{sk+1+j}, q_{sk+1+(i-s-1)} \rangle,$$

where $i \in \{1, 2, \ldots, s+1\}$ and $j \in \{1, 2, \ldots, s\}$. We then expand out v_{sk+1+j} as a sum of the vectors $v_{sk+1} = q_{sk+1}$, Aq_{sk+1}, $\ldots$, $A^{j-1}q_{sk+1}$:

$$\langle v_{sk+1+j}, q_{sk+1+(i-s-1)} \rangle = \sum_{l=0}^{j-1} \sigma_{jl} \langle A^l q_{sk+1}, q_{sk+1+(i-s-1)} \rangle,$$

where the σ_{jl} are possibly nonzero constants whose values we won't need. (The use of the monomial basis here has nothing to do with the s-step basis which we are using for CA-Lanczos; it just makes the proof easier.) By expanding out the Lanczos recurrence (either for $A^l q_{sk+1}$, or for $A^l q_{sk+1+(i-s-1)}$, the latter only if we exploit the symmetry of A by moving A^l to the other argument of the inner product), we see that this is nonzero only if $i+j \geq s+2$. (By adjusting the range of i appropriately, we can also see that $\underline{Q}_j^* \underline{\acute{V}}_k = 0_{s+1,s}$ for $j < k-1$.)

Since $\underline{\acute{R}}_{k-1,k}(i,j)$ is nonzero only if $i+j \geq s+2$, $\underline{\acute{R}}_{k-1,k}$ has a structure like the following example for $s = 5$:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & x \\ 0 & 0 & 0 & x & x \\ 0 & 0 & x & x & x \\ 0 & x & x & x & x \\ x & x & x & x & x \end{pmatrix},$$

where x indicates a possibly nonzero value. In fact, we can't assume anything more specific than that about the nonzero structure of $\underline{\acute{R}}_{k-1,k}$, because the s-step basis can express the basis vector v_{sk+1+j} as $p_j(A)q_{sk+1}$ for *any* monic degree j polynomial $p_j(\cdot)$. When we apply the update

$$\underline{\acute{V}}_k := \underline{\acute{V}}_k - \underline{Q}_{k-1} \underline{\acute{R}}_{k-1,k},$$

we can exploit the above structure of $\underline{\acute{R}}_{k-1,k}$ in order to save some computation and communication (not latency, but bandwidth). We are not required to exploit that structure, and it could be that not exploiting the structure could produce an orthogonalization of better quality in floating-point arithmetic.

Repartitioning the R factor

When showing how to compute the Lanczos tridiagonal matrix from the R factor in the above QR factorization update, it simplifies the notation if we first "repartition" the block matrix $\underline{\mathfrak{R}}_k$. This only means dividing it up into blocks a different way; repartitioning requires no computation and insignificant data movement.

To this end, we first define four variations on the block inner product:

$$\begin{aligned} R_{k-1,k} &= Q_{k-1}^* V_k, \\ \underline{R}_{k-1,k} &= Q_{k-1}^* \underline{V}_k, \\ \acute{R}_{k-1,k} &= \underline{Q}_{k-1}^* \acute{V}_k, \\ \underline{\acute{R}}_{k-1,k} &= \underline{Q}_{k-1}^* \underline{\acute{V}}_k. \end{aligned} \tag{4.3}$$

The notation is complicated but the pattern is this: the acute accent means $\acute{V}_k$ is involved, and the underline means $\underline{V}_k$ is involved. Both an acute accent and an underline means that $\underline{\acute{V}}_k$ is involved. Given this notation, we can partition $\mathfrak{R}_k$ and $\underline{\mathfrak{R}}_k$ in two different ways:

$$\begin{aligned}
\mathfrak{R}_k &= \begin{pmatrix} \underline{\mathfrak{R}}_{k-1} & \begin{pmatrix} 0_{s(k-2),s-1} \\ \acute{R}_{k-1,k} \end{pmatrix} \\ 0_{s,s(k-1)+1} & \acute{R}_k \end{pmatrix} = \begin{pmatrix} \mathfrak{R}_{k-1} & \begin{pmatrix} 0_{s(k-2),s} \\ R_{k-1,k} \end{pmatrix} \\ 0_{s,s(k-1)} & R_k \end{pmatrix}, \text{ and} \\
\underline{\mathfrak{R}}_k &= \begin{pmatrix} \underline{\mathfrak{R}}_{k-1} & \begin{pmatrix} 0_{s(k-2),s} \\ \underline{\acute{R}}_{k-1,k} \end{pmatrix} \\ 0_{s,s(k-1)+1} & \underline{\acute{R}}_k \end{pmatrix} = \begin{pmatrix} \mathfrak{R}_{k-1} & \begin{pmatrix} 0_{s(k-2),s+1} \\ \underline{R}_{k-1,k} \end{pmatrix} \\ 0_{s,s(k-1)} & \underline{R}_k \end{pmatrix}.
\end{aligned} \tag{4.4}$$

4.2.2 Updating the tridiagonal matrix

If CA-Lanczos performs all its iterations without breaking down, it produces the following matrix relation:

$$A\mathfrak{Q}_k = \underline{\mathfrak{Q}}_k \underline{\mathfrak{T}}_k,$$

where

$$\begin{aligned}
\mathfrak{Q}_k &= [Q_0, Q_1, \ldots, Q_k], \\
\underline{\mathfrak{Q}}_k &= [\mathfrak{Q}_k, q_{s(k+1)+1}],
\end{aligned}$$

$$\mathfrak{T}_k = \begin{cases} T_0 & \text{for } k = 0, \\ \begin{pmatrix} \mathfrak{T}_{k-1} & \beta_{sk+1} e_{s(k-1)} e_1^T \\ \beta_{sk+1} e_1 e_{s(k-1)}^T & T_k \end{pmatrix} & \text{for } k > 0, \end{cases}$$

and

$$\underline{\mathfrak{T}}_k = \begin{pmatrix} \mathfrak{T}_k \\ \beta_{s(k+1)+1} e_{s(k+1)}^T \end{pmatrix}.$$

That is the end result, but how do we compute the symmetric tridiagonal matrix $\underline{\mathfrak{T}}_k$? We begin just as we did with Arnoldi(s, t) in Section 3.3, using a similar notation, and then gradually apply the additional assumptions of the Lanczos process until we reach a computational method of the desired efficiency. Suppose that we have already computed $\underline{\mathfrak{T}}_{k-1}$ and the new basis $\underline{V}_k$. Then, $A\mathfrak{Q}_{k-1} = \underline{\mathfrak{Q}}_{k-1}\underline{\mathfrak{T}}_{k-1}$ and thus

$$A[\mathfrak{Q}_{k-1}, V_k] = [\mathfrak{Q}_k, \underline{V}_k]\underline{\mathfrak{B}}_k, \tag{4.5}$$

where

$$\begin{aligned}
\mathfrak{B}_k &= \begin{pmatrix} \mathfrak{T}_{k-1} & 0_{s(k-1),s} \\ \beta_{sk+1} e_1 e_{s(k-1)}^T & B_k \end{pmatrix}, \text{ and} \\
\underline{\mathfrak{B}}_k &= \begin{pmatrix} \mathfrak{T}_{k-1} & 0_{s(k-1),s} \\ \beta_{sk+1} e_1 e_{s(k-1)}^T & \underline{B}_k \end{pmatrix}.
\end{aligned}$$

Now, we combine the QR factorization update in Section 4.2.1 with the Lanczos relation

$A\mathfrak{Q}_k = \underline{\mathfrak{Q}}_k \underline{\mathfrak{T}}_k$:

$$A[\mathfrak{Q}_{k-1}, Q_k] \begin{pmatrix} I_{s(k-1),s(k-1)} & \begin{pmatrix} 0_{s(k-2),s} \\ R_{k-1,k} \end{pmatrix} \\ 0_{s,s(k-1)} & R_k \end{pmatrix} = [\mathfrak{Q}_{k-1}, \underline{Q}_k] \begin{pmatrix} I_{s(k-1),s(k-1)} & \begin{pmatrix} 0_{s(k-2),s} \\ \underline{R}_{k-1,k} \end{pmatrix} \\ 0_{s,s(k-1)} & \underline{R}_k \end{pmatrix} \begin{pmatrix} \mathfrak{T}_{k-1} & 0_{s(k-1),s} \\ \beta_{sk+1} e_1 e_{s(k-1)}^T & \underline{B}_k \end{pmatrix}.$$

Here, we use the repartitioning of the R factors to make sure that the subblocks of the $\mathfrak{R}$, $\underline{\mathfrak{R}}$, and $\underline{\mathfrak{B}}_k$ matrices line up. Assuming that the Lanczos recurrence hasn't broken down, we then have

$$\underline{\mathfrak{T}}_k = \begin{pmatrix} I_{s(k-1),s(k-1)} & \begin{pmatrix} 0_{s(k-2),s} \\ \underline{R}_{k-1,k} \end{pmatrix} \\ 0_{s,s(k-1)} & \underline{R}_k \end{pmatrix} \begin{pmatrix} \mathfrak{T}_{k-1} & 0_{s(k-1),s} \\ \beta_{sk+1} e_1 e_{s(k-1)}^T & \underline{B}_k \end{pmatrix} \begin{pmatrix} I_{s(k-1),s(k-1)} & \begin{pmatrix} 0_{s(k-2),s} \\ R_{k-1,k} \end{pmatrix} \\ 0_{s,s(k-1)} & R_k \end{pmatrix}^{-1} = \begin{pmatrix} I & \begin{pmatrix} 0_{s(k-2),s} \\ \underline{R}_{k-1,k} \end{pmatrix} \\ 0_{s,s(k-1)} & \underline{R}_k \end{pmatrix} \begin{pmatrix} \mathfrak{T}_{k-1} & 0 \\ \beta_{sk+1} e_1 e_{s(k-1)}^T & \underline{B}_k \end{pmatrix} \begin{pmatrix} I & \begin{pmatrix} 0_{s(k-2),s} \\ -R_{k-1,k} R_k^{-1} \end{pmatrix} \\ 0 & R_k^{-1} \end{pmatrix}.$$

Now we can compute this product (algebraically, not computationally – the algorithm doesn't need to do any of this work):

$$\underline{\mathfrak{T}}_k = \begin{pmatrix} I & \begin{pmatrix} 0_{s(k-2),s} \\ \underline{R}_{k-1,k} \end{pmatrix} \\ 0 & \underline{R}_k \end{pmatrix} \begin{pmatrix} \mathfrak{T}_{k-1} & \begin{pmatrix} 0_{s(k-2),s} \\ -T_{k-1} \underline{R}_{k-1,k} R_k^{-1} \end{pmatrix} \\ \beta_{sk+1} e_1 e_{s(k-1)}^T & -\beta_{sk+1} e_1 e_s^T R_{k-1,k} R_k^{-1} + \underline{B}_k R_k^{-1} \end{pmatrix} = \begin{pmatrix} \mathfrak{T}_{k-1} + \beta_{sk+1} \begin{pmatrix} 0_{s(k-2),s} \\ \underline{R}_{k-1,k} e_1 \end{pmatrix} e_{s(k-1)}^T & \begin{pmatrix} 0_{s(k-2),s} \\ \text{Mess} \end{pmatrix} \\ \beta_{sk+1} \underline{R}_k e_1 e_{s(k-1)}^T & -\beta_{sk+1} \underline{R}_k e_1 e_s^T R_{k-1,k} R_k^{-1} + \underline{R}_k \underline{B}_k R_k^{-1} \end{pmatrix}. \quad (4.6)$$

In Equation (4.6) above, Mess in the $1,2$ block of $\underline{\mathfrak{T}}_k$ is an expression taking up enough space that we write it separately here:

$$\text{Mess} = -T_{k-1} R_{k-1,k} R_k^{-1} - \beta_{sk+1} \underline{R}_{k-1,k} e_1 e_s^T R_{k-1,k} R_k^{-1} + \underline{R}_{k-1,k} \underline{B}_k R_k^{-1}. \quad (4.7)$$

We can simplify $\underline{\mathfrak{T}}_k$ in Equation (4.6) by making the following simplifications:

- $\underline{R}_{k-1,k} e_1 = 0_{s,1}$ and $R_{k-1,k} e_1 = 0_{s,1}$
- $\underline{R}_k e_1 = e_1$ and $R_k e_1 = e_1$
- $\underline{R}_{k-1,k} \underline{B}_k R_k^{-1} = \beta_{sk+1} e_s + T_{k-1} R_{k-1,k} R_k^{-1}$

The first simplification follows from the first column of $\underline{V}_k$ (namely q_{sk+1}) being already both orthogonal to the columns of Q_{k-1} and of unit length. The QR factorization update thus makes the $1,1$ entry of R_k one. The second simplification follows from

$$\underline{R}_{k-1,k}e_1 = Q_{k-1}^* \underline{V}_k e_1 = Q_{k-1}^* q_{sk+1} = 0_{s,1}.$$

Finally, the third simplification is a long chain of deductions:

$$\begin{aligned}
\underline{R}_{k-1,k}\underline{B}_k R_k^{-1} = Q_{k-1}^*(\underline{V}_k \underline{B}_k)R_k^{-1} = Q_{k-1}^*(AV_k)R_k^{-1} = \\
Q_{k-1}^* A(Q_k R_k + Q_{k-1}R_{k-1,k})R_k^{-1} = Q_{k-1}^*(AQ_k + AQ_{k-1}R_{k-1,k}R_k^{-1}) = \\
Q_{k-1}^*\left(\beta_{sk+1}q_{sk} + \underline{Q}_k\underline{T}_k + \beta_{s(k-1)+1}q_{s(k-1)}R_k^{-1} + \underline{Q}_{k-1}\underline{T}_{k-1}R_{k-1,k}R_k^{-1}\right) = \\
\beta_{sk+1}e_s + [I_{s,s}, 0_{s,1}]\begin{pmatrix} T_{k-1} \\ \beta_{sk+1}e_s^T \end{pmatrix} R_{k-1,k}R_k^{-1} = \\
\beta_{sk+1}e_s + T_{k-1}R_{k-1,k}R_k^{-1}.
\end{aligned}$$

Note the use of the Lanczos recurrence in

$$\begin{aligned}
AQ_k &= \beta_{sk+1}q_{sk} + \underline{Q}_k\underline{T}_k, \\
AQ_{k-1} &= \beta_{s(k-1)+1}q_{s(k-1)} + \underline{Q}_{k-1}\underline{T}_{k-1}.
\end{aligned}$$

Given all these simplifications, the expression for $\underline{\mathfrak{T}}_k$ in Equation (4.6) reduces nicely to

$$\underline{\mathfrak{T}}_k = \begin{pmatrix} \mathfrak{T}_{k-1} & \beta_{sk+1}e_{s(k-1)}e_1^T \\ \beta_{sk+1}e_1 e_{s(k-1)}^T & \underline{R}_k\underline{B}_k R_k^{-1} - \beta_{sk+1}e_1 e_s^T R_{k-1,k}R_k^{-1} \end{pmatrix}. \tag{4.8}$$

This means that the way we compute $\underline{\mathfrak{T}}_k$ makes it structurally at least block tridiagonal. Furthermore, this requires only a light assumption on the orthogonality of the basis vectors, namely that neighboring groups of basis vectors Q_{k-1} and $\underline{Q}_k$ are orthogonal. This assumption is easily satisfied nearly to machine precision by the QR update process, which uses a kind of modified block Gram-Schmidt.

Let $\underline{T}_k$ be the $s+1$ by s lower left block of $\underline{\mathfrak{T}}_k$. We can simplify it further. First,

$$\begin{aligned}
\underline{R}_k\underline{B}_k R_k^{-1} &= \begin{pmatrix} R_k & z_k \\ 0 & \rho_k \end{pmatrix}\begin{pmatrix} B_k \\ b_k e_s^T \end{pmatrix} = \begin{pmatrix} R_k B_k + b_k z_k e_s^T \\ \rho_k b_k e_s^T \end{pmatrix} R_k^{-1} \\
&= \begin{pmatrix} R_k B_k R_k^{-1} + \tilde{\rho}_k^{-1} b_k z_k e_s^T \\ \tilde{\rho}_k^{-1}\rho_k b_k e_s^T \end{pmatrix},
\end{aligned}$$

where we define $\tilde{\rho}_k = R_k(s,s)$. This gives us an expression for $\underline{T}_k$:

$$\begin{aligned}
\underline{T}_k &= \underline{R}_k\underline{B}_k R_k - \beta_{sk+1}e_1 e_s^T R_{k-1,k}R_k^{-1} \\
&= \begin{pmatrix} R_k B_k R_k^{-1} + \tilde{\rho}_k^{-1} b_k z_k e_s^T - \beta_{sk+1}e_1 e_s^T R_{k-1,k}R_k^{-1} \\ \tilde{\rho}_k^{-1}\rho_k b_k e_s^T \end{pmatrix},
\end{aligned} \tag{4.9}$$

which we can separate into an expression for T_k:

$$T_k = R_k B_k R_k^{-1} + \tilde{\rho}_k^{-1} b_k z_k e_s^T - \beta_{sk+1}e_1 e_s^T R_{k-1,k}R_k^{-1}, \tag{4.10}$$

and an expression for $\beta_{s(k+1)+1} = \underline{T}_k(s+1, s)$:

$$\beta_{s(k+1)+1} = \frac{\rho_k}{\tilde{\rho}_k} b_k. \tag{4.11}$$

It would be interesting to prove structurally that T_k, the $s \times s$ upper submatrix of the lower block entry

$$\underline{T}_k = \underline{R}_k \underline{B}_k R_k^{-1} - \beta_{sk+1} e_1 e_s^T R_{k-1,k} R_k^{-1}$$

of $\underline{\mathfrak{T}}_k$, is symmetric. We already know that it must be, via the Lanczos process on a symmetric matrix A. It's important to identify, though, what assumptions about orthogonality we might be making in forcing T_k to be symmetric. Certainly $\underline{B}_k$ need not be symmetric – it may be any upper Hessenberg matrix with nonzero subdiagonal – so it seems we have to compute $\underline{T}_k$ as a sum of products of dense (or at least triangular or Hessenberg) matrices. These are all of dimension $s \times s$, $s+1$ by $s+1$, or $s+1$ by s, though, so the cost of the dense products is not much compared with the cost of assembling the tridiagonal matrix in standard Lanczos iteration. Furthermore, in CA-Lanczos we have the flexibility to treat T_k as dense rather than as tridiagonal, and see if that improves accuracy.

4.2.3 The CA-Lanczos algorithm

Algorithm 25 shows the final CA-Lanczos algorithm. It avoids communication in the sparse matrix operations, by using the matrix powers kernel (Section 2.1) in Line 4. It also avoids communication in the dense vector operations. Instead of using dot products and AXPYs as in the standard Lanczos recurrence (Equation (4.2) in Section 4.1), it uses a combination of TSQR (Lines 7 and 14) and a BGS update (Lines 12 and 13). This requires a factor of $\Theta(s)$ more storage and arithmetic, but the same number of messages and the same amount of memory traffic as standard Lanczos.

4.3 Preconditioned CA-Lanczos

In this section, we derive communication-avoiding symmetric Lanczos (CA-Lanczos) algorithms with (symmetric) split preconditioning and left preconditioning. We do this not because we want to precondition Lanczos iteration; that is not typically done.[2] Rather, we discuss "preconditioned Lanczos" for two reasons. The first is that we can use it to solve the generalized eigenvalue problem $Ax = \lambda Mx$. That problem can be converted into a "preconditioned eigenvalue problem," in any of the following ways:

- If M is nonsingular, solving the left-preconditioned eigenvalue problem $M^{-1}Ax = \lambda x$ is equivalent to solving the generalized eigenvalue problem $Ax = \lambda Mx$.
- If M is nonsingular, solving the right-preconditioned eigenvalue problem $AM^{-1}x = \lambda x$ is equivalent to solving the generalized eigenvalue problem $Ay = \lambda My$ and the linear system $y = M^{-1}x$.

[2] Balancing could be considered a type of preconditioning for eigenvalue problems, but symmetric matrices do not need balancing.

Algorithm 25 CA-Lanczos

Input: $n \times n$ symmetric matrix A and $n \times 1$ starting vector r

1: $\beta_0 := \|r\|_2$, $q_1 := r/\beta_0$
2: **for** $k = 0$ to $t - 1$ **do**
3: Fix basis conversion matrix $\underline{B}_k$
4: Compute $\underline{\acute{V}}_k$ from q_{sk+1} and A using matrix powers kernel
5: **if** $k = 0$ **then**
6: $\underline{\mathfrak{B}}_0 := \underline{B}_0$
7: Compute (via TSQR) the QR factorization $\underline{V}_0 = \underline{Q}_0 \underline{R}_0$
8: $\underline{\mathfrak{Q}}_0 := \underline{Q}_k$, $\underline{\mathfrak{R}}_0 := \underline{R}_0$
9: $\underline{T}_0 := \underline{R}_0 \underline{B}_0 R_0^{-1}$, $\underline{\mathfrak{T}}_0 := \underline{T}_0$
10: **else**
11: $\underline{\mathfrak{B}}_k := \begin{pmatrix} \mathfrak{T}_{k-1} & 0 \cdot e_1 e_s^T \\ \beta_{sk+1} e_1 e_{s(k-1)}^T & \underline{B}_k \end{pmatrix}$
12: $\underline{\acute{R}}_{k-1,k} := \underline{Q}_{k-1}^* \underline{\acute{V}}_k$
13: $\underline{\acute{V}}_k := \underline{\acute{V}}_k - \underline{Q}_{k-1} \underline{\acute{R}}_{k-1,k}$
14: Compute (via TSQR) the QR factorization $\underline{\acute{V}}_k = \underline{\acute{Q}}_k \underline{\acute{R}}_k$
15: $\underline{\mathfrak{Q}}_k := [\underline{\mathfrak{Q}}_{k-1}, \underline{\acute{Q}}_k]$
16: Compute T_k (Eq. (4.10)) and $\beta_{s(k+1)+1}$ (Eq. (4.11))
17: $\underline{\mathfrak{T}}_k := \begin{pmatrix} \mathfrak{T}_{k-1} & \beta_{sk+1} e_{s(k-1)} e_1^T \\ \beta_{sk+1} e_1 e_{s(k-1)}^T & T_k \\ 0_{1,s(k-1)} & \beta_{s(k+1)+1} e_s^T \end{pmatrix}$
18: **end if**
19: **end for**

- If M is nonsingular with factorization $M = M_1 M_2$, solving the eigenvalue problem $M_1^{-1} A M_2^{-1} x = \lambda x$ is equivalent to solving the generalized eigenvalue problem $Ay = \lambda My$ and the linear system $y = M_2^{-1} x$.

We do not discuss this application of preconditioned Lanczos further in this thesis. The second reason we develop preconditioned CA-Lanczos is because the algorithm will serve as a template for deriving other preconditioned communication-avoiding Krylov methods. For example, we will use the approach of this section to derive left-preconditioned communication-avoiding CG in Section 5.5, and to point out how communication-avoiding versions of nonsymmetric Lanczos and BiCG may be derived (see Chapter 6). We will therefore develop left-preconditioned and split-preconditioned communication-avoiding versions of Lanczos iteration (CA-Lanczos) in this section. The derivation of right-preconditioned CA-Lanczos is similar enough to the left-preconditioned version that we do not discuss it further in this thesis. Furthermore, we assume in this section that the split preconditioner is symmetric, for reasons that we will explain later.

As we mentioned in Sections 4.2, to make Lanczos "communication-avoiding" means reducing the amount of memory traffic, and the number of messages in parallel, for both the sparse matrix operations and the dense vector operations. We do so by using a preconditioned form of the matrix powers kernel (Section 2.2) to replace the sparse-matrix vector products

and preconditioner applications, along with some combination of Block Gram-Schmidt (Section 2.4) and either TSQR or CholeskyQR (both described in Section 2.3) to orthogonalize the basis vectors. The way we do this for preconditioned CA-Lanczos depends on whether it is left-preconditioned or symmetrically split-preconditioned. This is because the form of the preconditioner affects both the form of the matrix powers kernel, and the methods that may be used to orthogonalize the Krylov basis vectors. We discuss this in detail below, but in brief, it relates to whether the *preconditioned matrix* is symmetric. (In the case of left preconditioning with preconditioner M, the preconditioned matrix is $M^{-1}A$, and in the case of (symmetric) split preconditioning with preconditioner $M = LL^*$, the preconditioned matrix is $L^{-1}AL^{-*}$.) For example, symmetric split-preconditioned CA-Lanczos may use a combination of TSQR and BGS to orthogonalize its Krylov basis vectors, just as in non-preconditioned CA-Lanczos (Algorithm 25 in Section 4.2). Left-preconditioned CA-Lanczos must use a variant of CholeskyQR instead of TSQR (see Section 2.3 as well as Algorithm 2 in Section 1.2.5). It also requires two different matrix powers kernels, which we will present below.

Our preconditioned CA-Lanczos algorithms originate in s-step Krylov methods, just as all the other communication-avoiding Krylov methods in this thesis do. However, we are the first, as far as we know, to derive a preconditioned version of s-step symmetric Lanczos, and to suggest using it to solve generalized eigenvalue problems. We are also the first to avoid communication in both the sparse matrix and dense vector operations in symmetric Lanczos.

We are also the first, as far as we can tell, to suggest equilibration as a way to control s-step basis condition number growth. (See Section 7.5.3 for details.) Finally, we are the first, as far as we know, to show the different matrix powers kernel required for left and right preconditioning CG and symmetric Lanczos, when the matrix A is symmetric, but the preconditioned matrix is not.

In Section 4.3.1, we give the standard versions of split-preconditioned, left-preconditioned, and right-preconditioned Lanczos iteration. We also summarize the connection between the three-term Lanczos recurrence and orthogonal polynomials. In Section 4.3.2, we use this connection to show why s-step versions of left- and right-preconditioned Lanczos require a different derivation than unpreconditioned and symmetric split-preconditioned Lanczos. Sections 4.3.3 and 4.3.4 then show our development of left-preconditioned CA-Lanczos, which we do in two phases: one in which we avoid communication only when computing the sparse matrix-vector products and preconditioner applications, and one in which we avoid communication both there and in the vector-vector operations. We defer many of the details of those derivations until Section 4.3.5, which readers will only find useful if they want to check our formulae.

4.3.1 Lanczos and orthogonal polynomials

We derive the split-preconditioned and left-preconditioned communication-avoiding Lanczos algorithms by working with the orthogonal polynomials closely associated with the Lanczos process. Gautschi [109] and Meurant [176] explain the relationship between the polynomials, the Lanczos coefficients, and the inner product in which the Lanczos basis vectors are orthogonal.

Unpreconditioned symmetric Lanczos (Algorithm 24 in Section 4.1) is based on the three-term polynomial recurrence

$$\beta_{k+1} q_{k+1}(t) = t \cdot q_k(t) - \alpha_k q_k(t) - \beta_k q_{k-1}(t), \tag{4.12}$$

in which the polynomials $q_k(t)$ are orthogonal with respect to a certain inner product. In unpreconditioned Lanczos, this inner product is the usual Euclidean one, and the recurrence translates into matrix-vector terms as

$$\beta_{k+1} q_{k+1} = A q_k - \alpha_k q_k - \beta_k q_{k-1}. \tag{4.13}$$

This translation is possible because the matrix A is symmetric.[3] This means that $\langle u, Av \rangle = \langle Au, v \rangle$ for all u and v, so that a matrix-vector product $A \cdot v$ corresponds directly to multiplying a polynomial $p(t)$ by the variable t. (The aforementioned authors explain this correspondence more formally.) This property makes the three-term recurrence (4.12) work; otherwise, some recurrence may hold, but it may be longer than three terms. (The Arnoldi recurrence is the most general example, where the recurrence is of maximum possible length.) There is an extensive theory of the matrix properties required for a recurrence of a certain length ≥ 3 to exist. (This is generally called "Faber-Manteuffel theory" from these authors' 1984 work [95], though Voevodin in 1983 produced the earliest known proof that there is no short-recurrence version of CG [234].) For our purposes, the three-term recurrence suffices, since matrices that have longer but still bounded recurrences have very special properties that are too expensive to check in practice.

For a preconditioned iterative method, the matrix on which the iterative method is performed is no longer the original matrix A, but the preconditioned matrix. In split-preconditioned Lanczos, the preconditioned matrix is $L^{-T} A L^T$, and the three-term recurrence becomes

$$\beta_{k+1} q_{k+1} = L^{-T} A L^{-1} q_k - \alpha_k q_k - \beta_k q_{k-1}. \tag{4.14}$$

Since the preconditioned matrix is symmetric, the recurrence still works as before. Thus, the q_k vectors are still orthogonal with respect to the usual Euclidean inner product.

Algorithm 26 shows standard left-preconditioned symmetric Lanczos iteration, and Algorithm 27 shows the right-preconditioned version of symmetric Lanczos. We chose Greenbaum's notation for the latter; she shows how it can be derived from split-preconditioned Lanczos, if one assumes that the preconditioner is SPD [120, p. 121]. Unlike in the split-preconditioned version, in left-preconditioned Lanczos, the preconditioned matrix $M^{-1}A$ is not necessarily self-adjoint with respect to the usual (Euclidean) inner product, even if M and A are both symmetric. That is, there may exist u and v such that $\langle u, M^{-1}Av \rangle \neq \langle M^{-1}Au, v \rangle$. Similarly, the right-preconditioned matrix AM^{-1} is also not necessarily self-adjoint with respect to the usual inner product. However, the left-preconditioned matrix $M^{-1}A$ is self-adjoint with respect to a different inner product, namely the so-called M *inner product* $\langle u, v \rangle_M := v^* M u$. For example,

$$\begin{aligned} \langle u, M^{-1}Av \rangle_M &= v^* A M^{-1} \cdot Mu &&= v^* Au = \langle u, Av \rangle \\ \langle M^{-1}Au, v \rangle_M &= v^* M \cdot M^{-1} Au &&= v^* Au = \langle u, Av \rangle \end{aligned}$$

[3] We label as "symmetric," here as elsewhere in this thesis, both real symmetric matrices and complex Hermitian matrices. All the algorithms shown should work in either the complex case or the real case, unless specified.

Algorithm 26 Left-preconditioned symmetric Lanczos iteration

Require: r is the starting vector

$q_0 := 0$, $z_0 := 0$
$z_1 := r$, $q_1 := M^{-1} z_1$
$\beta_1 := \sqrt{\langle q_1, z_1 \rangle}$
for $k = 1$ to convergence **do**
    $w := A q_k - \beta_k z_{k-1}$
    $\alpha_k := \langle w, q_k \rangle$
    $w := w - \alpha_k z_k$
    $z := M^{-1} w$
    $\beta_{k+1} := \sqrt{\langle w, z \rangle}$
    $q_{k+1} := w / \beta_{k+1}$
    $z_{k+1} := z / \beta_{k+1}$
end for

Algorithm 27 Right-preconditioned symmetric Lanczos iteration

Require: r is the starting vector

$q_0 := 0$, $z_0 := 0$
$z := M^{-1} r$
$\beta_0 := \sqrt{\langle r, w \rangle}$
$q_1 := r / \beta_1$
$z_1 := z / \beta_1$
for $k = 1$ to convergence **do**
    $v_k := A z_k - \beta_k q_{k-1}$
    $\alpha_k := \langle v_k, z_k \rangle$
    $w := w - \alpha_k q_k$
    $z := M^{-1} w$
    $\beta_{k+1} := \sqrt{\langle w, z \rangle}$
    $q_{k+1} := w / \beta_{k+1}$
    $z_{k+1} := z / \beta_{k+1}$
end for

This means that the usual three-term recurrence holds

$$\beta_{k+1} q_{k+1} = M^{-1} A q_k - \alpha_k q_k - \beta_k q_{k-1},$$

but the q_k are *M-orthogonal*: $\langle q_i, q_j \rangle_M = \langle q_i, M q_j \rangle = \delta_{ij}$. An analogous property holds for right-preconditioned Lanczos:

$$\beta_{k+1} q_{k+1} = A M^{-1} q_k - \alpha_k q_k - \beta_k q_{k-1},$$

in which the vectors q_k are M^{-1}-orthogonal. From now on, we will restrict our discussion to left-preconditioned Lanczos, with the understanding that analogous approaches apply to the right-preconditioned version of the algorithm.

4.3.2 Left preconditioning excludes QR

In left-preconditioned Lanczos, the basis vectors q_k are not necessarily orthogonal in the Euclidean sense. That means we cannot generate them directly by means of a matrix factorization based on orthogonal transformations, such as Householder QR or TSQR. Standard left-preconditioned Lanczos produces the basis vectors one at a time by means of a process equivalent to Modified Gram-Schmidt (MGS) orthogonalization in the M inner product. Each inner product requires a global reduction, so if we choose this approach in our s-step version of Lanczos, we will lose the property that the number of messages is independent of s.

Preconditioned conjugate gradient has the same problem with its residual vectors: In unpreconditioned and split-preconditioned CG, the residual vectors are orthogonal, but in left-preconditioned CG, the residual vectors are M-orthogonal, and in right-preconditioned CG, the residual vectors are M^{-1}-orthogonal. This is because the residual vectors in CG are just scaled versions of the symmetric Lanczos basis vectors (see e.g., [120]). Chronopoulos and Gear avoided this problem by generating a (monomial) basis for the $M^{-1}A$-conjugate vectors p_k in preconditioned conjugate gradient, computing their Gram matrix, and using its Cholesky factorization to reconstruct the p_k vectors themselves [57]. Computing the Gram matrix sacrifices stability because the Gram matrix's condition number is the square of the condition number of the basis itself. For an already poorly conditioned basis, this may mean that the Gram matrix is numerically rank-deficient. This may make it impossible to use the Gram matrix to make the basis vectors M-orthogonal. Nevertheless, it is not clear how to make the basis vectors M-orthogonal without computing the Gram matrix, so we have chosen this method when developing our algorithms.

4.3.3 Left-preconditioned CA-Lanczos with MGS

In this section and the next, we will derive the fully communication-avoiding version of left-preconditioned Lanczos. We do so in two steps. In this section, we replace all the sparse matrix-vector products with invocations of the matrix powers kernel, but leave the inner products for computing the Lanczos coefficients. These inner products belong to a kind of modified Gram-Schmidt orthogonalization procedure in the M inner product. The resulting algorithm, CA-Lanczos with MGS, avoids communication in the sparse matrix-vector products, but not in the inner products. In Section 4.3.4, we will replace the inner products with quantities obtained from the Gram matrix of the basis vectors from the matrix powers kernel. We call the resulting algorithm CA-Lanczos without MGS, or just "CA-Lanczos." The latter algorithm attains our goal, namely avoiding communication in both the sparse matrix-vector products and the inner products.

We omit many of the details of our derivations in this section and the next. These details await the particularly persistent reader in Section 4.3.5. It is not necessary to read that section in order to understand our CA-Lanczos algorithms, however.

Ansatz

Suppose we start left-preconditioned symmetric Lanczos using the $n \times n$ matrix A, the preconditioner M^{-1}, and the starting vector v. The symmetric Lanczos process produces

two sets of basis vectors: q_1, q_2, ..., q_{s+1} and z_1, z_2, ..., z_{s+1}. These vectors satisfy the following properties

1. $z_1 = v$ (the unpreconditioned starting vector);
2. $q_i = M^{-1} z_i$ for all $i \geq 1$;
3. for $1 \leq j \leq s+1$, $\text{span}\{q_1, \ldots, q_j\} = \text{span}\{q_1, (M^{-1}A)q_1, \ldots, (M^{-1}A)^j q_1\}$
4. for $1 \leq j \leq s+1$, $\text{span}\{z_1, \ldots, z_j\} = \text{span}\{z_1, (AM^{-1})z_1, \ldots, (AM^{-1})^{j-1} z_{sk+1}\}$;
5. $\langle q_i, q_j \rangle_M = \langle q_i, z_j \rangle = \delta_{ij}$ (the q_i vectors are M-orthogonal); and
6. $\langle z_i, z_j \rangle_{M^{-1}} = \langle q_i, z_j \rangle = \delta_{ij}$ (the z_i vectors are M^{-1}-orthogonal).

Preconditioned Lanczos needs two bases because the preconditioned matrix $M^{-1}A$ is not symmetric, but it is self-adjoint with respect to the M inner product $\langle u, v \rangle_M$, as well as with respect to the M^{-1} inner product $\langle u, v \rangle_{M^{-1}}$. Rather than computing these inner products directly, symmetric Lanczos relies on Properties 5–6 to perform Gram-Schmidt orthogonalization in order to generate the two sets of basis vectors.

This suggests we should also use two sets of basis vectors in communication-avoiding left-preconditioned symmetric Lanczos. We use the matrix powers kernel to construct two bases:

- $v_1, \ldots, v_{s+1}$ as a basis for $\text{span}\{q_1, (M^{-1}A)q_1, \ldots, (M^{-1}A)^s q_1\}$, and
- $w_1, \ldots, w_{s+1}$ as a basis for $\text{span}\{z_1, (AM^{-1})z_1, \ldots, (AM^{-1})^s z_1\}$.

These bases satisfy Properties 1–4, but not Properties 5–6. We then construct the "Gram matrix" $[v_1, \ldots, v_{s+1}]^*[w_1, \ldots, w_{s+1}]$ and use it to reconstruct the original Lanczos basis vectors, thus transforming the v_i vectors into q_i vectors and the w_i vectors into z_i vectors.

Notation

Our notation is similar to that we presented for Arnoldi(s, t) in Section 3.3.2. Each outer iteration k of our communication-avoiding preconditioned Lanczos will require two different n by $s+1$ basis matrices $\underline{V}_k$ and $\underline{W}_k$, with

$$\underline{V}_k = [v_{sk+1}, v_{sk+2}, \ldots, v_{sk+s}, v_{sk+s+1}] = [V_k, v_{sk+s+1}]$$

and

$$\underline{W}_k = [w_{sk+1}, w_{sk+2}, \ldots, w_{sk+s}, w_{sk+s+1}] = [W_k, w_{sk+s+1}] .$$

These two matrices are produced by one invocation of the matrix powers kernel. Note that the superscript of $\underline{V}_k$ and $\underline{W}_k$ refers to the outer iteration, whereas the superscript of a single basis vector (such as v_{sk+1}) refers to the total number of inner iterations (across all the outer iterations of the algorithm).

For $j = 1, 2, \ldots, s+1$, the n by $s+1$ basis matrix $\underline{V}_k$ satisfies

$$\mathcal{R}\left(\underline{V}_k(:, 1:j)\right) = \operatorname{span}\{q_{sk+1}, (M^{-1}A)q_{sk+1}, (M^{-1}A)^2 q_{sk+1}, \ldots, (M^{-1}A)^{j-1} q_{sk+1}\}, \quad (4.15)$$

and for $j = 1, 2, \ldots, s+1$, the n by $s+1$ basis matrix $\underline{W}_k$ satisfies

$$\mathcal{R}\left(\underline{W}_k(:, 1:j)\right) = \operatorname{span}\{z_{sk+1}, AM^{-1}z_{sk+1}, (AM^{-1})^2 z_{sk+1}, \ldots, (AM^{-1})^{j-1} z_{sk+1}\}. \quad (4.16)$$

(We will see more about the two subspaces in the above equations.) Furthermore, we choose these bases so that $M^{-1}w_{sk+j} = v_{sk+j}$ for $j = 1, \ldots, s+1$.

As with Arnoldi(s) in Section 3.2.2, we refer to the $s+1$ by s change of basis matrix $\underline{B}_k$, where k refers to the current outer iteration number. The basis (and therefore the change of basis matrix $\underline{B}_k$) may be different for each outer iteration k. In left-preconditioned CA-Lanczos, we have two sets of basis vectors; for simplicity, we require that they satisfy the following:

$$M^{-1}AV_k(:, 1:j) = \underline{V}_k(:, 1:j+1)\underline{B}_k(1:j+1, 1:j) \quad (4.17)$$

and

$$AM^{-1}W_k(:, 1:j) = \underline{W}_k(:, 1:j+1)\underline{B}_k(1:j+1, 1:j) \quad (4.18)$$

for $j = 1, \ldots, s$.

We also will need notation for four other basis matrices. Suppose we are at outer iteration $k \geq 0$ and inner iteration j (with $1 \leq j \leq s$). If r is the starting vector of Lanczos, then the vectors $z_1, z_2, \ldots, z_{sk+j}$ form a basis of the Krylov subspace $\mathcal{K}_{sk+j}(AM^{-1}, r)$, as long as the Lanczos process does not break down. (These z_i vectors are the same in exact arithmetic as in the usual left-preconditioned Lanczos (Algorithm 26).) In case $k = 0$, we set $z_{sk} = 0$ and in general $z_i = 0$ for $i \leq 0$. Then we define the $n \times s$ basis matrix Z_k:

$$Z_k := [z_{sk+1}, \ldots, z_{sk+s}]$$

and the n by $s+1$ basis matrix $\underline{Z}_k$:

$$\underline{Z}_k := [z_{sk+1}, \ldots, z_{sk+s}, z_{sk+s+1}] = [Z_k, z_{sk+1}].$$

Here, the underline has the same implication as the underline in $\underline{V}_k$: "append a vector to the end." Similarly, we define the $n \times s$ basis matrix Q_k:

$$Q_k := [q_{sk+1}, \ldots, q_{sk+s}]$$

and the n by $s+1$ basis matrix $\underline{Q}_k$:

$$\underline{Q}_k := [q_{sk+1}, \ldots, q_{sk+s+1}] = [Q_k, q_{sk+1}].$$

We have $M^{-1}Z_k = Q_k$ by definition. Also, it is helpful to define $q_i = 0$ and $z_i = 0$ for $i \leq 0$.

Deducing the algorithm

We will deduce what we need by starting with the usual left-preconditioned symmetric Lanczos algorithm. We borrow notation from Section 3.3 (the Arnoldi(s, t) algorithm) whenever possible, and clarify when we introduce new notation or change the meaning of existing notation.

We begin the algorithm *in media res*, at outer iteration k and at inner iteration j. The outer iteration index k increases starting at 0, and the inner iteration index j ranges from 1 up to and including s. Here as in the rest of this work, the fixed parameter s governs the number of vectors generated by the matrix powers kernel.

If we were running the standard Lanczos process instead, the corresponding iteration index to outer iteration k and inner iteration j would be $sk + j$. There, the usual left-preconditioned Lanczos recurrences take this form:

$$\begin{aligned} \beta_{sk+j+1} q_{sk+j+1} &= M^{-1} A q_{sk+j} - \alpha_{sk+j} q_{sk+j} - \beta_{sk+j} q_{sk+j-1}, \\ \beta_{sk+j+1} z_{sk+j+1} &= A q_{sk+j} - \alpha_{sk+j} z_{sk+j} - \beta_{sk+j} z_{sk+j-1}, \end{aligned} \tag{4.19}$$

where the q vectors are the M-orthogonal Lanczos basis vectors, and $z_i = M q_i$ for all $i > 0$. The complicated subscripts let us compare the standard Lanczos process with our communication-avoiding version.

Suppose we want to compute α_{sk+j}, β_{sk+j+1}, q_{sk+j+1}, and z_{sk+j+1}. In the standard Lanczos process, we would do the following:

1: $u := A q_{sk+j} - \beta_{sk+j} z_{sk+j-1}$
2: $\alpha_{sk+j} := \langle u, q_{sk+j} \rangle$
3: $u := u - \alpha_{sk+j} z_{sk+j}$
4: $v := M^{-1} u$
5: $\beta_{sk+j+1} := \sqrt{\langle u, v \rangle}$
6: $z_{sk+j+1} := u / \beta_{sk+j+1}$
7: $q_{sk+j+1} := v / \beta_{sk+j+1}$

In communication-avoiding Lanczos, we are not free to compute $A q_{sk+j}$ or $M^{-1} u$ directly, because these involve matrix-vector products that are not part of the matrix powers kernel. Assembling these two vectors from the output of the matrix powers kernel is the topic of this section. If we wish not to perform inner products as well, then we must also think of other ways to compute α_{sk+j} and β_{sk+j+1}. This is the topic of Section 4.3.4.

In order to compute the Lanczos coefficient α_{sk+j}, we need $A q_{sk+j}$. The coefficient α_{sk+j} satisfies

$$\begin{aligned} \alpha_{sk+j} &= \langle M^{-1} A q_{sk+j} - \beta_{sk+j} q_{sk+j-1}, q_{sk+j} \rangle_M \\ &= \langle A q_{sk+j} - \beta_{sk+j} M q_{sk+j-1}, q_{sk+j} \rangle \\ &= \langle A q_{sk+j} - \beta_{sk+j} z_{sk+j-1}, q_{sk+j} \rangle. \end{aligned}$$

Given $A q_{sk+j}$, we could compute α_{sk+j} with one inner product. It turns out (see Lemma 5 in Section 4.3.5 for the proof) that for $k \geq 0$ and $1 \leq j \leq s$, there is a vector of $2s + 1$ coefficients d_{sk+j} such that

$$A q_{sk+j} = [Z_{k-1}, \underline{W}_k] d_{sk+j}, \tag{4.20}$$

and that d_{sk+j} can always be computed from known scalar quantities. Given this vector of

coefficients, we can compute α_{sk+j} directly from

$$\alpha_{sk+j} = \langle [Z_{k-1}, \underline{W}_k] d_{sk+j} - \beta_{sk+j} z_{sk+j-1}, q_{sk+j} \rangle.$$

The next Lanczos coefficient to find is

$$\beta_{sk+j+1} = \langle Aq_{sk+j} - \alpha_{sk+j} z_{sk+j} - \beta_{sk+j} z_{sk+j-1}, \\ M^{-1} Aq_{sk+j} - \alpha_{sk+j} q_{sk+j} - \beta_{sk+j} q_{sk+j-1} \rangle^{1/2}.$$

It also turns out (see Lemma 6 in Section 4.3.5 for the proof) that for $k \geq 0$ and $1 \leq j \leq s$, the same vector of $2s+1$ coefficients d_{sk+j} as above satisfies

$$M^{-1} Aq_{sk+j} = [Q_{k-1}, \underline{V}_k] d_{sk+j}.$$

Then we could compute β_{sk+j+1} via

$$\beta_{sk+j+1} = \langle [Z_{k-1}, \underline{W}_k] d_{sk+j} - \alpha_{sk+j} z_{sk+j} - \beta_{sk+j} z_{sk+j-1}, \\ [Q_{k-1}, \underline{V}_k] d_{sk+j} - \alpha_{sk+j} q_{sk+j} - \beta_{sk+j} q_{sk+j-1} \rangle^{1/2},$$

q_{sk+j+1} via

$$q_{sk+j+1} = \beta_{sk+j+1}^{-1} \left([Q_{k-1}, \underline{V}_k] d_{sk+j} - \alpha_{sk+j} q_{sk+j} - \beta_{sk+j} q_{sk+j-1} \right),$$

and z_{sk+j+1} via

$$z_{sk+j+1} = \beta_{sk+j+1}^{-1} \left([Z_{k-1}, \underline{W}_k] d_{sk+j} - \alpha_{sk+j} z_{sk+j} - \beta_{sk+j} z_{sk+j-1} \right).$$

That gives us everything we need for one iteration of left-preconditioned Lanczos. Algorithm 28 summarizes the resulting iteration, in which we use the superscript in parentheses to denote the outer iteration number.

Now we have expressions for the next group of Lanczos coefficients as well as the next Lanczos vector, in terms of dot products of known vectors and recurrence relations involving known coefficients. In this subsection, the Lanczos coefficients α_k and β_{k+1} were computed using the same modified Gram-Schmidt process as in standard Lanczos iteration; the only difference is the computation of intermediate vector terms, which come from recurrence relations. This means we can restrict any stability analysis to the recurrence relations alone, and then reconstruct the stability of the whole algorithm from what we know about the usual Lanczos process. However, we have only managed to avoid communication in the matrix-vector products and preconditioner applications, not in the orthogonalization process. In the next subsection, we will avoid communication in the inner products of the orthogonalization as well.

4.3.4 Left-preconditioned CA-Lanczos without MGS

In the previous subsection, we managed to avoid communication in computing Lanczos vectors, but not in the inner products. Now we will handle the inner products in a process that we call *inner product coalescing*.

Algorithm 28 Left-preconditioned CA-Lanczos with MGS

Input: $A \in \mathbb{C}^{n \times n}$: SPD resp. HPD matrix
Input: $r \in \mathbb{C}^n$: starting vector

$q_0 := 0$, $z_0 := 0$
$z_1 := r$, $q_1 := M^{-1} z_1$
$\alpha_0 := 0$, $\beta_0 := 0$
for $k = 0, 1, \ldots$ until convergence **do**
    Compute via matrix powers kernel $\underline{V}_k = [v_{sk+1}, v_{sk+2}, \ldots, v_{sk+s+1}]$. Its columns are a basis for $\text{span}\{q_{sk+1}, M^{-1}Aq_{sk+1}, \ldots, (M^{-1}A)^s q_{sk+1}\}$.
    Compute via matrix powers kernel $\underline{W}_k = [w_{sk+1}, w_{sk+2}, \ldots, w_{sk+s+1}]$. Its columns are a basis for $\text{span}\{z_{sk+1}, AM^{-1}z_{sk+1}, \ldots, (AM^{-1})^s z_{sk+1}\}$ and it also satisfies $M^{-1}\underline{W}_k = \underline{V}_k$.
    for $j = 1$ to s **do**
        Using Equation (4.33) in Section 4.3.5, compute $d_{sk+j} \in \mathbb{C}^{2s+1}$ such that $Aq_{sk+j} = [Z_{k-1}, \underline{W}_k] d_{sk+j}$ and $M^{-1}Aq_{sk+j} = [Q_{k-1}, \underline{V}_k] d_{sk+j}$
        $z := [Z_{k-1}, \underline{W}_k] d_{sk+j}$
        $z := z - \beta_{sk+j} z_{sk+j-1}$
        $\alpha_{sk+j} := \langle z, q_{sk+j} \rangle$
        $z := z - \alpha_{sk+j} z_{sk+j}$
Assert: $z = Aq_{sk+j} - \alpha_{sk+j} z_{sk+j} - \beta_{sk+j} z_{sk+j-1}$
        $q := [Q_{k-1}, \underline{V}_k] d_{sk+j}$
        $q := q - \alpha_{sk+j} q_{sk+j} - \beta_{sk+j} q_{sk+j-1}$
Assert: $q = M^{-1}Aq_{sk+j} - \alpha_{sk+j} q_{sk+j} - \beta_{sk+j} q_{sk+j-1}$
        $\beta_{sk+j+1} := \sqrt{\langle z, q \rangle}$
        $z_{sk+j+1} := z / \beta_{sk+j+1}$
        $q_{sk+j+1} := q / \beta_{sk+j+1}$
    end for
end for

Deducing the algorithm

Consider first the computation of α_{sk+j} as shown above:

$$\alpha_{sk+j} = \langle [Z_{k-1}, \underline{W}_k] d_{sk+j} - \beta_{sk+j} z_{sk+j-1}, q_{sk+j} \rangle.$$

It turns out (see Lemmas 7 and 8 in Section 4.3.5 for the proof) that for $k > 0$ and $j = 1, 2, \ldots, s$, we can find a vector of coefficients $g_{sk+j} \in \mathbb{C}^{2s+1}$ such that $z_{sk+j} = [Z_{k-1}, \underline{W}_k] g_{sk+j}$ and $q_{sk+j} = [Q_{k-1}, \underline{V}_k] g_{sk+j}$, and that g_{sk+j} can always be computed from known scalar quantities. This means that instead of computing α_{sk+j} via the above dot product, we can compute this coefficient using the Gram matrix of the basis vectors. We begin with

$$\begin{aligned}
\alpha_{sk+j} = \langle Aq_{sk+j} - \beta_{sk+j} z_{sk+j-1}, q_{sk+j} \rangle = \\
\langle [Z_{k-1}, \underline{W}_k] d_{sk+j} - \beta_{sk+j} [Z_{k-1}, \underline{W}_k] g_{sk+j-1}, [Q_{k-1}, \underline{V}_k] g_{sk+j} \rangle = \\
g_{sk+j}^* [Q_{k-1}, \underline{V}_k]^* [Z_{k-1}, \underline{W}_k] \left(d_{sk+j} - \beta_{sk+j} g_{sk+j-1} \right).
\end{aligned}$$

We define the Gram matrix G_k by

$$G_k = [Q_{k-1}, \underline{V_k}]^* [Z_{k-1}, \underline{W_k}] = \begin{pmatrix} Q_{k-1}^* Z_{k-1} & Q_{k-1}^* \underline{W_k} \\ \underline{V_k}^* Z_{k-1} & \underline{V_k}^* \underline{W_k} \end{pmatrix}. \tag{4.21}$$

Note that (assuming that $M^* = M$)

$$(Q_{k-1}^* \underline{W_k})^* = \underline{W_k}^* Q_{k-1} = \underline{V_k}^* M^* M^{-1} Z_{k-1} = V_k^* Z_{k-1},$$

so we only need to compute one of the $(1,2)$ and $(2,1)$ blocks of G_k. Furthermore, if we assume that the q_i vectors are M-orthogonal, so that $q_i^* z_j = \delta_{ij}$, then

$$G_k = \begin{pmatrix} I_{s \times s} & Q_{k-1}^* \underline{W_k} \\ (Q_{k-1}^* \underline{W_k})^* & \underline{V_k}^* \underline{W_k} \end{pmatrix}. \tag{4.22}$$

This suggests computing G_k via the block inner product

$$\text{Block}(1:2,2)_k = [Q_{k-1}, \underline{V_k}]^* \underline{W_k}, \tag{4.23}$$

which is the product of an n by $2s+1$ matrix and an n by $s+1$ matrix. In parallel over P processors, this computation requires only $O(\log P)$ messages.

The orthogonality assumption matters because in practice, rounding error causes rapid loss of orthogonality in the Lanczos basis vectors, unless some kind of reorthogonalization is done. Reorthogonalization is expensive, but computing $Q_{k-1}^* Z_{k-1}$ (a kind of partial reorthogonalization) is less expensive and a possible alternative in this case. This would change the block inner product used to compute G_k from that in Equation (4.23). If the goal is to minimize the number of messages at any cost, either a single block product that computes $Q_{k-1}^* \underline{W_k}$ and $(Q_{k-1}^* \underline{W_k})^*$ redundantly, or two block inner products

$$\text{Block}(1:2,2)_k = [Q_{k-1}, \underline{V_k}]^* \underline{W_k},$$

and

$$\text{Block}(1,1)_k = Q_{k-1}^* Z_{k-1},$$

could be used. These two block inner products may be computed simultaneously.

Regardless of how we compute the Gram matrix G_k, we can use G_k to recover α_{sk+j} via the formula

$$\alpha_{sk+j} = g_{sk+j}^* G_k \left(d_{sk+j} - \beta_{sk+j} g_{sk+j-1} \right). \tag{4.24}$$

Note that we are still using the modified Gram-Schmidt process to calculate α_{sk+j}: we do not use the mathematical identity $\alpha_{sk+j} = \langle A q_{sk+j}, q_{sk+j} \rangle = \langle [Z_{k-1} \underline{W_k}] d_{sk+j}, [Q_{k-1}, \underline{V_k} g_{sk+j} \rangle$, which reduces to the classical Gram-Schmidt process. In essence, Equation (4.24) projects the Lanczos vectors onto a lower-dimensional space in order to compute the Lanczos coefficient.

We can follow an analogous process to compute β_{sk+j+1}. We have

$$|\beta_{sk+j+1}|^2 = \langle A q_{sk+j} - \alpha_{sk+j} z_{sk+j} - \beta_{sk+j} z_{sk+j-1} \\ M^{-1} A q_{sk+j} - \alpha_{sk+j} q_{sk+j} - \beta_{sk+j} q_{sk+j-1} \rangle,$$

and so, if we let

$$\xi_{sk+j} = d_{sk+j} - \alpha_{sk+j} g_{sk+j} - \beta_{sk+j} g_{sk+j-1}, \tag{4.25}$$

then

$$|\beta_{sk+j+1}|^2 = \langle [Z_{k-1}, \underline{W}_k] \xi_{sk+j}, [Q_{k-1}, \underline{V}_k] \xi_{sk+j} \rangle = \xi_{sk+j}^* G_k \xi_{sk+j}. \tag{4.26}$$

Note that our development of Equation (4.26) does *not* apply the mathematical identity

$$|\beta_{sk+j+1}|^2 = \left\langle M^{-1} A q_{sk+j}, A q_{sk+j} \right\rangle - |\alpha_{sk+j}|^2 - |\beta_{sk+j}|^2.$$

Using this identity to compute β_{sk+j+1} could result in catastrophic cancellation, as it would compute a quantity that should be positive using subtractions of quantities that are possibly close in value. Our formula also uses subtractions, but it is equivalent to the M inner product of two identical vectors. Furthermore, the Gram matrix G_k is guaranteed to be positive definite as long as the Lanczos process has not broken down and the basis vectors are not too ill-conditioned. This makes the above formula for β_{sk+j+1} robust. Also, we can determine whether CA-Lanczos has broken down by testing whether G_k is positive definite; this can be done without communication in $O(s^3)$ time, as long as G_k fits in fast memory.

We call the above computations *inner product coalescing*, because they coalesce $2s$ inner products (to compute $2s$ Lanczos coefficients from s Lanczos vectors) into one or two block inner products. Each block inner product requires only one communication step, for which the number of messages is the same as that of a single inner product, and the total number of words sent and received is no more than $\Theta(s)$ times more than for those $2s$ inner products. Algorithm 29 shows the complete communication-avoiding left-preconditioned Lanczos procedure.

4.3.5 Details of left-preconditioned CA-Lanczos

In this section, we work out some details of left-preconditioned CA-Lanczos that we omitted explaining in Sections 4.3.3 and 4.3.4. In particular, we show how to construct the "intermediate" vectors Aq_{sk+j}, $M^{-1}Aq_{sk+j}$, (from Section 4.3.3), z_{sk+j}, and q_{sk+j} (from Section 4.3.4) without extra communication, using previously computed quantities in the algorithm.

We begin in Section 4.3.5 by proving four lemmas showing which sets of vectors we need in order to compute these intermediate quantities. Then, in Section 4.3.5, we derive recurrences that enable us to construct the intermediate vectors out of those sets and other previously computed coefficients.

Spaces containing intermediate vectors

In this section, we prove four lemmas. Lemma 1 shows us what vectors we need to construct Aq_{sk+j} and Lemma 2 shows which vectors are needed to construct $M^{-1}Aq_{sk+j}$. Both lemmas are required in Section 4.3.3. Lemma 3 shows which vectors are needed to construct z_{sk+j}, and Lemma 4 shows which vectors are needed to construct q_{sk+j}. The latter two lemmas are required in Section 4.3.4.

Lemma 1. *For $k > 0$ and $j = 1, 2, 3, \ldots, s$, the vector Aq_{sk+j} in CA-Lanczos satisfies*

$$Aq_{sk+j} \in \text{span}\{w_{sk+j+1}, \ldots, w_{sk+2}, w_{sk+1}\} \oplus \text{span}\{z_{sk}, \ldots, z_{sk-j+1}\}. \tag{4.27}$$

Algorithm 29 Left-preconditioned CA-Lanczos without MGS

Input: $A \in \mathbb{C}^{n \times n}$: SPD resp. HPD matrix
Input: $r \in \mathbb{C}^n$: starting vector

1: $q_0 := 0$, $z_0 := 0$
2: $z_1 := r$, $q_1 := M^{-1} z_1$
3: $\alpha_0 := 0$, $\beta_0 := 0$
4: **for** $k = 0, 1, \ldots$ until convergence **do**
5: Compute via matrix powers kernel $\underline{V}_k = [v_{sk+1}, v_{sk+2}, \ldots, v_{sk+s+1}]$. Its columns are a basis for $\text{span}\{q_{sk+1}, M^{-1}Aq_{sk+1}, \ldots, (M^{-1}A)^s q_{sk+1}\}$.
6: Compute via matrix powers kernel $\underline{W}_k = [w_{sk+1}, w_{sk+2}, \ldots, w_{sk+s+1}]$. Its columns are a basis for $\text{span}\{z_{sk+1}, AM^{-1}z_{sk+1}, \ldots, (AM^{-1})^s z_{sk+1}\}$ and it also satisfies $M^{-1}\underline{W}_k = \underline{V}_k$.
7: Compute the Gram matrix G_k (see Equation (4.21))
8: **for** $j = 1$ to s **do**
9: Using Equation (4.33) in Section 4.3.5, compute d_{sk+j} such that $Aq_{sk+j} = [Z_{k-1}, \underline{W}_k] d_{sk+j}$ and $M^{-1}Aq_{sk+j} = [Q_{k-1}, \underline{V}_k] d_{sk+j}$
10: Using Equation (4.34) in Section 4.3.5, compute g_{sk+j} such that $z_{sk+j} = [Z_{k-1}, \underline{W}_k] g_{sk+j}$ and $q_{sk+j} = [Q_{k-1}, \underline{V}_k] g_{sk+j}$
11: $\alpha_{sk+j} := g_{sk+j}^* G_k \left(d_{sk+j} - \beta_{sk+j} g_{sk+j-1}\right)$ (see Equation (4.24))
12: $\xi_{sk+j} := d_{sk+j} - \alpha_{sk+j} g_{sk+j} - \beta_{sk+j} g_{sk+j-1}$ (see Equation (4.25))
13: $\beta_{sk+j+1} := \xi_{sk+j}^* G_k \xi_{sk+j}$ (see Equation (4.26))
14: $z := [Z_{k-1}, \underline{W}_k] d_{sk+j} - \alpha_{sk+j} z_{sk+j} - \beta_{sk+j} z_{sk+j-1}$
15: $q := [Q_{k-1}, \underline{V}_k] d_{sk+j} - \alpha_{sk+j} q_{sk+j} - \beta_{sk+j} q_{sk+j-1}$
16: $q_{sk+j+1} := w / \beta_{sk+j+1}$
17: $z_{sk+j+1} := z / \beta_{sk+j+1}$
18: **end for**
19: **end for**

The same holds for $k = 0$ if we define $z_i = 0$ for $i \leq 0$.

Proof. For $j = 1$, we have by the properties of the w_i basis vectors that

$$\begin{aligned} Aq_{sk+1} &= AM^{-1} z_{sk+1} = [w_{sk+1}, w_{sk+2}] B_k(1:2, 1) \\ &\in \text{span}\{w_{sk+1}, w_{sk+2}\}. \end{aligned}$$

For $j = 2$, we have $\beta_{sk+2} z_{sk+2} = AM^{-1} z_{sk+1} - \alpha_{sk+1} z_{sk+1} - \beta_{sk+1} z_{sk}$, and therefore

$$\begin{aligned} Aq_{sk+2} &= AM^{-1} z_{sk+2} \\ &\in \text{span}\{z_{sk}, AM^{-1} w_{sk+1}, (AM^{-1})^2 w_{sk+1}\} \\ &= \text{span}\{z_{sk}, [w_{sk+1}, w_{sk+2}] B_k(1:2,1), (AM^{-1})[w_{sk+1}, w_{sk+2}] B_k(1:2,1)\} \\ &= \text{span}\{z_{sk}, [w_{sk+1}, w_{sk+2}] B_k(1:2,1), [w_{sk+1}, w_{sk+2}, w_{sk+3}] B_k(1:3,2) B_k(1:2,1)\} \\ &= \text{span}\{z_{sk}, w_{sk+1}, w_{sk+2}, w_{sk+3}\}. \end{aligned}$$

Now assume $2 < j \leq s$. From the Lanczos recurrence,

$$\beta_{sk+j+1} z_{sk+j} = Aq_{sk+j-1} - \alpha_{sk+j} z_{sk+j-1} - \beta_{sk+j} z_{sk+j-2}.$$

Now multiply both sides by AM^{-1}:

$$\beta_{sk+j+1} Aq_{sk+j} = AM^{-1} Aq_{sk+j-1} - \alpha_{sk+j} Aq_{sk+j-1} - \beta_{sk+j} Aq_{sk+j-2}.$$

First of all,

$$AM^{-1} Aq_{sk+j-1} \in \text{span}\{Aq_{sk+j}, Aq_{sk+j-1}, Aq_{sk+j-2}\}.$$

By the inductive hypothesis,

$$Aq_{sk+j-2}, Aq_{sk+j-1} \in \text{span}\{w_{sk+j}, \ldots, w_{sk+1}\} \oplus \text{span}\{z_{sk}, \ldots, z_{sk-j+2}\}.$$

If we multiply z_{sk-j+2} on the left by AM^{-1} and apply the Lanczos recurrence, we get $AM^{-1} z_{sk-j+2} \in \text{span}\{z_{sk-j+1}, z_{sk-j+2}, z_{sk-j+3}\}$. This gives us the desired result, namely that

$$Aq_{sk+j} \in \text{span}\{w_{sk+j+1}, \ldots, w_{sk+1}\} \oplus \text{span}\{z_{sk+1}, z_{sk}, \ldots, z_{sk-j+1}\}$$

for $j = 2, 3, \ldots, s$. □

Lemma 2. *For $k > 0$ and $j = 1, 2, 3, \ldots, s$, the vector $M^{-1} Aq_{sk+j}$ in CA-Lanczos satisfies*

$$M^{-1} Aq_{sk+j} \in \text{span}\{v_{sk+j+1}, \ldots, v_{sk+1}\} \oplus \text{span}\{q_{sk}, q_{sk-1} \ldots, q_{sk-j+1}\}. \tag{4.28}$$

The same holds for $k = 0$ if we define $z_i = 0$ for $i \leq 0$.

Proof. Observe that $z_i = Mq_i$ and $w_i = Mv_i$ for all i, and apply Lemma 1. □

Lemma 3. *For $k > 0$ and $1 \leq j \leq s$, the Lanczos basis vector z_{sk+j} satisfies*

$$z_{sk+j} \in \text{span}\{w_{sk+j}, w_{sk+j-1}, \ldots, w_{sk+2}, w_{sk+1}\} \oplus \text{span}\{z_{sk}, \ldots, z_{sk-j+2}\}. \tag{4.29}$$

The same holds for $k = 0$ if we define $z_i = 0$ for $i \leq 0$.

Proof. The case $j = 1$ is trivially true: $z_{sk+1} = w_{sk+1}$ by definition. Consider the case $j = 2$. The z_{sk+j} vectors satisfy the Lanczos three-term recurrence, so

$$\begin{aligned} \beta_{sk+2} z_{sk+2} &= AM^{-1} z_{sk+1} - \alpha_{sk+1} z_{sk+1} - \beta_{sk+1} z_{sk} \\ &= [w_{sk+1}, w_{sk+2}] B_k(1:2, 1) - \alpha_{sk+1} z_{sk+1} - \beta_{sk+1} z_{sk} \\ &\in \text{span}\{w_{sk+2}, w_{sk+1}, z_{sk}\}. \end{aligned}$$

Now consider the case $2 < j \leq s$. The z_{sk+j} vectors satisfy the Lanczos recurrence

$$\beta_{sk+j} z_{sk+j} = AM^{-1} z_{sk+j-1} - \alpha_{sk+j-1} z_{sk+j-1} - \beta_{sk+j-1} z_{sk+j-2}.$$

Lemma 1 tells us that

$$AM^{-1} z_{sk+j-1} \in \text{span}\{w_{sk+j}, \ldots, w_{sk+2}, w_{sk+1}\} \oplus \text{span}\{z_{sk}, \ldots, z_{sk-j+2}\}.$$

The desired result follows from this and induction on z_{sk+j-2} and z_{sk+j-1}. □

Lemma 4. *For $k > 0$ and $1 \leq j \leq s$, the Lanczos basis vector q_{sk+j} satisfies*

$$q_{sk+j} \in \text{span}\{v_{sk+j}, v_{sk+j-1}, \ldots, v_{sk+1}\} \oplus \text{span}\{q_{sk+1}, q_{sk}, \ldots, q_{sk-j+2}\}. \tag{4.30}$$

The same holds for $k = 0$ if we define $z_i = 0$ for $i \leq 0$.

Proof. Recall that $z_i = Mq_i$ and $w_i = Mv_i$ for all i, and apply Lemma 3. □

Construction of intermediate vectors

In this section, we prove four lemmas which explain how to compute various vectors from previously computed vectors and coefficients. These lemmas are the computational analogues of the theoretical lemmas presented in Section 4.3.5; they show how to compute the vectors out of known data. This section may be of interest to implementers of CA-Lanczos and CA-CG, as well as to those who want to check the numerical stability of the recursion formulae.

We will begin by defining some matrices of basis vectors. Let

$$\begin{aligned} Q_k &= [q_{sk+1}, q_{sk+2}, \ldots, q_{sk+s}] \text{ and} \\ \underline{Q}_k &= [Q_k, q_{sk+s+1}], \end{aligned}$$

and let

$$\begin{aligned} Z_k &= [z_{sk+1}, z_{sk+2}, \ldots, z_{sk+s}] \text{ and} \\ \underline{Z}_k &= [Z_k, z_{sk+1}]. \end{aligned}$$

Note that $\underline{Z}_k(:, 1:j) = M\underline{Q}_k(:, 1:j)$ for $1 \leq j \leq s+1$. We know that for $1 \leq j \leq s+1$,

$$\mathcal{R}\left(\underline{Q}_k(:, 1:j)\right) = \text{span}\{M^{-1}z_{sk+1}, (M^{-1}A)M^{-1}z_{sk+1}, \ldots, (M^{-1}A)^{k-1}M^{-1}z_{sk+1}\}, \quad (4.31)$$

and therefore for $1 \leq j \leq s+1$,

$$\begin{aligned} \mathcal{R}\left(\underline{Z}_k(:, 1:j)\right) &= \mathcal{R}\left(M\underline{Q}_k(:, 1:j)\right) \\ &= \text{span}\{z_{sk+1}, (AM^{-1})z_{sk+1}, \ldots, (AM^{-1})^{j-1}z_{sk+1}\}. \end{aligned} \quad (4.32)$$

Note also that for $1 \leq j \leq s+1$,

$$\begin{aligned} \mathcal{R}\left(\underline{V}_k(:, 1:j)\right) &= \mathcal{R}\left(\underline{Q}_k(:, 1:j)\right) \text{ and} \\ \mathcal{R}\left(\underline{W}_k(:, 1:j)\right) &= \mathcal{R}\left(\underline{Z}_k(:, 1:j)\right). \end{aligned}$$

We also have the useful properties that

$$\begin{aligned} M^{-1}AV_k(:, 1:j) &= \underline{V}_k(:, 1:j+1)\underline{B}_k(1:j+1, 1:j) \text{ and} \\ AM^{-1}W_k(:, 1:j) &= \underline{W}_k(:, 1:j+1)\underline{B}_k(1:j+1, 1:j) \end{aligned}$$

for $1 \leq j \leq s$. In case $k = 0$, we set $q_{sk} = 0$ and $z_{sk} = 0$.

Lemma 5. *For $k > 0$ and $j = 0, 1, 2, \ldots, s$, we can find a vector of coefficients $d_{sk+j} \in \mathbb{C}^{2s+1}$ such that*

$$Aq_{sk+j} = [Z_{k-1}, \underline{W}_k]d_{sk+j}.$$

Furthermore, only elements $s-j+1$ to $s+j+1$ (inclusive) of the $2s+1$ elements of d_{sk+1} are nonzero. This lemma also holds for $k = 0$ if we set $Z_{-1} = 0_{n \times s}$.

Proof. For $j = 0$, we have $Aq_{sk} = \beta_{sk+1}z_{sk+1} + \alpha_{sk}z_{sk} + \beta_{sk}z_{sk-1}$, and therefore

$$d_{sk} = (0_{s-2}, \beta_{sk}, \alpha_{sk}, \beta_{sk+1}, 0_s)^T.$$

For $j = 1$, we have $Aq_{sk+1} = AM^{-1}w_{sk+1} = w_{sk+1}B_k(1,1) + w_{sk+2}B_k(2,1)$, and therefore

$$d_{sk+1} = (0_s, B_k(1,1), B_k(2,1), 0_{s-1})^T.$$

For $j = 2, 3, \ldots, s$, we have by Lemma 1 that

$$\begin{aligned} Aq_{sk+j} &\in \text{span}\{z_{sk+1}, w_{sk+1}, \ldots, w_{sk+j}\} \oplus \text{span}\{z_{sk-j+1}, \ldots, z_{sk}\} \\ &\in \mathcal{R}\left(Z_{k-1}\right) \oplus \mathcal{R}\left(\underline{W}_k\right). \end{aligned}$$

The maximum value of j under consideration is s, so the least index of the z_i vectors required to express all the Aq_{sk+j} is $i = sk-s+1 = s(k-1)+1$. Furthermore, only elements $s-j+1$ to $s+j+1$ (inclusive) of d_{sk+j} are nonzero. By the Lanczos recurrence, for $j = 2, \ldots, s$, we have

$$\beta_{sk+j}z_{sk+j} = Aq_{sk+j-1} - \alpha_{sk+j-1}z_{sk+j-1} - \beta_{sk+j-1}z_{sk+j-2}$$

and therefore

$$\beta_{sk+j}z_{sk+j} = [Z_{k-1}, \underline{W}_k]d_{sk+j-1} - \alpha_{sk+j-1}z_{sk+j-1} - \beta_{sk+j-1}z_{sk+j-2}.$$

Multiply both sides by AM^{-1} on the left:

$$\beta_{sk+j}AM^{-1}z_{sk+j} = \begin{cases} AM^{-1}[Z_{k-1}, \underline{W}_k]d_{sk+j-1} - \\ \alpha_{sk+j-1}AM^{-1}z_{sk+j-1} - \\ \beta_{sk+j-1}AM^{-1}z_{sk+j-2}, \end{cases}$$

$$\beta_{sk+j}Aq_{sk+j} = \begin{cases} AM^{-1}[Z_{k-1}, \underline{W}_k]d_{sk+j-1} - \\ \alpha_{sk+j-1}Aq_{sk+j-1} - \\ \beta_{sk+j-1}Aq_{sk+j-2}, \end{cases}$$

$$\beta_{sk+j}[Z_{k-1}, \underline{W}_k]d_{sk+j} = \begin{cases} AM^{-1}[Z_{k-1}, \underline{W}_k]d_{sk+j-1} - \\ \alpha_{sk+j-1}[Z_{k-1}, \underline{W}_k]d_{sk+j-1} - \\ \beta_{sk+j-1}[Z_{k-1}, \underline{W}_k]d_{sk+j-2}. \end{cases}$$

Since at least the first and last entries of d_{sk+j-1} are zero,

$$\begin{aligned} AM^{-1}[Z_{k-1}, \underline{W}_k]d_{sk+j-1} &= [AQ_{k-1}, \underline{W}_k\underline{B}_k]d_{sk+j-1} \\ &= [\underline{Z}_{k-1}\underline{T}_{k-1}, \underline{W}_k\underline{B}_k]d_{sk+j-1}. \end{aligned}$$

Therefore, for $j = 2, 3, \ldots, s$,

$$\begin{aligned} \beta_{sk+j}[Z_{k-1}, \underline{W}_k]d_{sk+j} = [\underline{Z}_{k-1}\underline{T}_{k-1}, \underline{W}_k\underline{B}_k]d_{sk+j-1} \\ - \alpha_{sk+j-1}[Z_{k-1}, \underline{W}_k]d_{sk+j-1} - \beta_{sk+j-1}[Z_{k-1}, \underline{W}_k]d_{sk+j-2}. \end{aligned}$$

By matching up identical vector terms, this gives us a recurrence for the entries of d_{sk+j}. We show the final result below as Equation (4.33).

$$d_{sk+j} = \begin{cases} (0_{s-2}, \beta_{sk}, \alpha_{sk}, \beta_{sk+1}, 0_s)^T, & j = 0, \\ (0_s, \underline{B}_k(1,1), \underline{B}_k(2,1), 0_{s-1})^T, & j = 1, \\ \begin{cases} \beta_{sk+j}^{-1}(\underline{T}_{k-1}d_{sk+j-1}(1:s), 0_s)^T \\ +\beta_{sk+j}^{-1}(0_s, \underline{B}_k d_{sk+j-1}(s+2:2s+1))^T \\ -\beta_{sk+j}^{-1}\left(\alpha_{sk+j-1}d_{sk+j-1} + \beta_{sk+j-1}d_{sk+j-2}\right) \end{cases} & 2 \le j \le s. \end{cases} \tag{4.33}$$

□

Lemma 6. *For $k > 0$ and $j = 0, 1, 2, \ldots, s$, the vector of coefficients $d_{sk+j} \in \mathbb{C}^{2s+1}$ from Lemma 5 also satisfies*

$$M^{-1}Aq_{sk+j} = [Q_{k-1}, \underline{V_k}]d_{sk+j}.$$

Proof. See proof of Lemma 2. □

Lemma 7. *For $k > 0$ and $j = 1, 2, \ldots, s$, we can find a vector of coefficients $g_{sk+j} \in \mathbb{C}^{2s+1}$ such that*

$$z_{sk+j} = [Z_{k-1}, \underline{W_k}]g_{sk+j}.$$

Furthermore, only elements $s - j + 2$ to $s + j - 1$ (inclusive) of the $2s + 1$ elements of g_{sk+j} are nonzero. This lemma also holds for $k = 0$ if we set $Z_{-1} = 0_{n\times s}$.

Proof. For $j = 1$, the lemma holds trivially with $g_{sk+1} = (0_s, 1, 0_s)^T$. For $j = 2, 3, \ldots, s$, we have by Lemma 3 that

$$\begin{aligned} z_{sk+j} &\in \operatorname{span}\{w_{sk+j-1}, w_{sk+j-2}, \ldots, w_{sk+1}\} \oplus \operatorname{span}\{z_{sk}, \ldots, z_{sk-j+2}\} \\ &\in \mathcal{R}\left(\underline{W_k}\right) \oplus \mathcal{R}\left(Z_{k-1}\right). \end{aligned}$$

The maximum value of j under consideration is s, so the least index of the z_i vectors required to express all the z_{sk+j} is $i = sk - s + 2 = s(k-1) + 2$. Furthermore, the first $s - j + 1$ elements of g_{sk+j} as well as the last $s - j + 1$ elements of g_{sk+j} are all zero.

Here, we show how to compute g_{sk+j} for $j > 1$. As mentioned above, for $j = 1$ we have $g_{sk+1} = (0_s, 1, 0_s)^T$. For $j = 2$, we have

$$\begin{aligned} \beta_{sk+2}z_{sk+2} &= AM^{-1}z_{sk+1} - \alpha_{sk+1}z_{sk+1} - \beta_{sk+1}z_{sk} \\ &= [w_{sk+1}, w_{sk+2}]B_k(1:2,1) - \alpha_{sk+1}w_{sk+1} - \beta_{sk+1}z_{sk} \end{aligned}$$

and therefore

$$g_{sk+2} = \beta_{sk+2}^{-1}(0_{s-1}, -\beta_{sk+1}, B_k(1,1) - \alpha_{sk+1}, B_k(2,1), 0_{s-1})^T.$$

For $3 \le j \le s + 1$, we proceed as follows. Start with the usual Lanczos recurrence for the z_i vectors:

$$\beta_{sk+j}z_{sk+j} = Aq_{sk+j-1} - \alpha_{sk+j-1}z_{sk+j-1} - \beta_{sk+j-1}z_{sk+j-2}$$

and apply Lemma 5 to Aq_{sk+j-1}:

$$\begin{aligned} &\beta_{sk+j}[Z_{k-1}, \underline{W_k}]g_{sk+j} = \\ &\qquad [Z_{k-1}, \underline{W_k}]d_{sk+j-1} - \alpha_{sk+j-1}[Z_{k-1}, \underline{W_k}]g_{sk+j-1} - \beta_{sk+j-1}[Z_{k-1}, \underline{W_k}]g_{sk+j-2}. \end{aligned}$$

This gives us a recursion formula for the g_{sk+j} coefficients:

$$g_{sk+j} = \begin{cases} (0_s, 1, 0_s)^T, & j = 1, \\ \beta_{sk+2}^{-1}(0_{s-1}, -\beta_{sk+1}, B_k(1,1) - \alpha_{sk+1}, B_k(2,1), 0_{s-1})^T, & j = 2, \\ \beta_{sk+j}^{-1}\left(d_{sk+j-1} - \alpha_{sk+j-1}g_{sk+j-1} - \beta_{sk+j-1}g_{sk+j-2}\right), & 3 \le j \le s+1. \end{cases} \tag{4.34}$$

□

Lemma 8. *For $k > 0$ and $j = 1, 2, \ldots, s$, the vector of coefficients $g_{sk+j} \in \mathbb{C}^{2s+1}$ from Lemma 7 also satisfies*

$$q_{sk+j} = [Q_{k-1}, \underline{V_k}]g_{sk+j}.$$

Proof. See proof of Lemma 4. □

Chapter 5

Communication-avoiding CG

In this chapter, we will develop a communication-avoiding version of the conjugate gradient (CG) method for solving SPD linear systems $Ax = b$. We call this algorithm Communication-Avoiding CG (CA-CG). The parallel version of CA-CG requires a factor of $\Theta(s)$ fewer messages than standard CG, where s is the number of basis vectors generated by the matrix powers kernel. The sequential version of CA-CG reads the sparse matrix A a factor of s fewer times than standard CG. Both algorithms come at an additional memory cost: they require $2s$ vectors to be stored, rather than 4 (as in standard CG). We develop both nonpreconditioned and preconditioned versions of CA-CG. As far as we know, we are the first to develop communication-avoiding versions of CG that can be preconditioned, that work with an arbitrary s-step basis, and that avoid communication both in parallel and between levels of the memory hierarchy.

We begin this chapter with Section 5.1, where we show the standard CG algorithm. In Section 5.2, we summarize prior work on Communication-Avoiding CG. In Section 5.3, we try to make the most commonly taught version of the CG algorithm communication-avoiding. The attempt shows us that we should start with a different variant of CG, the "three-term recurrence variant" we call CG3. We use CG3 to develop CA-CG in Section 5.4. In Section 5.5, we use a similar approach to construct a left-preconditioned version of CA-CG, called LP-CA-CG. Finally, we summarize in Section 5.6.

5.1 Conjugate gradient method

The conjugate gradient method (CG) is used to solve symmetric positive definite linear systems $Ax = b$. Its authors Hestenes and Stiefel [131] conceived of it as a direct method for solving linear systems, but it has proven much more useful as an iterative method. It has since been understood within the framework of Krylov subspace methods as mathematically equivalent to the symmetric Lanczos process. See e.g., Greenbaum [120] or Saad [208], who explain what it means for these methods to be mathematically equivalent. There are many different versions of CG that are equivalent in exact arithmetic; Saad [208] presents two of them. Algorithm 30 shows the formulation of CG that most people learn in textbooks. There, the r_j vectors are computed residuals that in exact arithmetic are the Lanczos orthogonal basis vectors, scaled by constant factors. The p_j vectors are basis vectors that are

Algorithm 30 Conjugate gradient method

Input: $Ax = b$: $n \times n$ SPD linear system
Input: x_1: initial guess
Output: Approximate solution x_j of $Ax = b$

1: $r_1 := b - Ax_1$
2: $p_1 := r_1$
3: **for** $j = 1, 2, \ldots,$ until convergence **do**
4: Compute $w := Ap_j$
5: $\alpha_j := \langle r_j, r_j \rangle / \langle w, p_j \rangle$
6: $x_{j+1} := x_j + \alpha_j p_j$
7: $r_{j+1} := r_j - \alpha_j w$
8: $\beta_j := \langle r_{j+1}, r_{j+1} \rangle / \langle r_j, r_j \rangle$
9: $p_{j+1} := r_{j+1} + \beta_j p_j$
10: Test convergence (e.g., using β_j)
11: **end for**

A-conjugate, which is another way to say A-orthogonal: with respect to the inner product defined by $\langle u, v \rangle_A := v^* A u$, the vectors $p_1, \ldots, p_j$ are orthogonal.

5.2 Prior work

In this Section, we describe prior work relating to our CA-CG algorithm. These include algorithms with the specific goal of avoiding communication, as well as algorithms with other goals that are nevertheless predecessors of CA-CG. We devote Sections 5.2.1 and 5.2.2 to the two important predecessors of CA-CG: "s-step CG" by Chronopoulos and Gear [57], and "Krylov basis CG" by Toledo [222].

The very first reference we found to algorithms like our CA-CG is a work by Van Rosendale [228]. Working in a PRAM parallel performance model [101], Van Rosendale "unrolls" the CG iteration in order to eliminate the data dependencies between the inner products (to compute the coefficients), the AXPY vector operations (which use the coefficients), and the sparse matrix-vector products (which take as input the results of the previous iteration's vector operations). The resulting algorithm, when given an $n \times n$ sparse linear system $Ax = b$, requires only $\max\{\log(d), \log(\log(n))\}$ iterations to run on n processors (in the PRAM model). The "unrolled" algorithm is more or less an s-step version of CG using the monomial basis, where Van Rosendale sets s to the total number of iterations needed to converge. Van Rosendale discovered that the algorithm is not numerically unstable [229], which one may surmise is due to a combination of using the monomial basis, and the basis length s being too large. We can surmise the latter also by the doctoral thesis of Leland [166], which cites Van Rosendale [228] and sets $s = \log n$ in order to maximize parallelism under the PRAM model. He also discovered experimentally that the algorithm is not stable.

Van Rosendale did recognize that the data dependencies in standard CG limited potential parallelism. However, his reliance on the PRAM model prevented him from developing a practical performance model of the algorithm. The implementation lacked optimized kernels such as the matrix powers kernel, and was not numerically stable.

5.2.1 s-step CG

Our CA-CG algorithm was inspired in part by an algorithm by Chronopoulos and Gear called "s-step CG" [57]. Here, s is the s-step basis length, just as in CA-Lanczos (Section 4.2), CA-GMRES (Section 3.4), and CA-Arnoldi (Section 3.3). Unlike those algorithms, however, the s-step CG algorithm does not include restarting, so the authors did not provide a restart length parameter. (In CA-Lanczos, CA-GMRES, and CA-Arnoldi, the t parameter is the number of outer iterations, so that the restart length is $s \cdot t$. Restarting CA-CG is unnecessary for performance or memory capacity reasons, as neither the storage requirements nor the computational requirements increase as the algorithm progresses.)

Here is a list of differences between the s-step CG algorithm of Chronopoulos and Gear, and our CA-CG algorithm.

- Chronopoulos and Gear base s-step CG on Algorithm 30, the standard formulation of CG. We will base our CA-CG algorithm on the three-term recurrence variant of CG, Algorithm 32.

- The two authors do not use an optimized matrix powers kernel to compute the Krylov basis, although they do recognize that an optimized version of the kernel might exploit parallelism better than the usual CG. We exploit the full range of matrix powers kernel optimizations described in Sections 2.1 (for unpreconditioned CA-CG in Section 5.4) and 2.2 (for preconditioned CA-CG in Section 5.5).

- Chronopoulos and Gear compute a Gram matrix involving the basis vectors (see e.g., Section 5.4.5 for a definition of Gram matrix) in order to compute the coefficients in s-step CG. Our CA-CG also computes a Gram matrix, but unlike Chronopoulos and Gear's algorithm, CA-CG does not perform matrix subtractions in order to compute the Gram matrix. This ensures that the Gram matrix is always SPD as long as the s-step basis is sufficiently well-conditioned, so that the positive definiteness of the matrix A is preserved under projection onto the Krylov basis. This helps preserve numerical stability.

- The two authors use a monomial basis which spans the same space as the A-conjugate vectors p_j in standard CG (Algorithm 30). As we explain in Chapter 7, the monomial basis limits them to small values of s. Otherwise the iterations diverge, since the s-step "basis" is no longer a basis; it is no longer numerically full rank. In fact, Chronopoulos and Gear observe that $s = 5$ is the upper limit for numerical stability in the examples they tested. Later, Chronopoulos and other collaborators would use modified Gram-Schmidt orthogonalization, at significant computational expense, to improve the condition number of the s-step basis (see e.g., [60]). In our CA-CG algorithm, we have several choices of s-step basis (as described in Chapter 7), which allow us to use larger basis lengths s without sacrificing numerical stability, and also without requiring expensive orthogonalization. Larger s mean greater potential for performance improvements, since CA-CG communicates a factor $\Theta(s)$ less than standard CG.

- Chronopoulos and Gear were only able to use preconditioners that commuted with the matrix A, such as polynomial preconditioners, or block diagonal preconditioners (that

require no parallel communication) [56]. Our methods do not have this limitation, as we discuss in Section 5.5.

5.2.2 Krylov basis CG

Toledo constructed an algorithm called "Krylov basis CG" [222], which was inspired by Chronopoulos and Gear's s-step CG. He begins with a kernel like the matrix powers kernel, and a sequential algorithm implementing it that saves communication between levels of the memory hierarchy. The algorithm works especially well for low-dimensional meshes. It also applies to matrices whose graph structure has nonlocal connections, although the benefits in that case are less. We discuss the latter in the context of multigrid in Section 1.6.4. While the algorithm does work for general sparse matrices, Toledo only implemented it for tridiagonal matrices, and considered the task of generalizing this implementation quite difficult. (He was not mistaken, as our collaborators discovered when implementing the matrix powers kernel for general sparse matrices.)

Toledo applies the kernel in order to avoid communication in his Krylov basis CG algorithm. He also observes that computing the Gram matrix instead of performing the inner products individually saves communication, which is an observation that previous authors either did not make or did not clearly state. Toledo proceeds much as we do in this section. First, he constructs a communication-avoiding version of CG for $s = 2$, called the "ClusteredDotProdsCG" algorithm [222, p. 117]. We do something similar in Section 5.3, although unlike him, we eventually use a different approach to develop CA-CG. Then, Toledo generalizes the algorithm into "KrylovBasisConjugateGradient" [222, p. 119].

The following list summarizes the differences between Toledo's work and ours.

- Toledo does have an optimized matrix powers kernel algorithm that can handle general sparse matrices, but his implementation only works for tridiagonal matrices. We have both algorithms and implementations of the matrix powers kernel for general sparse matrices (Section 2.1) and even include preconditioning (Section 2.2), which Toledo does not.

- Krylov basis CG requires about twice as many vectors as our CA-CG algorithm, and twice as many invocations of the matrix powers kernel. This is because he computes an s-step basis using both the current residual vector, and the current A-conjugate vector. Like Chronopoulos and Gear, Toledo bases his algorithm on the standard version of CG (Algorithm 30), which has both residual vectors and A-conjugate vectors. Since we base CA-CG on the three-term recurrence variant of CG (Algorithm 32), we only need to compute and store an s-step basis for the current residual vector.

- Toledo does suggest the use of different s-step bases other than the monomial basis, in order to improve stability and allow larger basis lengths s [222, p. 122]. We do the same, for the same reasons. However, Toledo does not show the necessary changes to the matrix powers kernel if preconditioning is included, which we do show (see Section 5.5).

- Toledo has an idea for an s-step basis which we do not explore in this thesis: using polynomials whose local components are orthogonal [222, pp. 123–4]. This means first

partitioning the matrix of s-step basis vectors into block rows, commensurate with the partitioning of the sparse matrix into subdomains for the matrix powers kernel. Then, as the matrix powers kernel computes each s-step basis vector in turn, the components of that vector in each block row are orthogonalized against the previous columns of that block row. This requires no communication, since all of the orthogonalization operations happen inside each subdomain. The resulting s-step basis vectors are not orthogonal, but this process supposedly makes them better conditioned. Toledo claims that doing this lets him use very large basis lengths s [222, p. 123]. However, he does not show numerical or performance results for this process. Furthermore, the resulting basis polynomials no longer satisfy a three-term recurrence, which increases the amount of fast memory required by the matrix powers kernel and thus may increase the number of block transfers between levels of the memory hierarchy (see Section 2.1).

- When Toledo uses the monomial basis, he suggests a clever technique to represent the Gram matrix implicitly via the FFT [222, p. 120]. This saves floating-point operations in the small dense matrix-vector operations used to compute the CG coefficients. The corresponding operations in our Communication-Avoiding CG (Algorithm 35 in Section 5.4.5) require $\Theta(s^2)$ operations per outer iteration; in Toledo's algorithm, they require only $\Theta(s \log s)$ operations. Of course, this does not affect the asymptotic number of flops, if one assumes (as we do) that $s^2 \ll n$.

- Like Chronopoulos and Gear, Toledo also suggests using polynomial preconditioning [222, p. 126–7]. In particular, since Chebyshev preconditioning can sometimes perform badly, even if the spectral bounds are accurate, he suggests using the polynomial preconditioning technique of Fisher and Freund [98], which exploits spectral information estimated from the CG coefficients. However, he does not test preconditioning either in numerical experiments or for performance, nor does he show how preconditioning would change the Krylov basis CG algorithm. Our left-preconditioned communication-avoiding CG (LP-CA-CG) (Algorithm 37 in Section 5.5) is able to avoid communication with a larger class of preconditioners (via the techniques of Section 2.2) than those that Toledo mentions in [222].

5.2.3 Summary

The main issues with previous communication-avoiding versions of CG were

1. Lack of effective matrix powers kernel
2. Limited to monomial basis (especially if preconditioning)
3. Limited preconditioning options

We have addressed issue #1 with algorithms and implementations in Demmel et al. [77, 78, 79]. Toledo (in the context of CG – other authors have introduced other bases for s-step GMRES) have addressed #2 by suggesting the use of s-step bases other than the monomial basis. However, they do not show how to use bases other than the monomial basis when including preconditioning, which we do show how to do in this thesis.

As far as we can tell, no authors dealt completely with issue #3. Either they do not show how to include preconditioning in the s-step algorithm, or they limit themselves either to block diagonal preconditioners (that require no communication) or preconditioners that commute with the matrix A (such as polynomial preconditioners). In [77], we first characterized the structure of effective preconditioners compatible with the matrix powers kernel. We summarize those results in Section 2.2. In this work, we show how to apply left preconditioning to CA-CG with preconditioners that need not commute with the matrix A.

Note that our innovation with CA-GMRES (Section 3.4) of being able to choose the s-step basis length s independently of the restart length $r = s \cdot t$ does not apply to CA-CG. Unlike CA-Arnoldi and CA-GMRES, the storage and computational requirements of CA-CG do not increase as the algorithm progresses, so it is not necessary to consider restarting.

We develop our communication-avoiding versions of CG directly from corresponding versions of the Lanczos process. For nonpreconditioned or split-preconditioned CG, we present our communication-avoiding CG (CA-CG) algorithm in Section 5.4. In Section 5.5, we develop a left-preconditioned version of CA-CG, using an analogous derivation approach to that of Sections 4.3.3 and 4.3.4 for left-preconditioned CA-Lanczos.

5.3 Communication-avoiding CG: first attempt

The standard formulation of CG (Algorithm 30) involves two coupled recurrences: one for the residuals r_j, and one for the A-conjugate basis vectors p_j. In this section, we will attempt to rearrange this formulation into an s-step method. The attempt fails, but this failure leads us to a different variant of CG which is amenable to such rearrangements, which we will present in Section 5.4. The conclusion is that trying to combine the A-conjugate vectors p_j with the residual vectors r_j complicates the algorithm; it is better to use one or the other. We use the residual vectors, since that lets us apply very similar recurrence techniques as we used in Sections 4.3.3 and 4.3.4 for the Lanczos process.

Suppose we want to perform two steps of CG in a single iteration, using the matrix powers kernel ($s = 2$). At step j, we know α_j, x_{j+1}, r_{j+1}, β_j and p_{j+1}, and we wish to compute α_{j+1}, x_{j+2}, r_{j+2}, β_{j+1} and p_{j+2}. (We increment j by two each iteration, to simplify the notation.) In order to compute α_{j+1}, we first need Ap_{j+1}. Given the equation $p_{j+1} = r_{j+1} + \beta_j p_j$, we pre-multiply both sides by A:

$$Ap_{j+1} = Ar_{j+1} + \beta_j Ap_j,$$

and then use the equation $r_{j+1} = r_j - \alpha_j Ap_j$ to obtain:

$$Ap_{j+1} = A(r_j - \alpha_j Ap_j) + \beta_j Ap_j = Ar_j - \alpha_j A^2 p_j + \beta_j Ap_j.$$

If $j = 0$, then $r_j = p_j = r_0$, and we have from the output of the matrix powers kernel the necessary quantities Ap_0 and $A^2 p_0$. If $j > 0$, then we need to solve for Ar_j:

$$Ar_j = Ap_j - \beta_{j-1} Ap_{j-1},$$

so that

$$\begin{aligned} Ap_{j+1} &= (Ap_j - \beta_{j-1} Ap_{j-1}) - \alpha_j A^2 p_j + \beta_j Ap_j \\ &= (1 + \beta_j) Ap_j - \beta_{j-1} Ap_{j-1} - \alpha_j A^2 p_j. \end{aligned} \tag{5.1}$$

Recall that j increases by increments of two. In the previous iteration $(j-2)$, we needed to compute $Ap_{(j-2)+1} = Ap_{j-1}$, so as long as we save that vector from the previous iteration, we can use it to compute Ar_j in the current iteration. That lets us compute Ap_{j+1}. Now we can compute α_{j+1}:

$$\alpha_{j+1} = \langle r_{j+1}, r_{j+1} \rangle / \langle Ap_{j+1}, p_{j+1} \rangle,$$

update the residual:

$$r_{j+2} = r_{j+1} - \alpha_{j+1} Ap_{j+1},$$

and compute β_{j+1} and p_{j+2}:

$$\begin{aligned} \beta_{j+1} &= \langle r_{j+2}, r_{j+2} \rangle / \langle r_{j+1}, r_{j+1} \rangle, \\ p_{j+2} &= r_{j+2} + \beta_{j+1} p_{j+1}. \end{aligned}$$

One step of the original CG procedure requires one sparse matrix-vector multiply (Ap_j), two inner products ($\langle Ap_j, p_j \rangle$ and $\langle r_{j+1}, r_{j+1} \rangle$), and three AXPYs. Two steps of ordinary CG need two sparse matrix-vector multiplies, four inner products, and six AXPYs. In either case, four vectors (r_j, p_j, Ap_j and x_j) must be saved. The modified CG procedure requires additional space for storing Ap_{j-1} and $A^2 p_j$. It only needs storage for Ap_{j-1} and Ap_j; it can overwrite Ap_{j-1} with Ap_{j+1} and keep Ap_j for the next iteration. The first half-iteration requires two dot products and three AXPYs. The second half-iteration requires five AXPYs and two dot products, so a single "double iteration" calls for eight vector additions and four inner products (which is only two more vector additions than standard CG).

Algorithm 31 shows the resulting "two-step" version of CG. Despite this promising beginning, however, this approach does not seem to generalize to an s-step method of arbitrary length. Note that at every iteration, Algorithm 31 needs the vectors Ap_{j-1}, $A^2 p_{j-1}$, ..., $A^{\rho-1} p_{j-1}$. For $s = 2$, this is not a problem, since we computed these vectors in the previous iteration. However, for $s > 2$, we lack access to these vectors, unless we compute them explicitly at additional expense. The problem comes from the cyclic dependency between p_j and r_j in the standard CG algorithm. A variant of CG (see Saad [208, p. 192]) uses a three-term recurrence to calculate x_{j+1} and r_{j+1}, thus avoiding the p_j vectors altogether, at the cost of keeping one additional vector r_{j-1}. Algorithm 32 shows the variant. We will show that this algorithm admits a simpler approach to generalizing CG for the matrix powers kernel.

5.4 Communication-avoiding CG

In this section, we develop two communication-avoiding versions of CG. Algorithm 34 avoids communication in the sparse matrix operations only, by replacing the sparse matrix-vector multiplies with the matrix powers kernel (Section 2.1). The result is that a single outer iteration of Algorithm 34 does the same work as s iterations of standard CG, but reads the sparse matrix from slow memory only $1 + o(1)$ times, rather than s times. Algorithm 34 is best for problems such that either SpMV or the sequential part of vector operations dominate the runtime of standard CG. The second communication-avoiding version of CG, Algorithm 35, incorporates both the matrix powers kernel (Section 2.1) and a block inner product. As a result, a single outer iteration of Algorithm 35 only requires $\Theta(\log P)$ messages for the

Algorithm 31 2-step conjugate gradient

Input: $Ax = b$: $n \times n$ SPD linear system
Input: x_1: initial guess
Output: Approximate solution x_j of $Ax = b$

1: $r_1 := b - Ax_1$
2: $p_1 := r_1$
3: $\beta_0 := 0$
4: $q := 0$
5: **for** $j = 1, 3, 5, \ldots,$ until convergence **do**
6: Compute $w_1 := Ap_j$ and $w_2 = A^2 p_j$
7: $\alpha_j := \langle r_j, r_j \rangle / \langle w_1, p_j \rangle$
8: $x_{j+1} := x_j + \alpha_j p_j$
9: $r_{j+1} := r_j - \alpha_j w_1$
10: $\beta_j := \langle r_{j+1}, r_{j+1} \rangle / \langle r_j, r_j \rangle$
11: $p_{j+1} := r_{j+1} + \beta_j p_j$
12: $w := (1 + \beta_j) w_1 - \alpha_j w_2 - \beta_{j-1} q$ ▷ Now $w = Ap_{j+1}$.
13: $\alpha_{j+1} := \langle r_{j+1}, r_{j+1} \rangle / w p_{j+1}$
14: $x_{j+2} := x_{j+1} + \alpha_{j+1} p_{j+1}$
15: $r_{j+2} := r_{j+1} - \alpha_{j+1} A p_{j+1}$
16: $\beta_{j+1} := \langle r_{j+2}, r_{j+2} \rangle / \langle r_{j+1}, r_{j+1} \rangle$
17: $p_{j+2} := r_{j+2} + \beta_{j+1} p_{j+1}$
18: $q := w$
Assert: $q = Ap_{j+1}$ from the last iteration
19: **end for**

vector operations, rather than $\Theta(s \log P)$ as in Algorithm 34 and standard CG. Furthermore, Algorithm 35 retains the property of Algorithm 34, that a single outer iteration only reads the sparse matrix from slow memory $1 + o(1)$ times, rather than s times as in standard CG. Algorithm 35 is best for problems such that the cost of global reductions is a significant part of the runtime of standard CG.

5.4.1 Three-term recurrence variant of CG

The discussion in the previous section suggests that the data dependencies between the residual vectors r_j, and the A-conjugate vectors p_j, hinder the development of a communication-avoiding version of CG. Despite the name "CG" (the "Method of Conjugate Gradients"), there are mathematically equivalent formulations of the algorithm which do not include the A-conjugate vectors. Our approach to developing a communication-avoiding version of CG is, therefore, to abandon standard CG and start instead with a different formulation, called CG3 (see e.g., Saad [208, Algorithm 6.19, p. 192]). CG3 is a three-term recurrence variant of CG which only uses the residual vectors r_j and the approximate solution vectors x_j. In exact arithmetic, CG3 computes the same approximate solutions x_j at each iteration as standard CG. CG3 does not communicate asymptotically more or less than standard CG; we present it only as an intermediate step between standard CG and the two communication-avoiding

versions of CG presented in this section.

Algorithm 32 CG3 (3-term recurrence variant of CG)

Input: $Ax = b$: $n \times n$ SPD linear system
Input: x_1: initial guess
Output: Approximate solution x_j of $Ax = b$

1: $x_0 := 0_{n,1}$, $r_0 := 0_{n,1}$, $r_1 := b - Ax_1$
2: **for** $j = 1, 2, \ldots,$ until convergence **do**
3: $\quad w_j := Ar_j$
4: $\quad \mu_j := \langle r_j, r_j \rangle$; test for convergence
5: $\quad \nu_j := \langle w_j, r_j \rangle$
6: $\quad \gamma_j := \mu_j / \nu_j$
7: $\quad$ **if** $j = 1$ **then**
8: $\quad\quad \rho_j := 1$
9: $\quad$ **else**
10: $\quad\quad \rho_j := \left(1 - \frac{\gamma_j}{\gamma_{j-1}} \frac{\mu_j}{\mu_{j-1}} \frac{1}{\rho_{j-1}}\right)^{-1}$
11: $\quad$ **end if**
12: $\quad x_{j+1} := \rho_j (x_j + \gamma_j r_j) + (1 - \rho_j) x_{j-1}$
13: $\quad r_{j+1} := \rho_j (r_j - \gamma_j w_j) + (1 - \rho_j) r_{j-1}$
14: **end for**

We show CG3 here as Algorithm 32. We give a detailed derivation of CG3 in Appendix C.1, where in particular we show that the coefficient ρ_j is always defined as long as CG3 has not broken down (like standard CG, CG3 is subject to "lucky" breakdown). Note that Saad's CG3 has a small error in the recurrence formula for x_{j+1}; we discuss this further in Appendix C.1.

CG3 amounts to a "decoupling" of the "coupled two-term" recurrences for r_j and p_j in the standard version of CG. Many Krylov subspace methods may be reformulated either as coupled two-term recurrences, or uncoupled three-term recurrences. The accuracy properties of these two formulations may differ in finite-precision arithmetic, which is investigated e.g., by Gutknecht and Strakoš [125]. The authors observe that in the three-term recurrence variant of algorithms such as BiCG,[1] local roundoff errors are amplified rapidly. As a result, the residual vector r_j computed by the recurrence (the *updated residual*) may differ significantly from $b - Ax_j$ (the *true residual*). While such behavior is possible for CG, in practice it is not commonly observed. This helps justify our use of CG3, rather than standard CG, as the base of CA-CG. Nevertheless, this observation requires further treatment in Chapter 6, when we discuss the development of communication-avoiding versions of nonsymmetric Lanczos and BiCG.

Our development of CA-CG is based on the orthogonality of the residual vectors r_j in CG3. The residual vectors r_j in CG3 are the same, in exact arithmetic, as the residual vectors in standard CG. That means they are orthogonal in exact arithmetic, and in fact they are just scaled versions of the symmetric Lanczos residual vectors. This means that we

[1] BiCG is the *method of biconjugate gradients*, for solving nonsymmetric linear systems. See Chapter 6 for details.

can construct CA-CG from CG3 using the same techniques in Sections 4.3.3 and 4.3.4 that we used to develop the Gram-Schmidt and Gram matrix versions of communication-avoiding Lanczos iteration.

5.4.2 Summary of CA-CG derivation

The two communication-avoiding CG algorithms are complicated, so we develop them in steps, which we summarize here. The final two steps will avoid communication, the first between levels of the memory hierarchy, and the second both in parallel and between levels of the memory hierarchy. We begin with CG3 (Algorithm 32) and then do the following:

1. Algorithm 33: Add an "outer iteration" loop, so that we can name quantities both with the inner iteration number j and the outer iteration number k. This differs from Algorithm 32 only notationally.

2. Algorithm 34: Replace the sparse matrix-vector products in Algorithm 33 with the matrix powers kernel. The result is that a single outer iteration of Algorithm 34 does the same work as s iterations of standard CG, but reads the sparse matrix from slow memory only $1 + o(1)$ times.

3. Algorithm 35: replace the inner products in Algorithm 34 with a new *inner product coalescing* kernel. The result is that a single outer iteration of Algorithm 35 only requires $\Theta(\log P)$ messages for the vector operations, rather than $\Theta(s \log P)$ as in Algorithm 34 and standard CG. Furthermore, Algorithm 35 retains the property of Algorithm 34, that a single outer iteration only reads the sparse matrix from slow memory $1 + o(1)$ times, rather than s times as in standard CG.

We begin with the first step. This adds an outer iteration loop to CG3 (Algorithm 32), resulting in Algorithm 33. This is only useful for setting up notation for future steps. For example, it lets us name quantities both with the inner iteration number j and the outer iteration number k. Many of these quantities will be the same (in exact arithmetic) in this algorithm as in CA-CG.

5.4.3 Recurrence for residual vectors

We now use Algorithm 33 to derive a matrix-style recurrence for the residual vectors, analogous to the Lanczos recurrence. This will serve as useful notation for later. CG3 computes residual vectors via a three-term recurrence

$$r_{sk+j+1} = \rho_{sk+j} \left(r_{sk+j} - \gamma_{sk+j} A r_{sk+j} \right) + (1 - \rho_{sk+j}) r_{sk+j-1}, \tag{5.2}$$

which we rearrange into

$$A r_{sk+j} = \frac{1 - \rho_{sk+j}}{\rho_{sk+j} \gamma_{sk+j}} r_{sk+j-1} + \frac{1}{\gamma_{sk+j}} r_{sk+j} - \frac{1}{\rho_{sk+j} \gamma_{sk+j}} r_{sk+j+1}. \tag{5.3}$$

This rearrangement is legitimate for the following reasons:

Algorithm 33 CG3 with outer and inner iteration indexing

Input: $Ax = b$: $n \times n$ SPD linear system
Input: x_1: initial guess
Output: Approximate solution x_{sk+j} of $Ax = b$

$x_0 := 0_{n,1}$, $r_0 := 0_{n,1}$, $r_1 := b - Ax_1$
for $k = 0, 1, 2, \ldots,$ until convergence **do**
  for $j = 1, 2, \ldots, s$ **do**
    $w_{sk+j} := Ar_{sk+j}$
    $\mu_{sk+j} := \langle r_{sk+j}, r_{sk+j} \rangle$; test for convergence
    $\nu_{sk+j} := \langle w_{sk+j}, r_{sk+j} \rangle$
    $\gamma_{sk+j} := \mu_{sk+j} / \nu_{sk+j}$ ▷ γ_i is defined for $i \geq 1$
    if $sk + j = 1$ **then** ▷ ρ_i is defined for $i \geq 1$
      $\rho_{sk+j} := 1$
    else
      $\rho_{sk+j} := \left(1 - \frac{\gamma_{sk+j}}{\gamma_{sk+j-1}} \frac{\mu_{sk+j}}{\mu_{sk+j-1}} \frac{1}{\rho_{sk+j-1}}\right)^{-1}$
    end if
    $x_{sk+j+1} := \rho_{sk+j}(x_{sk+j} + \gamma_{sk+j} r_{sk+j}) + (1 - \rho_{sk+j}) x_{sk+j-1}$
    $r_{sk+j+1} := \rho_{sk+j}(r_{sk+j} - \gamma_{sk+j} w_{sk+j}) + (1 - \rho_{sk+j}) r_{sk+j-1}$
  end for
end for

1. $\gamma_{sk+j} = \langle r_{sk+j}, r_{sk+j} \rangle / \langle Ar_{sk+j}, r_{sk+j} \rangle$ is always positive in exact arithmetic, as long as r_{sk+j} is nonzero. (This is because A is SPD.)

2. ρ_{sk+j} is always defined and nonzero in exact arithmetic, as long as CG3 has not broken down. (See Section C.1 for a detailed discussion.)

Thus, the above coefficients always will be defined (in exact arithmetic) as long as the iteration has not converged. From the above vector recurrence (Equation (5.3)), we can construct a matrix-style recurrence for the CG3 residual vectors analogous to the Lanczos recurrence. Define the matrix of residual vectors

$$R_k = [r_{sk+1}, \ldots, r_{sk+s}]$$

for $k = 0, 1, 2, \ldots$, and define

$$\underline{R}_k = [R_k, r_{sk+s+1}].$$

Note that this is not an R factor from a QR factorization. The notation does conflict with that of earlier chapters,[2] but our versions of CG will not involve QR factorizations, so there should be no confusion. We also define $R_{-1} = 0_{n,s}$ (an $n \times s$ matrix of zeros) and $\underline{R}_{-1} = [0_{n,s}, r_0]$.

From Equation (5.3), it follows that

$$AR_k = \frac{1 - \rho_{sk+1}}{\rho_{sk+j}\gamma_{sk+1}} r_{sk} e_1^T + \underline{R}_k \underline{T}_k, \tag{5.4}$$

[2] We didn't wish to use P_k, because the A-conjugate vectors in standard CG are named with lowercase p.

where the $s+1$ by s tridiagonal matrix $\underline{T}_k$ is given by

$$\underline{T}_k = \underline{\tilde{T}}_k \cdot \text{diag}\left(\rho_{sk+1}\gamma_{sk+1}, \ldots, \rho_{sk+s}\gamma_{sk+s}\right)^{-1},$$
$$\underline{\tilde{T}}_k = \begin{pmatrix} \rho_{sk+1} & 1-\rho_{sk+2} & 0 & \ldots & 0 \\ -1 & \rho_{sk+2} & 1-\rho_{sk+3} & & \vdots \\ 0 & -1 & \ddots & \ddots & \\ \vdots & & \ddots & & 1-\rho_{sk+s} \\ & & & -1 & \rho_{sk+s} \\ 0 & & & & -1 \end{pmatrix}. \tag{5.5}$$

This matrix should not be confused with the tridiagonal matrix from symmetric Lanczos. For instance, T_k, the leading $s \times s$ block of $\underline{T}_k$ is not necessarily symmetric.

5.4.4 Including the matrix powers kernel

In this section, we derive "CA-CG with inner products" (Algorithm 34). This involves replacing the sparse matrix-vector products in Algorithm 33 with an invocation of the matrix powers kernel, and adding new recurrences to make the resulting algorithm (Algorithm 34) equivalent in exact arithmetic to to Algorithm 33. One outer iteration of Algorithm 34 does the same work as s iterations of standard CG, but reads the sparse matrix from slow memory only $1+o(1)$ times, rather than s times. Furthermore, in parallel, one outer iteration of Algorithm 34 requires only about as many messages for the sparse matrix operations as s iterations of standard CG. Algorithm 34 is best for problems such that the sparse matrix operations dominate the runtime of standard CG. In the following Section 5.4.5, we will derive Algorithm 35, which also avoids messages in parallel for the vector operations.

CA-CG with inner products has the following rough outline:

1: $r_0 := 0$
2: **for** $k = 0, 1, 2, \ldots$ until convergence **do**
3: $\quad v_{sk+1} := r_{sk+1}$ ▷ Current residual vector
4: $\quad$ Use the matrix powers kernel to compute $\underline{V}_k = [v_{sk+1}, v_{sk+2}, \ldots, v_{sk+s+1}]$ whose columns are a basis for span$\{r_{sk+1}, Ar_{sk+1}, \ldots, A^s r_{sk+1}\}$
5: $\quad$ **for** $j = 1, 2, \ldots, s$ **do**
6: $\quad\quad$ Compute vector of $2s+1$ coefficients d_{sk+j} such that $Ar_{sk+j} = [R_{k-1}, \underline{V}_k]d_{sk+j}$
7: $\quad\quad w_{sk+j} := [R_{k-1}, \underline{V}_k]d_{sk+j}$ ▷ $w_{sk+j} = Ar_{sk+j}$
8: $\quad\quad \mu_{sk+j} := \langle r_{sk+j}, r_{sk+j}\rangle$; test for convergence
9: $\quad\quad \nu_{sk+j} := \langle w_{sk+j}, r_{sk+j}\rangle$
10: $\quad\quad \gamma_{sk+j} := \mu_{sk+j}/\nu_{sk+j}$
11: $\quad\quad$ Compute ρ_{sk+j} using previously computed quantities γ_{sk+j-1}, γ_{sk+j}, $\langle r_{sk+j}, r_{sk+j}\rangle$, $\langle r_{sk+j-1}, r_{sk+j-1}\rangle$, and ρ_{sk+j-1}
12: $\quad\quad x_{sk+j+1} := \rho_{sk+j}(x_{sk+j} + \gamma_{sk+j}r_{sk+j}) + (1-\rho_{sk+j})x_{sk+j-1}$
13: $\quad\quad r_{sk+j+1} := \rho_{sk+j}\left(r_{sk+j} - \gamma_{sk+j}w\right) + \left(1-\rho_{sk+j}\right)r_{sk+j-1}$
14: $\quad$ **end for**
15: **end for**

The key difference here between ordinary three-term CG and the communication-avoiding algorithm is the vector of $2s+1$ coefficients d_{sk+j} such that

$$Ar_{sk+j} = [R_{k-1}, \underline{V}_k] d_{sk+j},$$

where $\underline{V}_k = [v_{sk+1}, v_{sk+2}, \ldots, v_{sk+s+1}]$ is the matrix of basis vectors generated by the matrix powers kernel. We can derive a recursion formula for d_{sk+j} by using the recurrence (5.3) for the residual vectors, which we repeat here:

$$Ar_{sk+j} = \frac{1-\rho_{sk+j}}{\rho_{sk+j}\gamma_{sk+j}} r_{sk+j-1} + \frac{1}{\gamma_{sk+j}} r_{sk+j} - \frac{1}{\rho_{sk+j}\gamma_{sk+j}} r_{sk+j+1}.$$

This holds for $k = 0, 1, \ldots$ and for $j = 1, 2, \ldots, s$. For any $k \geq 0$, we will need to compute d_{sk}, d_{sk+1}, $\ldots$, d_{sk+s}, each using already computed quantities. By inspection, for $k = 0$, $d_{sk} = d_0 = 0_{2s+1}$, and for $k > 0$, $d_{sk} = [0_{1,s-1}, 1, 0_{1,s+1}]^T$. This formula only makes sense for $s \geq 2$; of course $s < 2$ means the matrix powers kernel isn't used and the method is no different than CG3, so we can dismiss this case.

The next step in developing a recurrence for the d_{sk+j} coefficients is to use the coefficients in the s-step basis $\underline{V}_k$. Recall that when we began to explain the Arnoldi(s) algorithm in Section 3.2.1, we introduced the s-step basis equation (Equation (3.3)). We repeat it here:

$$AV_k = \underline{V}_k \underline{B}_k, \tag{5.6}$$

where

$$\underline{V}_k = [V_k, v_{sk+s+1}] = [v_{sk+1}, v_{sk+2}, \ldots, v_{sk+s}, v_{sk+s+1}],$$

and $\underline{B}_k$ is the $s+1$ by s "change-of-basis" matrix, whose entries depend only on the choice of basis in outer iteration k. See Chapter 7 for formulae for $\underline{B}_k$ for various s-step bases. For the bases we use in this thesis, the matrix $\underline{B}_k$ is always tridiagonal, and in some cases it is bidiagonal. However, in the derivation below, the strongest assumption we make is that it is upper Hessenberg. In the case of CA-Arnoldi (Section 3.3, CA-Lanczos (Section 3.3), and CA-GMRES (Section 3.4), the starting vector v_{sk+1} of the basis is unit-length. However, this assumption is not necessary. In a communication-avoiding version of CG, we can choose $v_{sk+1} = r_{sk+1}$ and obtain a perfectly legitimate Krylov subspace basis.

Given Equation (5.6) and $v_{sk+1} = r_{sk+1}$, we can immediately deduce a formula for d_{sk+1}:

$$d_{sk+1} = \left[0_{1,s}, \underline{B}_k(1,1), \underline{B}_k(2,1), 0_{1,s-1}\right]^T.$$

For $j > 1$, we derive a recurrence for the d_{sk+j} coefficients. Begin with the recurrence for the residual vectors:

$$r_{sk+j} = -\rho_{sk+j-1}\gamma_{sk+j-1} Ar_{sk+j-1} + \rho_{sk+j-1} r_{sk+j-1} + (1-\rho_{sk+j-1}) r_{sk+j-2},$$

and multiply both sides by A:

$$Ar_{sk+j} = -\rho_{sk+j-1}\gamma_{sk+j-1} A^2 r_{sk+j-1} + \rho_{sk+j-1} Ar_{sk+j-1} + (1-\rho_{sk+j-1}) Ar_{sk+j-2}.$$

Now use the definition of d_{sk+j-1} and d_{sk+j-2}. Note that in this case, $2 \leq j \leq s$.

$$\begin{aligned} Ar_{sk+j} = &-\rho_{sk+j-1}\gamma_{sk+j-1} A[R_{k-1}, \underline{V}_k] d_{sk+j-1} + \\ &\rho_{sk+j-1}[R_{k-1}, \underline{V}_k] d_{sk+j-1} + (1-\rho_{sk+j-1})[R_{k-1}, \underline{V}_k] d_{sk+j-2}. \end{aligned} \tag{5.7}$$

This almost gives us a recursion formula for d_{sk+j}, except for the term

$$A[R_{k-1}, \underline{V}_k] d_{sk+j-1}.$$

However, we can simplify this term using Equation (5.4), the basis equation $AV_k = \underline{V}_k \underline{B}_k$, and what we can deduce about the structure of d_{sk+j-1}. First, recall from Equation (5.4) that

$$AR_{k-1} = \frac{1 - \rho_{s(k-1)+1}}{\rho_{s(k-1)+1}\gamma_{s(k-1)+1}} r_{s(k-1)} e_1^T + \underline{R}_{k-1}\underline{T}_{k-1},$$

where the $s+1$ by s tridiagonal matrix $\underline{T}_{k-1}$ is zero for $k = 0$, and for $k > 0$ is given by Equation (5.5). Also, it's easy to prove by induction that the first entry of d_{sk+j-1} must be zero for $2 \leq j \leq s$. Thus,

$$\begin{aligned}
AR_{k-1} d_{sk+j-1}(1:s) &= \left(\frac{1 - \rho_{s(k-1)+1}}{\rho_{s(k-1)+1}\gamma_{s(k-1)+1}} r_{s(k-1)} e_1^T + \underline{R}_{k-1}\underline{T}_{k-1} \right) d_{sk+j-1}(1:s) \\
&= \left(\underline{R}_{k-1}\underline{T}_{k-1} \right) d_{sk+j-1}(1:s) \\
&= \left(R_{k-1}T_{k-1} - \left(\rho_{s(k-1)+s}\gamma_{s(k-1)+s}\right)^{-1} r_{sk+1} e_s^T \right) d_{sk+j-1}(1:s) \\
&= \left(R_{k-1}T_{k-1} \right) d_{sk+j-1}(1:s) - \left(\rho_{sk}\gamma_{sk}\right)^{-1} r_{sk+1} d_{sk+j-1}(s) \\
&= [R_{k-1}, \underline{V}_k] \begin{bmatrix} T_{k-1} d_{sk+j-1}(1:s) \\ -\left(\rho_{sk}\gamma_{sk}\right)^{-1} d_{sk+j-1}(s) \\ 0_s \end{bmatrix}.
\end{aligned}$$

From a similar easy induction, entry $2s+1$ (the last entry) of d_{sk+j-1} must also be zero for $2 \leq j \leq s$. Thus,

$$\begin{aligned}
A\underline{V}_k d_{sk+j-1}(s+1:2s+1) &= A[V_k, 0] d_{sk+j-1}(s+1:2s+1) \\
&= [\underline{V}_k \underline{B}_k, 0] d_{sk+j-1}(s+1:2s+1) \\
&= \underline{V}_k \underline{B}_k d_{sk+j-1}(s+1:2s).
\end{aligned}$$

It follows that

$$\begin{aligned}
&A[R_{k-1}, \underline{V}_k] d_{sk+j-1} = \\
&\qquad [R_{k-1}, \underline{V}_k] \begin{bmatrix} T_{k-1} d_{sk+j-1}(1:s) \\ -\left(\rho_{sk}\gamma_{sk}\right)^{-1} d_{sk+j-1}(s) e_1 + \underline{B}_k d_{sk+j-1}(s+1:2s) \end{bmatrix}. \quad (5.8)
\end{aligned}$$

We can now plug Equation (5.8) into Equation (5.7) to get the recurrence for d_{sk+j}, when $2 \leq j \leq s$:

$$\begin{aligned}
&d_{sk+j} = \rho_{sk+j-1} d_{sk+j-1} + (1 - \rho_{sk+j-1}) d_{sk+j-2} - \\
&\qquad \rho_{sk+j-1}\gamma_{sk+j-1} \begin{bmatrix} T_{k-1} d_{sk+j-1}(1:s) \\ -\left(\rho_{sk}\gamma_{sk}\right)^{-1} d_{sk+j-1}(s) e_1 + \underline{B}_k d_{sk+j-1}(s+1:2s) \end{bmatrix}. \quad (5.9)
\end{aligned}$$

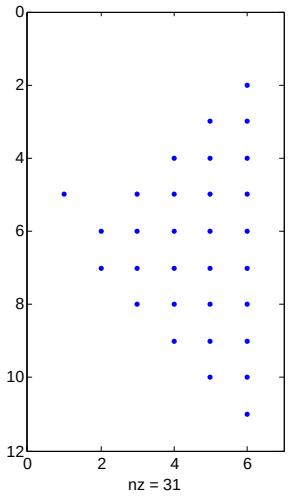


Figure 5.1: Spy plot (visualization of the sparsity pattern) of the $2s+1$ by $s+1$ matrix $[d_{sk}, d_{sk+1}, \ldots, d_{sk+s}]$, for the case $s = 5$. Each (potential) nonzero in the matrix is shown as a blue dot.

Equation (5.9) holds for all $k \geq 0$. Here is the complete recurrence for d_{sk+j}, with all the cases:

$$d_{sk+j} = \begin{cases} \begin{cases} 0_{2s+1,1} & \text{for } k = 0,\ j = 0, \\ [0_{1,s-1}, 1, 0_{1,s+1}]^T & \text{for } k > 0,\ j = 0, \end{cases} \\ [0_{1,s}, \underline{B}_k(1,1), \underline{B}_k(2,1), 0_{1,s-1}]^T & \text{for } j = 1, \\ \rho_{sk+j-1} d_{sk+j-1} + (1 - \rho_{sk+j-1})\, d_{sk+j-2} - \\ \quad \rho_{sk+j-1}\gamma_{sk+j-1} \begin{bmatrix} T_{k-1} d_{sk+j-1}(1:s) \\ -\frac{d_{sk+j-1}(s)}{\rho_{sk}\gamma_{sk}} e_1 \end{bmatrix} + & \text{for } j = 2, 3, \ldots, s. \\ \quad \rho_{sk+j-1}\gamma_{sk+j-1} \begin{bmatrix} 0_{s,1} \\ \underline{B}_k d_{sk+j-1}(s+1:2s) \end{bmatrix} \end{cases} \tag{5.10}$$

There is plenty of structure to exploit in these recurrences. In particular, the nonzeros of d_{sk+j} grow outward by one from the center, for each increase in j. Figure 5.1 shows a visualization of the nonzeros of the matrix $[d_{sk}, d_{sk+1}, \ldots, d_{sk+s+1}]$, for the illustrative case $s = 5$. We show the recurrences in the above matrix form for brevity and clarity.

The coefficients d_{sk+j} satisfying $Ar_{sk+j} = [R_{k-1}, \underline{V}_k] d_{sk+j}$ let us recover the CG residual vectors from the output $\underline{V}_k$ of the matrix powers kernel, without needing to compute the sparse matrix-vector product $w := Ar_{sk+j}$. From this comes Algorithm 34, a version of CA-CG that uses the matrix powers kernel for the sparse matrix-vector products, but retains the inner products of the original three-term recurrence version of CG. A single outer iteration of Algorithm 34 computes the same approximate solution (in exact arithmetic) as s iterations of standard CG, but reads the sparse matrix A from slow memory only $1 + o(1)$ times, rather than s times. Algorithm 34 is best for problems such that either SpMV or the sequential part of vector operations dominate the runtime of standard CG. If the parallel part of vector operations – that is, the global reductions and / or the synchronizations – takes a significant

part of the runtime of standard CG, then Algorithm 35 in the following Section 5.4.5 should be considered.

Algorithm 34 CA-CG with inner products

Input: $Ax = b$: $n \times n$ SPD linear system
Input: x_1: initial guess
Output: Approximate solution x_{sk+j} of $Ax = b$

1: $x_0 := 0_{n,1}$, $r_0 := 0_{n,1}$, $r_1 := b - Ax_1$
2: **for** $k = 0, 1, \ldots,$ until convergence **do**
3: $v_{sk+1} := r_{sk+1}$ ▷ Residual vector from previous outer iteration
4: Compute matrix powers kernel $\underline{V}_k = [v_{sk+1}, \ldots, v_{sk+s+1}]$
5: **for** $j = 1, 2, \ldots, s$ **do**
6: Compute coefficients d_{sk+j} such that $Ar_{sk+j} = [R_{k-1}, \underline{V}_k]d_{sk+j}$ (Eq. (5.10))
7: $w_{sk+j} := [R_{k-1}, \underline{V}_k]d_{sk+j}$
8: $\mu_{sk+j} := \langle r_{sk+j}, r_{sk+j} \rangle$; test for convergence
9: $\nu_{sk+j} := \langle w, r_{sk+j} \rangle$
10: $\gamma_{sk+j} := \mu_{sk+j}/\nu_{sk+j}$
11: **if** $sk + j = 1$ **then**
12: $\rho_{sk+j} := 1$
13: **else**
14: $\rho_{sk+j} := \left(1 - \frac{\gamma_{sk+j}}{\gamma_{sk+j-1}} \frac{\mu_{sk+j}}{\mu_{sk+j-1}} \frac{1}{\rho_{sk+j-1}}\right)^{-1}$
15: **end if**
16: $x_{sk+j+1} := \rho_{sk+j}(x_{sk+j} + \gamma_{sk+j} r_{sk+j}) + (1 - \rho_{sk+j}) x_{sk+j-1}$
17: $r_{sk+j+1} := \rho_{sk+j}(r_{sk+j} - \gamma_{sk+j} w_{sk+j}) + (1 - \rho_{sk+j}) r_{sk+j-1}$ ▷ Computing x_{sk+j+1} and r_{sk+j+1} requires no parallel communication, and no synchronization until we compute μ_{sk+j+1}
18: **end for**
19: **end for**

5.4.5 Removing the inner products

In this section, we derive CA-CG (Algorithm 35). This completes the derivation process of the previous two sections. Algorithm 35 is equivalent in exact arithmetic to standard CG, in that one outer iteration of Algorithm 35 does the same work as s iterations of standard CG. Like Algorithm 34, Algorithm 35 reads the sparse matrix from slow memory only $1 + o(1)$ times per outer iteration, rather than s times. In parallel, one outer iteration of Algorithm 35 requires only about as many messages for the sparse matrix operations as s iterations of standard CG. Furthermore, unlike CA-CG with inner products, one outer iteration of Algorithm 35 requires only $\Theta(\log P)$ messages in parallel for the vector operations. One outer iteration of Algorithm 34, or s iterations of standard CG, require $\Theta(s \log P)$ messages in parallel for the vector operations. Algorithm 35 is best for problems such that the parallel cost of global reductions is significant in standard CG.

The Gram matrix

We begin deriving CA-CG (Algorithm 35) by replacing the inner products in Algorithm 34 of the previous section with much shorter inner products involving the $2s+1$ by $2s+1$ Gram matrix

$$G_k := \langle [R_{k-1}, \underline{V}_k], [R_{k-1}, \underline{V}_k] \rangle = \begin{pmatrix} R_{k-1}^* R_{k-1} & R_{k-1}^* \underline{V}_k \\ \underline{V}_k^* R_{k-1} & \underline{V}_k^* \underline{V}_k \end{pmatrix}. \tag{5.11}$$

We defined the Gram matrix in Section 2.5. As we observe there, computing it in parallel requires only one reduction (over blocks of the R_{k-1} and $\underline{V}_k$ matrices. Sequentially, it only requires reading the input data once, which is no worse asymptotically than standard CG. The shorter inner products involving G_k require no communication. In the context of CG, we call the computation of the Gram matrix *inner product coalescing*, since it computes the same essential information (in exact arithmetic) as the inner products in CG3, but in only one communication step.

Note that Equation (5.11) can be simplified, since the columns of $[R_{k-1}, v_{sk+1}]$ are orthogonal in exact arithmetic. For example, $R_{k-1}^* R_{k-1} = D_{k-1}$, an $s \times s$ diagonal matrix with

$$D_{k-1}(j,j) = \langle r_{s(k-1)+j}, r_{s(k-1)+j} \rangle = \mu_{s(k-1)+j}. \tag{5.12}$$

At outer iteration k, we have already computed the $\mu_{s(k-1)+j}$ coefficients (in the previous outer iteration $k-1$). Thus, we do not need to recompute $R_{k-1}^* R_{k-1}$. Furthermore,

$$R_{k-1}^* \underline{V}_k = (\underline{V}_k^* R_{k-1})^*. \tag{5.13}$$

, i.e., the Gram matrix G_k is symmetric. This symmetry suggests that we can compute it in two block reductions, rather than one, in order to reduce the bandwidth cost (both in parallel and sequentially):

1. Compute $G_{k,k-1} := R_{k-1}^* \underline{V}_k$
2. Compute $G_{kk} := \underline{V}_k^* \underline{V}_k$

Then, we have

$$G_k = \begin{pmatrix} D_{k-1} & G_{k,k-1} \\ G_{k,k-1}^* & G_{kk} \end{pmatrix}. \tag{5.14}$$

Replacing the inner products

In this section, our goal is to replace the two inner products

$$\begin{aligned} \mu_{sk+j} &:= \langle r_{sk+j}, r_{sk+j} \rangle \qquad \text{and} \\ \nu_{sk+j} &:= \langle A r_{sk+j}, r_{sk+j} \rangle \end{aligned}$$

with much shorter inner products, involving the Gram matrix, that require no communication (other than to compute the Gram matrix). We do so much as we did in Section 4.3.4 when deriving left-preconditioned CA-Lanczos: we derive vectors of $2s+1$ coefficients d_{sk+j} and g_{sk+j}, for $j = 0, 1, \ldots, s$, where

$$[R_{k-1}, \underline{V}_k] d_{sk+j} = A r_{sk+j} \qquad \text{and} \qquad [R_{k-1}, \underline{V}_k] g_{sk+j} = r_{sk+j},$$

and combine them with the Gram matrix to recover the CG coefficients. We already derived a recurrence (Equation (5.10)) for d_{sk+j} in Section 5.4.4. Here, we derive a recurrence for the g_{sk+j} coefficients, for $j = 0, 1, \ldots, s$. This will let us compute μ_{sk+j} and μ_{sk+j} via

$$\mu_{sk+j} = g_{sk+j}^* G_k g_{sk+j} \qquad \text{and} \qquad \nu_{sk+j} = g_{sk+j}^* G_k d_{sk+j}.$$

Consider the vector of $2s + 1$ coefficients g_{sk+j} such that

$$r_{sk+j} = [R_{k-1}, \underline{V}_k] g_{sk+j}.$$

We can derive a recursion formula for g_{sk+j} (where $j = 0, 1, \ldots, s$) by using the CG3 recurrence (Equation (5.2)) for the residual vectors:

$$r_{sk+j} = \rho_{sk+j-1} \left(r_{sk+j-1} - \gamma_{sk+j-1} A r_{sk+j-1} \right) + (1 - \rho_{sk+j-1}) r_{sk+j-2}.$$

For $k = 0$ and $j = 0$, we have $g_{sk+j} = g_0 = 0_{2s+1,1}$ trivially. For $k > 0$ and $j = 0$, we have

$$g_{sk} = [0_{1,s-1}^T, 1, 0_{1,s+1}^T]^T$$

by inspection, since r_{sk} is column s of R_{k-1}. Also by inspection,

$$g_{sk+1} = [0_{1,s}^T, 1, 0_{1,s}^T]^T.$$

For $j = 2, 3, \ldots, s$, we have

$$\begin{aligned} [R_{k-1}, \underline{V}_k] g_{sk+j} = \rho_{sk+j-1} [R_{k-1}, \underline{V}_k] g_{sk+j-1} \\ - \rho_{sk+j-1} \gamma_{sk+j-1} [R_{k-1}, \underline{V}_k] d_{sk+j-1} + (1 - \rho_{sk+j-1}) [R_{k-1}, \underline{V}_k] g_{sk+j-2} \end{aligned}$$

and therefore

$$g_{sk+j} = \rho_{sk+j-1} g_{sk+j-1} - \rho_{sk+j-1} \gamma_{sk+j-1} d_{sk+j-1} + (1 - \rho_{sk+j-1}) g_{sk+j-2}.$$

This gives us a recurrence for g_{sk+j}:

$$g_{sk+j} = \begin{cases} \begin{cases} 0_{2s+1,1} & k = 0 \\ [0_{1,s-1}, 1, 0_{1,s+1}]^T & k > 0 \end{cases} & j = 0 \\ [0_{1,s}, 1, 0_{1,s}]^T & j = 1 \\ \rho_{sk+j-1} g_{sk+j-1} - \rho_{sk+j-1} \gamma_{sk+j-1} d_{sk+j-1} & \\ \qquad + (1 - \rho_{sk+j-1}) g_{sk+j-2}, & 2 \le j \le s. \end{cases} \tag{5.15}$$

Now we can compute the inner products in CG via

$$\begin{aligned} \langle r_{sk+j}, r_{sk+j} \rangle &:= g_{sk+j}^* G_k g_{sk+j} \qquad \text{and} \\ \langle A r_{sk+j}, r_{sk+j} \rangle &:= g_{sk+j}^* G_k d_{sk+j}. \end{aligned}$$

The resulting CG method, CA-CG, is shown as Algorithm 35.

Algorithm 35 CA-CG

Input: $Ax = b$: $n \times n$ SPD linear system
Input: x_1: initial guess
Output: Approximate solution x_{sk+j} of $Ax = b$

1: $x_0 := 0_{n,1}$, $r_0 := 0_{n,1}$, $R_{-1} = 0_{n,s}$, $r_1 := b - Ax_1$
2: **for** $k = 0, 1, \ldots,$ until convergence **do**
3: $v_{sk+1} := r_{sk+1}$ ▷ Residual vector from previous outer iteration
4: Compute matrix powers kernel $\underline{V}_k = [v_{sk+1}, \ldots, v_{sk+s+1}]$
5: $G_{k,k-1} := R_{k-1}^* \underline{V}_k$, $G_{kk} := \underline{V}_k^* \underline{V}_k$ ▷ 2 block reductions
6: $G_k := \begin{pmatrix} D_{k-1} & G_{k,k-1} \\ (G_{k,k-1})^* & G_{kk} \end{pmatrix}$ (Gram matrix, via Eq. (5.14))
7: **for** $j = 1, 2, \ldots, s$ **do**
8: Compute d_{sk+j} such that $Ar_{sk+j} = [R_{k-1}, \underline{V}_k] d_{sk+j}$ (Eq. (5.10))
9: Compute g_{sk+j} such that $r_{sk+j} = [R_{k-1}, \underline{V}_k] g_{sk+j}$ (Eq. (5.15)) ▷ d_{sk+j} and g_{sk+j} computed redundantly on each parallel processor
10: $w_{sk+j} := [R_{k-1}, \underline{V}_k] d_{sk+j}$ ▷ No parallel communication or synchronization
11: $\mu_{sk+j} := g_{sk+j}^* G_k g_{sk+j}$; test convergence ▷ $\mu_{sk+j} = \langle r_{sk+j}, r_{sk+j} \rangle$ in exact arithmetic
12: $\nu_{sk+j} := g_{sk+j}^* G_k d_{sk+j}$ ▷ $\nu_{sk+j} = \langle Ar_{sk+j}, r_{sk+j} \rangle$ in exact arithmetic
13: $\gamma_{sk+j} := \mu_{sk+j} / \nu_{sk+j}$ ▷ Computing μ_{sk+j} and ν_{sk+j} requires no communication
14: **if** $sk + j = 1$ **then**
15: $\rho_{sk+j} := 1$
16: **else**
17: $\rho_{sk+j} := \left(1 - \frac{\gamma_{sk+j}}{\gamma_{sk+j-1}} \frac{\mu_{sk+j}}{\mu_{sk+j-1}} \frac{1}{\rho_{sk+j-1}}\right)^{-1}$
18: **end if**
19: $x_{sk+j+1} := \rho_{sk+j}(x_{sk+j} + \gamma_{sk+j} r_{sk+j}) + (1 - \rho_{sk+j}) x_{sk+j-1}$
20: $r_{sk+j+1} := \rho_{sk+j}(r_{sk+j} - \gamma_{sk+j} w_{sk+j}) + (1 - \rho_{sk+j}) r_{sk+j-1}$ ▷ Computing x_{sk+j+1} and r_{sk+j+1} requires no parallel communication, and no synchronization until the next outer iteration (for the matrix powers kernel)
21: **end for**
22: **end for**

5.4.6 Eigenvalue estimates for basis construction

Some of the s-step bases in the matrix powers kernel, such as the Newton and Chebyshev bases, require estimates of the eigenvalues of the matrix A. See Chapter 7 for details. In CA-Arnoldi (Section 3.3), CA-GMRES (Section 3.4), and CA-Lanczos (Section 4.2), our approach is to execute s (or $2s$, in some cases) iterations of standard Arnoldi resp. standard GMRES resp. standard Lanczos, use the Ritz values collected from these steps to construct a good basis, and then continue the iteration (without restarting) using CA-Arnoldi resp. CA-GMRES resp. CA-Lanczos. CG is a method for solving linear systems, not eigenvalue problems. However, CG is equivalent in exact arithmetic to Lanczos iteration (see e.g., Greenbaum [120] or Saad [208]). Saad [208, pp. 192–3] shows how to compute the Lanczos tridiagonal matrix T using the CG coefficients. We adapt his calculations to show how to

compute T from s steps of CG3 (Algorithm 32). Since CA-CG and CG3 compute the same coefficients ρ_j and γ_j and the same residual and approximate solution vectors r_j resp. x_j, in exact arithmetic, it is easy to continue the CG3 iteration from where we left off using CA-CG. Without this technique, we would have to restart after collecting the Ritz values, and thus we would waste s steps running an unoptimized iterative method.

We begin by assuming that we have run s iterations of CG3. The residual vectors r_1, r_2, ..., r_s produced by CG3 and CA-CG are, in exact arithmetic, just scaled versions of the Lanczos basis vectors q_1, q_2, ... In particular,

$$q_j = \frac{e^{i\phi_j}}{\|r_j\|_2} r_j \text{ for } j = 1, 2, \ldots, s,$$

where $e^{i\phi_j}$ is a unit-length scaling whose value doesn't matter. We can find a formula for $\|r_j\|_2$ in terms of the CG3 coefficients, but this will not be needed.

The $s \times s$ tridiagonal matrix T produced by symmetric Lanczos has the following form:

$$\begin{pmatrix} \alpha_1 & \beta_1 & & \ldots & 0 \\ \beta_1 & \alpha_2 & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & \beta_{s-1} \\ 0 & \ldots & & \beta_{s-1} & \alpha_s \end{pmatrix}.$$

Here, for $j > 1$,

$$\alpha_j = \begin{cases} \langle Av_j, v_j \rangle & \text{for } j = 1, \\ \langle Av_j - \beta_j v_{j-1}, v_j \rangle & \text{for } j > 1; \end{cases}$$

and

$$\beta_j = \begin{cases} \langle Av_j - \alpha_j v_j, v_{j+1} \rangle & \text{for } j = 1, \\ \langle Av_j - \alpha_j v_j - \beta_{j-1} v_{j-1}, v_{j+1} \rangle & \text{for } j > 1. \end{cases}$$

We can simplify these formulae by defining $v_0 = 0$.

We compute α_j from the CG3 coefficients by using orthogonality of the Lanczos basis vectors:

$$\begin{aligned} \alpha_j &= \langle Av_j - \beta_j v_{j-1}, v_j \rangle = \langle Av_j, v_j \rangle \\ &= \frac{\langle Ar_j, r_j \rangle}{\|r_j\|_2 \|r_j\|_2} = \frac{\langle Ar_j, r_j \rangle}{\langle r_j, r_j \rangle} = \frac{1}{\gamma_j}. \end{aligned}$$

Orthogonality of the v_j gets us started with computing β_j:

$$\begin{aligned} \beta_j &= \langle Av_j - \alpha_j v_j - \beta_{j-1} v_{j-1}, v_{j+1} \rangle \\ &= \langle Av_j, v_{j+1} \rangle = \frac{\langle Ar_j, r_{j+1} \rangle}{\|r_j\|_2 \|r_{j+1}\|_2}. \end{aligned}$$

Now, we rearrange the three-term recurrence for the CG3 residual vectors, and take dot

products of both sides with r_{j+1}:

$$\begin{gathered} r_{j+1} = \rho_j (r_j - \gamma_j A r_j) + (1 - \rho_j) r_{j-1} \Longrightarrow \\ \rho_j \gamma_j A r_j = -r_{j+1} + \rho_j r_j + (1 - \rho_j) r_{j-1} \Longrightarrow \\ \rho_j \gamma_j \langle A r_j, r_{j+1} \rangle = -\langle r_{j+1}, r_{j+1} \rangle + \rho_j \langle r_j, r_{j+1} \rangle + (1 - \rho_j) \langle r_{j-1}, r_{j+1} \rangle \Longrightarrow \\ \rho_j \gamma_j \langle A r_j, r_{j+1} \rangle = -\langle r_{j+1}, r_{j+1} \rangle. \end{gathered}$$

This gives us

$$\langle A r_j, r_{j+1} \rangle = -\frac{\langle r_{j+1}, r_{j+1} \rangle}{\rho_j \gamma_j},$$

and therefore

$$\beta_j = \frac{\langle A r_j, r_{j+1} \rangle}{\|r_j\|_2 \|r_{j+1}\|_2} = -\frac{\langle r_{j+1}, r_{j+1} \rangle}{\rho_j \gamma_j \|r_j\|_2 \|r_{j+1}\|_2} = -\frac{\sqrt{\langle r_{j+1}, r_{j+1} \rangle}}{\rho_j \gamma_j \sqrt{\langle r_j, r_j \rangle}}.$$

The final formulae are

$$\alpha_j = \frac{1}{\gamma_j} \tag{5.16}$$

and

$$\beta_j = -\frac{\sqrt{\langle r_{j+1}, r_{j+1} \rangle}}{\rho_j \gamma_j \sqrt{\langle r_j, r_j \rangle}}. \tag{5.17}$$

We use these to recover the $s \times s$ tridiagonal matrix T and compute from it the Ritz values.

5.5 Left-preconditioned CA-CG

In this section, we derive the left-preconditioned communication-avoiding CG (CA-CG) algorithm, which we call LP-CA-CG (Algorithm 36). Analogous to the previous section's CA-CG (Algorithm 35), one outer iteration of LP-CA-CG requires reading the matrix A and preconditioner M^{-1} only $1 + o(1)$ times each from slow memory, rather than s times each as in standard preconditioned CG. Furthermore, one outer iteration of LP-CA-CG requires only $\Theta(\log P)$ messages on P parallel processors for the vector operations, rather than $\Theta(s \log P)$ messages, as standard preconditioned CG does. It is possible to derive a preconditioned version of "CG with inner products" (Algorithm 34), which does not save parallel messages for the vector operations. We omit the derivation for brevity's sake. As far as we know, we are the first to give a fully communication-avoiding version of left-preconditioned CG that works for arbitrary s-step bases. See Section 5.2 for a full discussion of prior work.

We only consider left preconditioning in this section without loss of generality, because the right-preconditioned version of CA-CG requires only a minor modification of left-preconditioned CA-CG. For reasons discussed in Section 4.3.1, the split-preconditioned algorithm differs from unpreconditioned CA-CG (as shown in Section 5.4) only in the matrix powers kernel. Thus, we do not show split-preconditioned CA-CG in this section either.

The reason we developed left-preconditioned CA-Lanczos in such detail in Sections 4.3.3 and 4.3.4 was to make it easier to develop other algorithms, such as left-preconditioned CA-CG in this section. As we discussed in Section 5.4, the residual vectors r_0, r_1, ... in standard unpreconditioned CG are just scalar multiples of the basis vectors q_1, q_2, ... in standard unpreconditioned Lanczos. In that section, we used that correspondence to derive CA-CG, and in this section, we use an analogous correspondence to derive LP-CA-CG.

We begin our development of LP-CA-CG in Section 5.5.1 by presenting the three-term recurrence variant of left-preconditioned CG, and using that to define some notation for LP-CA-CG. Section 5.5.2 adds another bit of notation, the tridiagonal (but nonsymmetric) matrix $\underline{T}_k$. Section 5.5.3 introduces the Gram matrix, which LP-CA-CG uses instead of computing inner products. In Section 5.5.4, we derive the recurrence relations that allow us to avoid communication when computing the CG coefficients. The following Section 5.5.6 shows how to compute those CG coefficients, once the recurrence relations are known. Finally, Section 5.5.7 shows the entire LP-CA-CG algorithm.

5.5.1 Left-preconditioned CG3

Algorithm 36 Left-preconditioned CG3

Input: $Ax = b$: $n \times n$ SPD linear system
Input: $n \times n$ SPD (left) preconditioner M^{-1}
Input: Initial guess x_1 to $Ax = b$
Output: Approximate solution x_j to $Ax = b$, with $j \in \{1, 2, \dots\}$

1: $z_0 := 0_{n,1}$, and $q_0 := 0_{n,1}$
2: $z_1 := b - Ax_0$, and $q_1 := M^{-1}z_1$ $\triangleright$ z_1 is the unpreconditioned initial residual, and q_1 the preconditioned initial residual.
3: **for** $j = 1, 2, \dots$ until convergence **do**
4: $\quad \mu_j := \langle z_j, q_j \rangle$; test for convergence
5: $\quad w_j := Aq_j$ and $v_j := M^{-1}w_j$
6: $\quad \nu_j := \langle w_j, q_j \rangle$
7: $\quad \gamma_j := \mu_j / \nu_j$
8: $\quad$ **if** $j = 1$ **then**
9: $\quad\quad \rho_j := 1$
10: $\quad$ **else**
11: $\quad\quad \rho_j := \left(1 - \frac{\gamma_j}{\gamma_{j-1}} \cdot \frac{\mu_j}{\mu_{j-1}} \cdot \frac{1}{\rho_{j-1}}\right)^{-1}$
12: $\quad$ **end if**
13: $\quad x_{j+1} := \rho_j (x_j + \gamma_j q_j) + (1 - \rho_j) x_{j-1}$
14: $\quad z_{j+1} := \rho_j (z_j - \gamma_j w_j) + (1 - \rho_j) z_{j-1}$
15: $\quad q_{j+1} := \rho_j (q_j - \gamma_j v_j) + (1 - \rho_j) q_{j-1}$
16: **end for**

In this section, we show as Algorithm 36 the left-preconditioned three-term recurrence variant of CG, which we call "LP-CG3." That algorithm is a straightforward application of preconditioning to CG3 (Algorithm 32), the three-term recurrence variant of CG, which we discussed in Section 5.4.1. LP-CA-CG is mathematically equivalent to LP-CG3, in that it

computes the same CG coefficients ρ_j and γ_j, the same unpreconditioned and preconditioned residual vectors z_j resp. q_j, and the same approximate solutions x_j as LP-CG3 does, in exact arithmetic. We therefore use LP-CG3 as a model for the notation of LP-CA-CG.

In LP-CG3, the SPD $n \times n$ linear system to solve is $Ax = b$, M^{-1} is an SPD preconditioner for that linear system, and the initial guess is x_1. (This differs from the conventional x_0 only so that we can avoid writing x_{-1}.) Note that the notation for the residual vectors in LP-CG3 differs from that in CG3. In CG3, there is only one set of residual vectors $r_j = b - Ax_j$ for $j = 1, 2, \ldots$. In LP-CG3, there are two sets of residual vectors: the unpreconditioned residual vectors $z_j = b - Ax_j$, and the preconditioned residual vectors $q_j = M^{-1}z_j$, for $j = 1, 2, \ldots$. We change the notation to emphasize the similarities between LP-CG3 (and the subsequent LP-CA-CG algorithm), and the left-preconditioned CA-Lanczos algorithms we derive in Sections 4.3.3 and 4.3.4. The main difference between the z_j and q_j vectors in LP-CG3 (and LP-CA-CG), and the corresponding vectors in left-preconditioned CA-Lanczos, is that in CA-Lanczos, the z_j and q_j vectors are scaled so that $\|z_j\|_M = 1$ and $\|q_j\|_M = 1$ (in exact arithmetic). In LP-CG3 and LP-CA-CG, the same z_j and q_j symbols are used to indicate the (unscaled) residual vectors. Since we want to use this notation to make it easy to compare LP-CG3 and LP-CA-CG with left-preconditioned CA-Lanczos, we do not use r_j to indicate the residual vectors.

The indexing of vectors helps illustrate the differences between LP-CA-CG and LP-CG3. LP-CA-CG has two levels of indexing, so that we write (for instance) x_{sk+j} instead of x_j for the current approximate solution. There, the fixed integer $s \in \{2, 3, \ldots\}$ indicates the matrix powers kernel basis length. The index $k = 0, 1, 2, \ldots$ represents the outer iteration number. In each outer iteration, the matrix powers kernel is invoked twice:

- once for the *right-side basis* $\underline{W}_k = [w_{sk+1}, \ldots, w_{sk+s+1}]$ such that $\mathcal{R}(\underline{W}_k) = \text{span}\{w_{sk+1}, AM^{-1}w_{sk+1}, (AM^{-1})^2 w_{sk+1}, \ldots, (AM^{-1})^s w_{sk+1}\} = \mathcal{K}_{s+1}(AM^{-1}, w_{sk+1})$, and
- once for the *left-side basis* $\underline{V}_k = [v_{sk+1}, \ldots, v_{sk+s+1}]$ such that $\mathcal{R}(\underline{V}_k) = \text{span}\{v_{sk+1}, M^{-1}Av_{sk+1}, (M^{-1}A)^2 v_{sk+1}, \ldots, (M^{-1}A)^s v_{sk+1}\} = \mathcal{K}_{s+1}(M^{-1}A, v_{sk+1})$.

The initial vectors for these invocations of the matrix powers kernel are $w_{sk+1} := z_{sk+1}$ resp. $v_{sk+1} := q_{sk+1}$, and by construction, $v_{sk+1} = M^{-1}w_{sk+1}$ for all k. (Note that in LP-CA-CG, these initial vectors for the matrix powers kernel are not necessarily unit length, as they are in CA-Lanczos.) Within an outer iteration, LP-CA-CG executes s inner iterations; $j = 1, 2, \ldots, s$ is used to index those inner iterations. Each inner iteration produces an unpreconditioned residual vector z_{sk+j+1}, a preconditioned residual vector q_{sk+j+1}, and a current approximate solution x_{sk+j+1}, as well as approximations to the current residual M-norm $\langle z_{sk+j}, q_{sk+j} \rangle$.

In LP-CA-CG, computation of x_{sk+j+1}, q_{sk+j+1}, and z_{sk+j+1} may be deferred until the end of outer iteration k. In that case, these vectors with indices $sk+2$, $sk+3$, $\ldots$, $sk+s+1 = s(k+1)+1$ may be computed all at the same time. This may have performance advantages.

Also in LP-CA-CG, we let

$$Z_k = [z_{sk+1}, \ldots, z_{sk+s}], \text{ and}$$
$$Q_k = [q_{sk+1}, \ldots, q_{sk+s}]$$

denote the matrices whose columns are the unpreconditioned resp. preconditioned residual vectors produced by outer iteration k. Occasionally we will need to extend those basis matrices by one additional vector, in which case we will write

$$\underline{Z}_k = [Z_k, z_{sk+s+1}], \text{ and}$$
$$\underline{Q}_k = [Q_k, q_{sk+s+1}].$$

We denote by G_k the $2s+1$ by $2s+1$ Gram matrix (which we will describe further in Section 5.5.3) at outer iteration k. Even at the first outer iteration $k = 0$, G_k is still $2s+1$ by $2s+1$, though all entries of it but the lower right $s+1$ by $s+1$ block are zero when $k = 0$. (This is purely a notational convenience.)

5.5.2 Tridiagonal matrix T_k

The first new notation we introduce to explain LP-CA-CG is the (nonsymmetric) $s \times s$ tridiagonal matrix T_k, and its $s+1$ by s extension $\underline{T}_k$. In LP-CG3 (Algorithm 36), the preconditioned residual vectors q_i satisfy the recurrence

$$\rho_i \gamma_i M^{-1} A q_i = (1 - \rho_i) q_{i-1} + \rho_i q_i - q_{i+1}.$$

In LP-CA-CG, the recurrence is the same, but the single index i is replaced with a double index $sk + j$, as mentioned above:

$$\rho_{sk+j} \gamma_{sk+j} M^{-1} A q_{sk+j} = (1 - \rho_{sk+j}) q_{sk+j-1} + \rho_{sk+j} q_{sk+j} - q_{sk+j+1}. \tag{5.18}$$

Recall that k is the outer iteration index, and $j = 1, 2, \ldots, s$ is the inner iteration index. This forms a three-term recurrence, much like the Lanczos three-term recurrence. In a similar way, we may write this recurrence in matrix form with a tridiagonal matrix. The case of LP-CA-CG differs in two ways:

1. The resulting tridiagonal matrix is not symmetric, and
2. We only want to write this recurrence in matrix form for the vectors of outer iteration k, not for all the vectors.

For each outer iteration, let Δ_k be the $s \times s$ diagonal matrix with $\Delta_k(j, j) = \rho_{sk+j} \gamma_{sk+j}$. Let $\tilde{T}_k$ be the following $s \times s$ tridiagonal matrix:

$$\tilde{T}_k = \begin{pmatrix} \rho_{sk+1} & 1 - \rho_{sk+2} & \cdots & 0 \\ -1 & \rho_{sk+2} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 - \rho_{sk+s} \\ 0 & \cdots & -1 & \rho_{sk+s} \end{pmatrix} \tag{5.19}$$

and let $\underline{\tilde{T}}_k$ be the $s+1$ by s tridiagonal matrix given by

$$\underline{\tilde{T}}_k = \begin{pmatrix} \tilde{T}_k \\ 0, \ldots, 0, -1 \end{pmatrix}. \tag{5.20}$$

Then we may write the three-term recurrence (5.18) in matrix form as follows:

$$M^{-1}AQ_k\Delta_k = (1-\rho_{sk+1})q_{sk}e_1^T + \underline{Q}_k\underline{\tilde{T}}_k. \tag{5.21}$$

where $\underline{Q}_k = [Q_k, q_{s(k+1)+1}]$. Note that the matrix formulation of the usual Lanczos recurrence would start with the first vector q_1, and not with the vector q_{sk+1}. This is why Equation (5.21) includes the "previous" vector q_{sk}, as well as the "final" vector $q_{s(k+1)+1}$.

We can simplify Equation (5.21) by eliminating the diagonal scaling Δ_k. We do so by multiplying both sides of Equation (5.21) on the right by Δ_k^{-1}:

$$\begin{aligned} M^{-1}AQ_k &= (1-\rho_{sk+1})q_{sk}e_1^T\Delta_k^{-1} + \underline{Q}_k\underline{\tilde{T}}_k\Delta_k^{-1} \\ &= \frac{1-\rho_{sk+1}}{\rho_{sk+1}\gamma_{sk+1}}q_{sk}e_1^T\Delta_k^{-1} + \underline{Q}_k\underline{\tilde{T}}_k\Delta_k^{-1}. \end{aligned} \tag{5.22}$$

If we then define the $s \times s$ tridiagonal matrix T_k as

$$T_k = \tilde{T}_k\Delta_k^{-1} \tag{5.23}$$

and the $s+1$ by s tridiagonal matrix $\underline{T}_k$ as

$$\underline{T}_k = \underline{\tilde{T}}_k\Delta_k^{-1} = \begin{pmatrix} \tilde{T}_k\Delta_k^{-1} \\ -\rho_{sk+s}^{-1}\gamma_{sk+s}^{-1} \cdot e_s^T \end{pmatrix}, \tag{5.24}$$

we may write Equation (5.21) in final matrix form as

$$M^{-1}AQ_k = \frac{1-\rho_{sk+1}}{\rho_{sk+1}\gamma_{sk+1}}q_{sk}e_1^T + \underline{Q}_k\underline{T}_k. \tag{5.25}$$

We will make use of this notation in Section 5.5.4 when developing formulae for other recurrences in LP-CA-CG.

5.5.3 The Gram matrix

In Section 4.3.1, we observed that the basis vectors in left-preconditioned Lanczos (both in standard Lanczos or CA-Lanczos) are not orthogonal with respect to the usual Euclidean inner product. Rather, the q_i basis vectors are orthogonal with respect to the M inner product (where M is the SPD preconditioner for the matrix A), so that $q_j^*Mq_i = \delta_{ij}$. Also, the z_i basis vectors are orthogonal with respect to the M^{-1} inner product, so that $z_j^*M^{-1}z_i = \delta_{ij}$. This means that one cannot use a QR factorization like Householder QR to orthogonalize the basis vectors. We solved this problem for CA-Lanczos in Section 4.3.4 by computing what we called a "Gram matrix" in each outer iteration. The Gram matrix is the matrix of inner products of the basis vectors. We show in Section 2.5 how computing the Gram matrix requires only one reduction over blocks, at a cost of $\log P$ messages in parallel, and how it can be used to M-orthogonalize the basis vectors. The $2s+1$ by $2s+1$ Gram matrix G_k is computed using both the current sets of $s+1$ not yet orthogonalized basis vectors $\underline{V}_k$ and $\underline{W}_k$, as well as the previous set of already M-orthogonalized s basis vectors $\underline{Q}_{k-1}$. In LP-CA-CG, we will use a combination of the Gram matrix and coefficients computed via a

recurrence relation to orthogonalize the current set of $s+1$ basis vectors against the previous set of s already orthogonal basis vectors. The properties of the Lanczos algorithm guarantee that in exact arithmetic, this suffices in order to make the current set of $s+1$ basis vectors orthogonal with respect to *all* the previously orthogonalized basis vectors.

For LP-CA-CG, Equation (5.26) gives the definition of the Gram matrix, analogous to Equation (4.21) in Section 4.3.4 for left-preconditioned CA-Lanczos:

$$G_k = \begin{pmatrix} Q_{k-1}^* Z_{k-1} & Q_{k-1}^* \underline{W}_k \\ \underline{V}_k^* Z_{k-1} & \underline{V}_k^* \underline{W}_k \end{pmatrix}. \tag{5.26}$$

The LP-CA-CG residual vectors z_i (which are unpreconditioned residuals) and q_i (which are preconditioned residuals) differ from the left-preconditioned CA-Lanczos basis vectors, in that the LP-CA-CG residual vectors do not necessarily have unit length in the M-norm. That means the CA-CG Gram matrix (Equation (5.26)) simplifies differently than the CA-Lanczos Gram matrix (Equation (4.21) in Section 4.3.4), even though we use the same notation for both. For example, in LP-CA-CG, $Q_{k-1}^* Z_{k-1} = D_{k-1}$, where D_{k-1} is an $s \times s$ diagonal matrix with

$$D_{k-1}(j,j) = \mu_{s(k-1)+j}. \tag{5.27}$$

This simplification assumes M-orthogonality of the residual vectors, so making it may result in loss of accuracy in finite-precision arithmetic. In CG, orthogonality of the residual vectors is less important than orthogonality of the basis vectors in Lanczos, so the simplification may be less risky. Note that if the current outer iteration is k, the entries $\mu_{s(k-1)+j}$ of this diagonal matrix have already been computed by outer iteration $k-1$ of CA-CG and are available for reuse in outer iteration k, when G_k is computed. The second simplification to the Gram matrix expression in Equation (5.26), namely

$$\underline{V}_k^* Z_{k-1} = (Q_{k-1}^* \underline{W}_k)^*,$$

makes no orthogonality assumptions on the residual vectors. We have already seen this simplification in left-preconditioned CA-Lanczos. If we include all the simplifications, the resulting Gram matrix is given by

$$G_k = \begin{pmatrix} D_{k-1} & Q_{k-1}^* \underline{W}_k \\ (Q_{k-1}^* \underline{W}_k)^* & \underline{V}_k^* \underline{W}_k \end{pmatrix}, \tag{5.28}$$

where D_{k-1} is given by Equation (5.27). Note that G_k is symmetric: $\underline{V}_k^* \underline{W}_k = \underline{V}_k^* M \underline{V}_k$, and M is symmetric (SPD, in fact). Furthermore, G_k is positive definite, as long as $\underline{V}_k$ is full rank and no diagonal entries of D_{k-1} are zero.

5.5.4 Vector coefficients

We derive LP-CA-CG from LP-CG3, and our derivation follows the pattern of our derivation of left-preconditioned CA-Lanczos in Section 4.3. Iteration i of LP-CG3 computes the following quantities directly, in the following partial order:

1. $\mu_i := \langle z_i, q_i \rangle$ (test convergence)

2. $w_i := Aq_i$
3. $v_i := M^{-1}(Aq_i)$ (depends on Aq_i)
4. $\nu_i := \langle Aq_i, q_i \rangle$ (depends on Aq_i)
5. z_{i+1} (depends on Aq_i, $\langle Aq_i, q_i \rangle$, and $\langle z_i, q_i \rangle$)
6. q_{i+1} (depends on $M^{-1}Aq_i$, $\langle Aq_i, q_i \rangle$, and $\langle z_i, q_i \rangle$)
7. x_{i+1} (depends on $\langle Aq_i, q_i \rangle$ and $\langle z_i, q_i \rangle$)

LP-CA-CG needs the same quantities at inner iteration $sk + j = i$, where $j = 1, 2, \ldots, s$ and k is the current outer iteration index. However, computing the first four quantities directly would require communication, assuming that the preconditioner is nontrivial. LP-CA-CG avoids this by means of the following approach:

- Express the vectors q_{sk+j} and $M^{-1}Aq_{sk+j}$ as linear combinations of the columns of $Q_{k-1} = [q_{s(k-1)+1}, \ldots, q_{sk}]$ and the columns of $\underline{V}_k = [v_{sk+1}, \ldots, v_{sk+s+1}]$.
- Express the vectors z_{sk+j} and Aq_{sk+j} $(= AM^{-1}z_{sk+j})$ as linear combinations of the columns of $Z_{k-1} = [z_{s(k-1)+1}, \ldots, z_{sk}]$ and the columns of $\underline{W}_k = [w_{sk+1}, \ldots, w_{sk+s+1}]$.
- The coefficients of those linear combinations form short (length $2s + 1$) vectors. Use those along with the $2s + 1$ by $2s + 1$ Gram matrix G_k to reconstruct the required inner products without communication, and use the values of those inner products to compute the vectors for the next inner iteration.

The computation of z_{sk+j+1} and q_{sk+j+1} for $j = 1, 2, \ldots, s$ in each outer iteration k, as well as the computation of the inner products $\langle z_{sk+j}, q_{sk+j} \rangle$ and $\langle Aq_{sk+j}, q_{sk+j} \rangle$ for $j = 1, 2, \ldots, s$ in outer iteration k, may be deferred until the end of that outer iteration. This may have performance advantages, both because it avoids synchronization points (the inner products are required to compute the vectors), and because it permits a BLAS 3 - style computation of the vectors, rather than the BLAS 1 computation in the standard algorithm. Deferring the computation of z_{sk+j+1} and q_{sk+j+1} until the end of outer iteration k also requires deferring the computation of the current approximate solution x_{sk+j+1} in the same way.

Analogously to the derivation in Section 5.4 of the nonpreconditioned version of CA-CG, we will use two sets of coefficients in our derivation of LP-CA-CG. For $j = 0, 1, \ldots, s$, the vector d_{sk+j} of $2s + 1$ coefficients satisfies

$$[Z_{k-1}, \underline{W}_k] d_{sk+j} = Aq_{sk+j}$$

and

$$[Q_{k-1}, \underline{V}_k] d_{sk+j} = M^{-1}Aq_{sk+j}.$$

For $j = 0, 1, \ldots, s + 1$, the vector g_{sk+j} of $2s + 1$ coefficients satisfies

$$[Z_{k-1}, \underline{W}_k] g_{sk+j} = z_{sk+j}$$

and

$$[Q_{k-1}, \underline{V}_k] g_{sk+j} = q_{sk+j}.$$

We derive below a recurrence relation for each of these coefficients. After that, in Section 5.5.6, we show how to use the d_{sk+j} and g_{sk+j} coefficients along with the Gram matrix G_k for communication-free computation of the values of the inner products $\langle z_{sk+j}, q_{sk+j} \rangle$ and $\langle A q_{sk+j}, q_{sk+j} \rangle$.

Coefficients d_{sk+j}

Here, we derive a recurrence for d_{sk+j}, for $j = 0, 1, 2, \ldots, s$. The vector d_{sk+j} of $2s+1$ coefficients satisfies two equations:

$$\begin{aligned} [Z_{k-1}, \underline{W}_k] d_{sk+j} &= A q_{sk+j}, \text{ and} \\ [Q_{k-1}, \underline{V}_k] d_{sk+j} &= M^{-1} A q_{sk+j}. \end{aligned} \tag{5.29}$$

We can use the recurrence

$$\rho_{sk+j} \gamma_{sk+j} A q_{sk+j} = -z_{sk+j+1} + \rho_{sk+j} z_{sk+j} + (1 - \rho_{sk+j}) z_{sk+j-1}. \tag{5.30}$$

for the CA-CG residual vectors to derive a recurrence for d_{sk+j}, just as we did to derive an analogous recurrence for left-preconditioned CA-Lanczos in Section 4.3.3.

The cases $j = 0$ and $j = 1$ are the base cases of the recurrence for d_{sk+j}. We can compute these cases directly, without recursion. For $j = 0$, we can find d_{sk} via

$$\begin{aligned} [Z_{k-1}, \underline{W}_k] d_{sk} &= A q_{sk} \\ &= \frac{1 - \rho_{sk}}{\rho_{sk} \gamma_{sk}} z_{sk-1} + \gamma_{sk}^{-1} z_{sk} - \rho_{sk}^{-1} \gamma_{sk}^{-1} z_{sk+1}, \end{aligned}$$

so that

$$d_{sk} = \left[0_{1,s-2}^T, \frac{1 - \rho_{sk}}{\rho_{sk} \gamma_{sk}}, \gamma_{sk}^{-1}, -\rho_{sk}^{-1} \gamma_{sk}^{-1}, 0_{1,s}^T \right]. \tag{5.31}$$

For $j = 1$, we can find d_{sk+1} via

$$\begin{aligned} [Z_{k-1}, \underline{W}_k] d_{sk+1} &= A q_{sk+1} \\ &= \underline{W}_k \underline{B}_k(:, 1:2) \end{aligned}$$

so that

$$d_{sk+1} = \left[0_{1,s}^T, \underline{B}_k(:,1), \underline{B}_k(:,2), 0_{1,s-1}^T \right]. \tag{5.32}$$

Now we need to find d_{sk+j} for $j = 2, 3, \ldots, s$. To do so, we begin by stating a fact that is obvious by inspection:

$$A q_{sk+j} \in \text{span}\{z_{sk-j+1}, \ldots, z_{sk}\} \oplus \text{span}\{w_{sk+1}, \ldots, w_{sk+j+1}\}, \qquad \text{for } j = 2, \ldots, s. \tag{5.33}$$

This means that d_{sk+j} has a particular sparsity structure, given by

- Only $d_{sk}(s-1 : s+1)$ are nonzero

- Only $d_{sk+1}(s+1:s+2)$ are nonzero
- For $j = 2, \ldots, s$, only $d_{sk+j}(1+s-j:1+s+j)$ are nonzero.

In particular, only for $j = s$ are the first and last entries of d_{sk+j} nonzero. The sparsity pattern of d_{sk+j} is the same as the vector of coefficients of the same name in unpreconditioned CA-CG (Algorithms 34 and 35). See Figure 5.1 in Section 5.4.4 for an illustration.

The sparsity pattern of d_{sk+j} matters not only for making computation more efficient, but also for the following derivation of a recurrence for d_{sk+j}. We begin with the definition of d_{sk+j-1}:

$$[Z_{k-1}, \underline{W}_k] d_{sk+j-1} = A q_{sk+j-1}, \tag{5.34}$$

where $j = 2, \ldots, s$. Multiply both sides of Equation (5.34) by AM^{-1}, using $Z_{k-1} = MQ_{k-1}$ and $\underline{W}_k = M\underline{V}_k$:

$$[AQ_{k-1}, A\underline{V}_k] d_{sk+j-1} = A(M^{-1} A q_{sk+j-1}). \tag{5.35}$$

The right-hand side of Equation (5.35) can be replaced by the recurrence (5.30) for the CG residual vectors, and multiplying through by A on the left:

$$\begin{aligned}[AQ_{k-1}, A\underline{V}_k] d_{sk+j-1} = \\ \frac{1-\rho_{sk+j-1}}{\rho_{sk+j-1}\gamma_{sk+j-1}} A q_{sk+j-2} + \gamma_{sk+j-1}^{-1} A q_{sk+j-1} - \rho_{sk+j-1}^{-1}\gamma_{sk+j-1}^{-1} A q_{sk+j}.\end{aligned} \tag{5.36}$$

Simplify the right-hand side of Equation (5.36) by applying the formula for d_{sk+j}:

$$\begin{aligned}[AQ_{k-1}, A\underline{V}_k] d_{sk+j-1} = \\ \frac{1-\rho_{sk+j-1}}{\rho_{sk+j-1}\gamma_{sk+j-1}} d_{sk+j-2} + \gamma_{sk+j-1}^{-1} d_{sk+j-1} - \rho_{sk+j-1}^{-1}\gamma_{sk+j-1}^{-1} d_{sk+j},\end{aligned} \tag{5.37}$$

and isolate the term d_{sk+j}, which we want to know:

$$\begin{aligned}\rho_{sk+j-1}^{-1}\gamma_{sk+j-1}^{-1} d_{sk+j} = \\ - [AQ_{k-1}, A\underline{V}_k] d_{sk+j-1} - \frac{1-\rho_{sk+j-1}}{\rho_{sk+j-1}\gamma_{sk+j-1}} d_{sk+j-2} - \gamma_{sk+j-1}^{-1} d_{sk+j-1}.\end{aligned} \tag{5.38}$$

The only term left to simplify in Equation (5.38) is $[AQ_{k-1}, A\underline{V}_k] d_{sk+j-1}$. We can do so by recalling the definition of the tridiagonal matrix $\underline{T}_k$, and the definition of the change-of-basis matrix $\underline{B}_k$:

$$AQ_{k-1} = \frac{1-\rho_{s(k-1)+1}}{\rho_{s(k-1)+1}\gamma_{s(k-1)+1}} z_{s(k-1)} e_1^* + \underline{Z}_{k-1}\underline{T}_{k-1} \tag{5.39}$$

and

$$A\underline{V}_k = \underline{W}_k \underline{B}_k + ? \cdot w_{sk+s+2} e_{s+1}^*. \tag{5.40}$$

In Equation (5.40), ? represents some coefficient whose value does not matter. The reason it does not matter is because for $j = 2, \ldots, s$, the first and last elements of d_{sk+j-1} are zero (see Equation (5.33)). As a result,

$$\begin{aligned}[AQ_{k-1}, A\underline{V}_k] d_{sk+j-1} &= \left[\underline{Z}_{k-1}\underline{T}_{k-1}, \underline{W}_k\underline{B}_k\right] \\ &= \left[Z_{k-1}T_{k-1}, \underline{T}_{k-1}(s+1, s) z_{sk+1} e_1^* + \underline{W}_k\underline{B}_k\right]\end{aligned} \tag{5.41}$$

Furthermore, $\underline{T}_{k-1}(s+1, s) = -\rho_{sk}^{-1}\gamma_{sk}^{-1}$ (see Equations (5.20) and (5.24)), so that

$$\begin{aligned}[AQ_{k-1}, A\underline{V}_k]d_{sk+j-1} &= \left[Z_{k-1}T_{k-1}, -\rho_{sk}^{-1}\gamma_{sk}^{-1}z_{sk+1}e_1^* + \underline{W}_k\underline{B}_k\right] \\ &= [Z_{k-1}, \underline{W}_k]\begin{pmatrix} T_{k-1} & 0_{s,s} \\ 0_{s+1,s} & -\rho_{sk}^{-1}\gamma_{sk}^{-1}e_1e_1^* + \underline{B}_k \end{pmatrix}.\end{aligned} \tag{5.42}$$

This makes it easy now to simplify Equation (5.38), so we can obtain a recurrence for d_{sk+j} for $j = 2, \ldots, s$:

$$\begin{aligned}d_{sk+j} = \rho_{sk+j-1}\gamma_{sk+j-1}\begin{pmatrix} -T_{k-1} & 0_{s,s} \\ 0_{s+1,s} & \rho_{sk}^{-1}\gamma_{sk}^{-1}e_1e_1^* - \underline{B}_k \end{pmatrix} d_{sk+j-1} \\ - (1 - \rho_{sk+j-1})d_{sk+j-2} - \rho_{sk+j-1}d_{sk+j-1}.\end{aligned} \tag{5.43}$$

We show the complete recurrence formula for d_{sk+j} as Equation (5.44):

$$d_{sk+j} = \begin{cases} [0_{1,s-2}^T, \frac{1-\rho_{sk}}{\rho_{sk}\gamma_{sk}}, \gamma_{sk}^{-1}, -\rho_{sk}^{-1}\gamma_{sk}^{-1}, 0_{1,s}^T] & \text{for } j = 0, \\ [0_{1,s}^T, \underline{B}_k(:,1), \underline{B}_k(:,2), 0_{1,s-1}^T] & \text{for } j = 1, \\ \rho_{sk+j-1}\gamma_{sk+j-1}\begin{pmatrix} -T_{k-1} & 0_{s,s} \\ 0_{s+1,s} & \rho_{sk}^{-1}\gamma_{sk}^{-1}e_1e_1^* - \underline{B}_k \end{pmatrix} d_{sk+j-1} & \\ \quad - \dfrac{1-\rho_{sk+j-1}}{d}_{sk+j-2} - \rho_{sk+j-1}d_{sk+j-1} & \text{for } j = 2, \ldots, s. \end{cases} \tag{5.44}$$

5.5.5 Coefficients g_{sk+j}

For $j = 0, 1, \ldots, s$, the vector g_{sk+j} of $2s+1$ coefficients satisfies two equations:

$$\begin{aligned}[Z_{k-1}, \underline{W}_k]g_{sk+j} &= z_{sk+j}, \text{ and} \\ [Q_{k-1}, \underline{V}_k]g_{sk+j} &= q_{sk+j}.\end{aligned} \tag{5.45}$$

We can use the recurrence for the CA-CG residual vectors to derive a recurrence for g_{sk+j}, just as we used the recurrence for the CA-Lanczos basis vectors to derive the analogous recurrence for the CA-Lanczos algorithm.

The two base cases of the recurrence are $j = 0$ and $j = 1$. There are furthermore two cases for $j = 0$: $k = 0$ (the first outer iteration), and $k > 0$ (subsequent outer iterations). For $k = 0$, $g_{sk} = 0$, else for $k > 0$, $g_{sk} = [0_{s-1}, 1, 0_{s+1}]^T$. For all k, $g_{sk+1} = [0_s, 1, 0_s]^T$. The recurrence, which holds for $j = 2, 3, \ldots, s$, is given by

$$g_{sk+j} = -\rho_{sk+j-1}\gamma_{sk+j-1}d_{sk+j-1} + \rho_{sk+j-1}g_{sk+j-1} + (1 - \rho_{sk+j-1})g_{sk+j-2}. \tag{5.46}$$

Thus, we can express g_{sk+j} for any valid k and j as follows:

$$g_{sk+j} = \begin{cases} 0_{2s+1} & k = 0 \text{ and } j = 0 \\ [0_{s-1}, 1, 0_{s+1}]^T & k > 0 \text{ and } j = 0 \\ [0_s, 1, 0_s]^T & j = 1, \\ -\rho_{sk+j-1}\gamma_{sk+j-1}d_{sk+j-1} + & \\ \quad \rho_{sk+j-1}g_{sk+j-1} + & j = 2, 3, \ldots, s. \\ \quad (1 - \rho_{sk+j-1})g_{sk+j-2} & \end{cases} \tag{5.47}$$

5.5.6 Inner products

Each inner iteration $sk + j$ of left-preconditioned CA-CG requires the value of two different inner products:

- $\langle z_{sk+j}, q_{sk+j} \rangle$
- $\langle Aq_{sk+j}, q_{sk+j} \rangle$

In fully communication-avoiding CA-CG, these inner products are not computed directly. Rather, they are computed using the above d_{sk+j} and g_{sk+j} coefficients and the Gram matrix G_k. The first inner product's value may be computed by

$$\langle z_{sk+j}, q_{sk+j} \rangle = g_{sk+j}^* G_k g_{sk+j}. \tag{5.48}$$

Recall that G_k is symmetric. It is also positive definite, as long as $\underline{V}_k$ is full rank. Thus, as long as $\underline{V}_k$ is sufficiently well conditioned, $g_{sk+j}^* G_k g_{sk+j} > 0$ as long as $g_{sk+j} \neq 0$, even in finite-precision arithmetic. The second inner product's value may be computed by

$$\langle Aq_{sk+j}, q_{sk+j} \rangle = g_{sk+j}^* G_k d_{sk+j}. \tag{5.49}$$

5.5.7 LP-CA-CG algorithm

Algorithm 37 shows left-preconditioned CA-CG in its entirety. Note that the vector operations are not strictly necessary until the end of each outer iteration. That means they can be coalesced into a single BLAS 3 - type operation, which also would reduce the number of synchronization points.

5.6 Summary

In this chapter, we presented both unpreconditioned and preconditioned communication-avoiding versions of CG. They improve on previous work first in how they avoid communication:

- Their use of the matrix powers kernel (Sections 2.1 and 2.2) means that they send a factor of $\Theta(s)$ fewer messages in parallel for the sparse matrix operations. They also read the sparse matrix (and preconditioner, if applicable) from slow memory a factor of s fewer times (each).
- If parallel messages are expensive, the algorithms can also avoid communication in the vector operations, by replacing the dot products with block reductions (the Gram matrix computation). This means they require a factor of s fewer messages in parallel than standard CG for the vector operations.

Second, the algorithms improve on accuracy and numerical stability:

- They can avoid communication in the sparse matrix operations with a much larger class of preconditioners (described in Section 2.2). Previous s-step CG algorithms were limited to preconditioners that require no communication.

Algorithm 37 Left-preconditioned CA-CG

Input: $Ax = b$: $n \times n$ SPD linear system
Input: Initial guess x_1 to $Ax = b$
Output: Approximate solution x_i to $Ax = b$, with $i \in \{1, 2, \dots\}$

1: $z_0 := 0, z_1 := b - Ax_0$ ▷ z_1 is the unpreconditioned initial residual
2: $q_0 := 0, q_1 := M^{-1}z_1$ ▷ q_1 is the preconditioned initial residual
3: **for** $k = 0, 1, \dots$ until convergence **do**
4: Compute $\underline{V}_k$ via matrix powers kernel ▷ $\mathcal{R}(\underline{V}_k) = \mathcal{K}_{s+1}(M^{-1}A, q_{sk+1})$
5: Compute $\underline{W}_k$ via matrix powers kernel ▷ $M^{-1}\underline{W}_k = \underline{V}_k$
6: $G_{k,k-1} := \underline{V}_k^* Z_{k-1}$, $G_{kk} := \underline{V}_k^* \underline{W}_k$ ▷ 2 block reductions; can compute by reading $\underline{V}_k$ only once, saving memory traffic
7: $G_k := \begin{pmatrix} D_{k-1} & G_{k,k-1}^* \\ G_{k,k-1} & G_{kk} \end{pmatrix}$ (Gram matrix, via Eq. (5.28))
8: **for** $j = 1$ to s **do**
9: Compute d_{sk+j} (Eq. (5.44)) and g_{sk+j} (Eq. (5.47))
10: $\mu_{sk+j} := g_{sk+j}^* G_k g_{sk+j}$ (Eq. (5.48)); test for convergence ▷ $\mu_{sk+j} = \langle z_{sk+j}, q_{sk+j} \rangle$
11: $\nu_{sk+j} := g_{sk+j}^* G_k d_{sk+j}$ (Eq. (5.49)) ▷ $\nu_{sk+j} = \langle Aq_{sk+j}, q_{sk+j} \rangle$
12: **if** $sk + j = 1$ **then**
13: $\rho_{sk+j} := 1$
14: **else**
15: $\rho_{sk+j} := \left(1 - \frac{\gamma_{sk+j}}{\gamma_{sk+j-1}} \cdot \frac{\langle z_{sk+j}, q_{sk+j} \rangle}{\langle z_{sk+j-1}, q_{sk+j-1} \rangle} \cdot \frac{1}{\rho_{sk+j-1}}\right)^{-1}$
16: **end if** ▷ Nothing from top of j loop to here requires communication
17: $x_{sk+j+1} := \rho_{sk+j}(x_{sk+j} + \gamma_{sk+j} q_{sk+j}) + (1 - \rho_{sk+j}) x_{sk+j-1}$
18: $u_{sk+j} := [Q_{k-1}, \underline{V}_k] d_{sk+j}$
19: $q_{sk+j+1} := \rho_{sk+j}(q_{sk+j} - \gamma_{sk+j} u_{sk+j}) + (1 - \rho_{sk+j}) q_{sk+j-1}$
20: $y_{sk+j} := [Z_{k-1}, \underline{W}_k] d_{sk+j}$
21: $z_{sk+j+1} := \rho_{sk+j}(z_{sk+j} - \gamma_{sk+j} y_{sk+j}) + (1 - \rho_{sk+j}) z_{sk+j-1}$ ▷ None of the last 5 lines require parallel communication or synchronization
22: **end for**
23: **end for**

- They can use any of the s-step bases we discuss in Chapter 7, with or without preconditioning. A better basis allows larger basis lengths s without sacrificing numerical stability or requiring expensive reorthogonalization of the basis vectors. As a result, our algorithms have larger potential speedups.

Numerical and benchmarking studies are required, but our successes with CA-GMRES (Sections 3.4 and 3.5) convince us of the potential of our CA-CG algorithms.

Chapter 6

Communication-avoiding nonsymmetric Lanczos and BiCG

In this chapter, we briefly discuss how one might go about developing versions of iterative methods related to nonsymmetric Lanczos iteration (for solving $Ax = \lambda x$) or BiCG (for solving $Ax = b$), that avoid communication in the sparse matrix operations and the vector operations.

Previous authors have come up with an s-step version of nonsymmetric Lanczos, though their algorithm did not use optimized kernels and required use of the monomial s-step basis. Using techniques discussed in this thesis, we outline in this chapter how one might remove these restrictions, for both nonsymmetric Lanczos (for solving eigenvalue problems) and BiCG (the Method of Biconjugate Gradients, for solving linear systems). Also in this chapter, we point out alternate ways to optimize algorithms related to nonsymmetric Lanczos and BiCG, including using the matrix powers kernel to accelerate the lookahead process.

In Section 6.1 of this chapter, we discuss nonsymmetric Lanczos, including the standard version of the algorithm, an existing s-step variant, and our suggested approach to develop a communication-avoiding version analogous to CA-Arnoldi and symmetric CA-Lanczos. We also suggest how one could use the matrix powers kernel to accelerate lookahead in nonsymmetric Lanczos. Finally, in Section 6.2, we cite our colleague Erin Carson's development of a new communication-avoiding version of BiCG.

6.1 Nonsymmetric Lanczos

6.1.1 The standard algorithm

Nonsymmetric Lanczos iteration, also called *Lanczos biorthogonalization*, is an iterative method for solving sparse nonsymmetric eigenvalue problems. It is shown here as Algorithm 38. (Note that we use a nonstandard notation for the algorithm. This will help us compare it in Section 6.1.6 to left-preconditioned symmetric Lanczos.) After k iterations, if the method does not break down (see Section 6.1.2), it produces a $k \times k$ possibly nonsym-

Algorithm 38 Nonsymmetric Lanczos iteration

Input: $n \times n$ matrix A and two length n starting vectors z_1 and q_1 ▷ q_k vectors span $\mathcal{K}(A, q_1)$, and z_k vectors span $\mathcal{K}(A^*, z_1)$

Require: $\langle z_1, q_1 \rangle = 1$

$q_0 := 0$, $z_0 := 0$
for $k = 1$ to convergence **do**
  $q'_{k+1} := Aq_k - \gamma_k q_{k-1}$
  $z'_{k+1} := A^* z_k - \beta_k z_{k-1}$
  $\alpha_k := \langle z'_{k+1}, q_k \rangle$
  $q''_{k+1} := q'_{k+1} - \alpha_k q_k$
  $z''_{k+1} := z'_{k+1} - \bar{\alpha}_k q_k$
  $\omega_{k+1} := \langle z''_{k+1}, q''_{k+1} \rangle$
  if $\omega_{k+1} = 0$ **then**
    The algorithm has broken down (see Section 6.1.2 for the causes and possible ways to recover from a breakdown).
  end if
  $\beta_{k+1} := \sqrt{|\omega_{k+1}|}$
  $\gamma_{k+1} := \omega_{k+1}/\beta_{k+1}$ ▷ One may choose any nonzero β_{k+1} and γ_{k+1} such that $\beta_{k+1}\gamma_{k+1} = \omega_{k+1}$; the choice shown here is typical.
  $q_{k+1} := q''_{k+1}/\beta_{k+1}$
  $z_{k+1} := z''_{k+1}/\gamma_{k+1}$
end for

metric tridiagonal matrix

$$T_k = \begin{pmatrix} \alpha_1 & \gamma_2 & \dots & 0 \\ \beta_2 & \alpha_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \gamma_k \\ 0 & \dots & \beta_k & \alpha_k \end{pmatrix}. \tag{6.1}$$

The eigenvalues (called either "Ritz values" or "Petrov values") of T_k are approximations of the eigenvalues of A. These k iterations also produce two sets of k basis vectors each: one set $\{q_1, q_2, \dots, q_k\}$ which is a basis for a Krylov subspace involving A, and the other set $\{z_1, z_2, \dots, z_k\}$ which is a basis for a Krylov subspace involving A^*. For more details on nonsymmetric Lanczos, see [16].

The *method of biconjugate gradients*, also called BiCG, is to nonsymmetric Lanczos as ordinary CG is to symmetric Lanczos. Several iterative methods for solving sparse linear systems originate from BiCG. "Originate" may mean either that they are modifications of BiCG in order to improve numerical stability, or that they are mathematically related (for example, they may use the same two Krylov subspaces that nonsymmetric Lanczos does, but in a different way). These algorithms include QMR [103], TFQMR [102], CGS [214], and BiCGSTAB [226], among others. For more details, see [23].

6.1.2 Breakdown of nonsymmetric Lanczos

Symmetric Lanczos, Arnoldi, and related Krylov methods such as CG, all "break down" in exact arithmetic, when the maximum dimension of their Krylov subspace has been reached. This dimension may be smaller than the dimension of the matrix. Regardless, one often calls this a "lucky breakdown," because it means that an invariant subspace has been reached. When solving linear systems $Ax = b$ for nonsingular A, it means one has computed the exact solution (hence "lucky"). When solving eigenvalue problems, it means one has reached an invariant subspace and the subspace's corresponding eigenvalues have converged.

Nonsymmetric Lanczos does have an analogous "lucky breakdown" condition, but it also has another case, which Wilkinson termed a "serious breakdown" (see e.g., [191, p. 107]). Both possibilities are summarized in e.g., Parlett et al. [191], and they both relate to the terms in the inner product

$$\omega_{k+1} := \langle z''_{k+1}, q''_{k+1} \rangle \tag{6.2}$$

in Line 8 of Algorithm 38. The first case, when either of the vectors z''_{k+1} or q''_{k+1} are zero, is analogous to the aforementioned "lucky breakdown" in Arnoldi or symmetric Lanczos. The second case – the "serious breakdown" mentioned above – is when neither of the vectors in the inner product in Equation (6.2) are zero, but ω_{k+1} is zero.

The possibility of "serious breakdown" has important consequences for an s-step version of nonsymmetric Lanczos, as we will discuss below. In fact, s-step versions of nonsymmetric Lanczos may skip over serious breakdowns, but they also may make it harder to distinguish between the different breakdown cases and the third case of an ill-conditioned s-step basis.

6.1.3 Lookahead and the s-step basis

Nonsymmetric Lanczos requires modifications in order to recover from serious breakdown. One option is *lookahead*, which is given an overview e.g., in [16]. Lookahead involves "skipping over" the offending iteration in the algorithm, by constructing different bases and relaxing the requirement for that step that the Lanczos tridiagonal matrix be tridiagonal. Two papers on this subject are Parlett et al. [191] and Freund et al. [104].

We mention lookahead because the lookahead approaches mentioned above entail the construction of either a the monomial (in the case of Parlett et al.) or Chebyshev (in the case of Freund et al.) s-step basis. As far as we know, no one has suggested accelerating this computation by using an optimized matrix powers kernel. We make this suggestion now as a direction for future work.

Since serious breakdown may not occur often (or at all), it is not clear whether the cost of setting up the matrix powers kernel is justified. As Section 2.1 indicates, this setup cost may be significant. However, one could make an s-step version of nonsymmetric Lanczos with lookahead, and exploit the same matrix powers kernel for both the s-step basis and lookahead. We will discuss serious breakdown and s-step versions of nonsymmetric Lanczos below.

6.1.4 s-step nonsymmetric Lanczos

Kim and Chronopoulos [153] develop an s-step version of nonsymmetric Lanczos iteration. The authors sought both to improve data locality, and to reduce the number of global synchronization steps. Their algorithm replaces the $2s$ inner products in s iterations of standard nonsymmetric Lanczos, with a single block inner product (a matrix multiply of $s \times n$ by $n \times s$, where n is the dimension of A). That block inner product can be implemented with a single reduction operation. The two "tall and skinny" $n \times s$ matrices involved in this block inner product are generated by two independent invocations of the s-step monomial basis. The authors intend by their use of an s-step basis, as Chronopoulos and his coauthors intended in their other s-step methods, to expose the data locality latent in that operation. However, Kim and Chronopoulos do not have a specialized matrix powers kernel for computing the s-step basis. Just like s-step CG or symmetric Lanczos, the algorithm requires a factor of $\Theta(s)$ additional vectors to be stored.[1]

The s-step nonsymmetric Lanczos iteration computes the same approximate eigenvalues and eigenvectors in the same number of iterations, in exact arithmetic, as does standard nonsymmetric Lanczos. However, unlike standard nonsymmetric Lanczos, this s-step version produces a block (nonsymmetric) tridiagonal matrix, where the blocks are $s \times s$. This does not significantly increase the computational cost, since s is typically a small constant.

Kim and Chronopoulos' algorithm is restricted to the monomial basis. Their derivation of the method (see [153, p. 365], equations (3.7.1), (3.7.2), (3.8.1), and (3.7.2)) depends on the use of the monomial basis. (It seems, though we have not proven this, that this dependency is related to the monomial basis being the only basis (modulo scaling the vectors) for which the matrix of moments is equal to the Gram matrix of the basis vectors.) As we mentioned before in the context of CG and symmetric Lanczos, limiting oneself to the monomial basis has the potential to hinder numerical stability. It also may limit performance, by forcing s to be smaller than it should be for optimal matrix powers kernel performance. Also, the authors perform no scaling of the matrix A, which has the potential to make this worse, as we discuss in Chapter 7.

Another possible deficiency in the algorithm is that it only makes available the s-step basis vectors. It does not compute the actual "Lanczos vectors," which are mutually biorthogonal in exact arithmetic, and the vectors it does compute are not mutually biorthogonal. In standard nonsymmetric Lanczos, the Lanczos biorthogonal vectors do tend to lose their mutual biorthogonality in finite-precision arithmetic, sometimes making re-biorthogonalization necessary. (See Day [69] for a discussion of different re-biorthogonalization approaches, and an argument for re-biorthogonalizing under a "semiduality" condition analogous to the semiorthogonality condition in symmetric Lanczos (see e.g., [16]).) In Kim and Chronopoulos' algorithm, this loss of mutual biorthogonality may be exacerbated by the Lanczos vectors being represented implicitly, as a collection of vectors that are not mutually biorthogonal. This may also complicate the process of detecting whether re-biorthogonalization is necessary.

[1] While symmetric and nonsymmetric Lanczos both may need to keep the Lanczos basis vectors from all past iterations in order to perform re(bi)orthogonalization, all but the previous two vectors do not need to be accessed unless re(bi)orthogonalization takes place. Thus, they need not occupy space even in slow memory, for example; they may be pushed to a backing store.

6.1.5 Does s-step Lanczos need lookahead?

Kim and Chronopoulos point out that their use of an s-step basis may sometimes overcome the breakdown conditions that may make nonsymmetric Lanczos unreliable in some case. In particular, they point out [153, p. 368] that in exact arithmetic, if breakdown were to occur at a particular iteration k of standard nonsymmetric Lanczos, then in s-step nonsymmetric Lanczos, that same breakdown could only occur if k is a multiple of s. This is because their algorithm uses bases resulting in a block tridiagonal Lanczos matrix, which has the same effect as performing lookahead once every s iterations of the standard algorithm. Detecting a breakdown in s-step nonsymmetric Lanczos in exact arithmetic amounts to making sure that none of the block inner products of the two s-step bases have determinant zero. However, it is unclear how to translate the above exact-arithmetic observation into a useful test when working in finite-precision arithmetic, especially given the loss of mutual biorthogonality mentioned in Section 6.1.4. Kim and Chronopoulos do not address this issue in their paper.

6.1.6 Towards nonsymmetric CA-Lanczos

Algorithm 39 Left-preconditioned symmetric Lanczos iteration

Input: $n \times n$ SPD matrix A
Input: $n \times n$ SPD preconditioner M^{-1}
Input: Length n starting vector z_1 with $\|z_1\|_2 = 1$ ▷ q_k vectors span $\mathcal{K}(M^{-1}A, M^{-1}z_1)$, and z_k vectors span $\mathcal{K}(AM^{-1}, z_1)$

1: $q_0 := 0$, $z_0 := 0$
2: $q_1 := M^{-1}z_1$
3: **for** $k = 1$ to convergence **do**
4: $\quad z'_{k+1} := Aq_k - \beta_k z_{k-1}$
5: $\quad \alpha_k := \langle z'_{k+1}, q_k \rangle$
6: $\quad z''_{k+1} := z'_{k+1} - \alpha_k z_k$
7: $\quad q''_{k+1} := M^{-1}z''_{k+1}$
8: $\quad \omega_{k+1} := \sqrt{\langle z''_{k+1}, q''_{k+1} \rangle}$
9: $\quad q_{k+1} := q''_{k+1}/\omega_{k+1}$
10: $\quad z_{k+1} := z''_{k+1}/\omega_{k+1}$
11: **end for**

For the reasons mentioned in Section 6.1.4, we think it necessary to develop a "nonsymmetric CA-Lanczos" algorithm, rather than to use the one of Kim and Chronopoulos. We do not develop this algorithm in full in this thesis; we leave it for future work. However, doing so should require only small changes to the left-preconditioned symmetric CA-Lanczos algorithm of Section 4.3.4 (Algorithm 29). The only major changes, in fact, are to what the two invocations of the matrix powers kernel per outer iteration should compute. We will show in the paragraphs below why we believe this to be the case.

We first compare the z_k and q_k basis vectors in the left-preconditioned version of (standard) symmetric Lanczos (Algorithm 26 in Section 4.3.1), with the two sets of basis vectors in (nonpreconditioned) nonsymmetric Lanczos. We reproduce left-preconditioned symmetric Lanczos here as Algorithm 39, with a few small changes in notation to make clear the

similarities between the two algorithms. In Algorithm 39, the basis vectors q_1, q_2, ..., q_k span the Krylov subspace

$$\mathcal{K}_k(M^{-1}A, q_1) = \text{span}\{q_1, M^{-1}Aq_1, (M^{-1}A)^2q_1, \ldots, (M^{-1}A)^{k-1}q_1\},$$

and the basis vectors z_1, z_2, ..., z_k span the Krylov subspace

$$\mathcal{K}_k(AM^{-1}, z_1) = \text{span}\{z_1, AM^{-1}z_1, (AM^{-1})^2z_1, \ldots, (AM^{-1})^{k-1}z_1\}.$$

In nonsymmetric Lanczos, the basis vectors q_1, q_2, ..., q_k span the Krylov subspace

$$\mathcal{K}_k(A, q_1) = \text{span}\{q_1, Aq_1, A^2q_1, \ldots, A^{k-1}q_1\},$$

and the basis vectors z_1, z_2, ..., z_k span the Krylov subspace

$$\mathcal{K}_k(A^*, z_1) = \text{span}\{z_1, A^*z_1, (A^*)^2z_1, \ldots, (A^*)^{k-1}z_1\}.$$

Abstractly, in each algorithm there are two different Krylov subspaces: $\mathcal{K}_k(A_1, q_1)$ and $\mathcal{K}_k(A_2, z_1)$. In symmetric Lanczos, $A_1 = M^{-1}A$ and $A_2 = AM^{-1}$, and in nonsymmetric Lanczos, $A_1 = A$ and $A_2 = A^*$. In fact, in both cases $A_1 = A_2^*$ (since in symmetric Lanczos, we assume that A and M^{-1} are both symmetric). The only difference is that in symmetric Lanczos, the basis vectors q_1 and z_1 are related by preconditioning: $q_1 = M^{-1}z_1$. We call z_1 in symmetric Lanczos therefore the "unpreconditioned starting vector," and q_1 the "preconditioned starting vector." In nonsymmetric Lanczos, the two vectors need only satisfy $\langle z_1, q_1\rangle = 1$.

Another similarity between left-preconditioned symmetric Lanczos and nonsymmetric Lanczos are the inner product computations. In Algorithm 39, these are lines 5 and 8. In Algorithm 38, these are lines 5 and 8. (The computation of β_{k+1} in symmetric Lanczos corresponds to the computation of ω_{k+1} in nonsymmetric Lanczos.) The reason the corresponding inner products in the two algorithms look so similar is because they are both implicitly computing an orthogonalization. In the case of symmetric Lanczos, they implicitly orthogonalize the q_k vectors with respect to the M inner product, and the z_k vectors with respect to the M^{-1} inner product. In the case of nonsymmetric Lanczos, they implicitly *bi*orthogonalize the q_k vectors against the z_k vectors, so that $\langle z_j, q_i\rangle$ is 1 if $i = j$, and is zero otherwise.

These two similarities mean that a nonsymmetric CA-Lanczos algorithm may use an analogous Gram matrix computation to perform the biorthogonalization, just as left-preconditioned symmetric CA-Lanczos computes a Gram matrix and uses it to M-orthogonalize the q_{sk+j} vectors and M^{-1}-orthogonalize the z_{sk+j} vectors (see Algorithm 29 and Section 4.3.4).

The basic structure of the algorithm is outlined above. We do not discuss further details in this thesis, because we do not have time to finish all the details and develop a robust algorithm. We discuss below why making the algorithm robust may be more challenging than previous authors may have thought.

6.1.7 CA-Lanczos and breakdown

We believe the interaction between possible breakdown and the s-step part of the method in finite-precision arithmetic may be challenging to understand. In particular, it may be difficult to distinguish between the following three conditions:

1. "Lucky" breakdown
2. Serious breakdown
3. One or both of the s-step bases in an outer iteration are not numerically full rank

or to formulate a strategy that makes it unnecessary to distinguish between these conditions. In this thesis, we will only speculate on some strategies for distinguishing between them. We leave refinement of this speculation for future work. We can think of at least two possible approaches:

1. Run "recklessly," without breakdown tests, until some kind of failure (overflow via division by zero, for example). Then backtrack, run a few iterations of standard nonsymmetric Lanczos with careful tests for breakdown, and use one of the existing lookahead techniques if breakdown is detected. If no breakdown is detected, then assume that one or both of the s-step bases were ill-conditioned; reduce s and / or otherwise improve the basis, and continue with nonsymmetric CA-Lanczos.
2. If the block inner product (Gram matrix) in nonsymmetric CA-Lanczos is not numerically full rank in a particular outer iteration, first test each of the two s-step bases (via rank-revealing TSQR – see Section 2.4.8) to see if they are both numerically full rank. If they are, assume that serious breakdown has occurred. Adjust s as necessary in order to look ahead of the breakdown spot.

6.1.8 Future work

The above discussion should make it clear that nonsymmetric CA-Lanczos requires more work. For example, the s-step nonsymmetric Lanczos algorithm of Kim and Chronopoulos needs further investigation, in particular to see if the breakdown condition they discuss is sensible in finite-precision arithmetic. The nonsymmetric CA-Lanczos algorithm should be completed, and compared with the algorithm of Kim and Chronopoulos, both in terms of accuracy and in terms of performance. We do not have time to solve these problems in this thesis, but we plan to investigate them further at a later time.

6.2 BiCG

The Method of Biconjugate Gradients (BiCG) is a Krylov subspace method for solving systems of nonsymmetric linear equations. It is mathematically equivalent to nonsymmetric Lanczos, in that it produces two sets of basis vectors for the same two Krylov subspaces as in the latter iteration. The BiCG basis vectors differ only by scaling from the nonsymmetric Lanczos basis vectors.

When we discussed with our colleague Erin Carson the suggestions of Section 6.1.6, she wrote out a "Communication-Avoiding BiCG" algorithm (CA-BiCG) and performed preliminary numerical experiments with it (see [47]), which appear successful. We will not show the algorithm or experiment with it in this thesis, though we do plan on collaborating with our colleague as needed to develop and improve the algorithm.

Chapter 7

Choosing the basis

This chapter discusses the s-step basis computed by the matrix powers kernel. This basis affects the stability of the s-step Krylov methods discussed in this thesis more than anything else. We deferred discussion of the choice of basis until this chapter, because this subject is complicated enough to deserve study on its own, yet we can describe our iterative methods algebraically without considering the type of basis.

Our main contributions in this chapter are:

- We are the first to suggest scaling the rows and columns of the matrix A as a preprocessing step for s-step Krylov methods. This improves both numerical stability and the convergence rate, especially when A is badly scaled.
- We are the first to show how to incorporate preconditioning into an s-step basis other than the monomial basis.

Scaling the matrix A avoids rapid increase or decrease in the lengths of the s-step basis vectors. In the worst case, this may cause underflow resp. overflow in finite-precision arithmetic. Even without underflow or overflow, it may cause inaccuracy in the s-step method. Previous authors observed this phenomenon, but proposed fixing it by scaling each basis vector in turn after computing it. This requires a factor of s additional communication per computation of the s-step basis. In contrast, scaling the matrix A need only be done once, requires work equivalent to a small number of sparse matrix-vector multiplies, and greatly improves numerical stability of the iterative method. In fact, for CA-GMRES (Section 3.4), we found that scaling the matrix A could improve numerical stability more than choosing the "right" s-step basis, if the matrix A is badly scaled (see Section 3.5). However, if A is not badly scaled but ill-conditioned, the type of basis matters more. Thus, scaling the matrix A is not enough, nor is choosing the right basis enough: the two must be done together.

As we discussed in Section 2.2, preconditioning is important for accelerating the convergence of Krylov methods. Previous work in s-step Krylov methods only showed how to apply conditioning when using the monomial basis $v, Av, A^2v, \ldots, A^sv$. In Section 2.2, we showed the form of the s-step basis for different kinds of preconditioning, including split, left, and right preconditioning. In this chapter, we show the form of the "change-of-basis matrix" $\underline{B}$, for the different types of preconditioned s-step basis. The change-of-basis matrix relates to the so-called "Krylov decomposition," in a way that we will describe below.

7.1 Chapter outline

We begin with Section 7.2, where we review notation for an s-step basis, and explain how the numerical rank of the basis affects numerical stability and convergence of s-step Krylov methods. In Section 7.3, we present the three types of s-step basis – monomial, Newton, and Chebyshev – used in previous research in s-step Krylov methods. In this thesis, we only experiment with the monomial and Newton bases, but we describe the Chebyshev basis as another possibility. Next, in Section 7.4, we explain more formally why the condition number of the s-step basis affects the numerical stability and convergence of s-step Krylov methods. We also summarize known lower bounds on the growth of the basis condition number with respect to s, for the monomial, Newton, and Chebyshev bases. In addition, we talk about how the accuracy of orthogonalizing the s-step basis against previously orthogonalized basis vectors (as one does, for example, in CA-GMRES (Section 3.4), CA-Arnoldi (Section 3.3), and CA-Lanczos (Section 4.2)) influences numerical stability. In Section 7.5, we discuss the phenomenon of rapid growth or decrease in the lengths of the s-step basis vectors, and propose scaling the rows and columns of the matrix A in order to prevent this problem. We suggest two different scaling schemes – balancing for solving eigenvalue problems, and equilibration when solving linear systems – and we summarize inexpensive and effective algorithms for each of these schemes. Finally, in Section 7.6, we suggest future directions for research in s-step basis selection.

7.2 Introduction

The matrix powers kernel discussed in Section 2.1 requires a choice of *s-step basis*. If we say "basis" in this chapter, we mean s-step basis, unless otherwise specified. We can express the choice of basis as a set of $s + 1$ polynomials $p_0(z)$, $p_1(z)$, ..., $p_s(z)$, where z is an abstract variable, and p_j has degree j for $j = 0, 1, \ldots, s$. Our presentation assumes that the polynomials have complex coefficients and that the matrix A is complex, unless we say otherwise, but the discussion applies also if the matrix is complex and the coefficients real, or if both matrix and coefficients are real.

The matrix powers kernel uses these polynomials, along with the starting vector v, to generate the s basis vectors $p_0(A)v$, $p_1(A)v$, ..., $p_s(A)v$. The simplest such basis is the *monomial basis*, where $p_j(z) = z^j$. We have already mentioned the monomial basis and other choices of basis in Section 3.2.2, in the context of Arnoldi iteration, but in this chapter, we will discuss three bases in detail: the monomial basis (Section 7.3.1), the Newton basis (Section 7.3.2), and the Chebyshev basis (Section 7.3.3).

Any set of polynomials may be used for the basis, as long as p_j has degree j for $j = 0, 1, \ldots, s$. However, both performance and numerical stability concerns constrain the properties of the basis. For better performance, we prefer that the polynomial basis satisfy either a two-term or a three-term recurrence. This means that $p_{j+1}(A)v$ be computed as a linear combination of $Ap_j(A)v$, $p_j(A)v$, and possibly also $p_{j-1}(A)v$, for $j > 0$. Longer recurrences mean more floating-point operations, and they may also mean less fast memory available for cache blocks, or more memory traffic at smaller levels of the memory hierarchy. We discuss the performance issues in detail in Section 2.1; this chapter only covers numerical

stability issues.

In order for the basis in exact arithmetic to be a basis in finite-precision arithmetic, it should be numerically full rank. Experiments and theory show that as the basis length s increases, the basis vectors tend to become increasingly dependent. The rate of this process depends on the choice of basis. Eventually, the basis reaches a length at which it becomes numerically rank deficient, even though in exact arithmetic it still forms a basis. This phenomenon is not unique to the s-step bases we use in this work. In fact, it seems to occur for any Krylov subspace basis, even when the basis is guaranteed to be orthogonal in exact arithmetic (such as in symmetric Lanczos), as long as the basis is constructed without careful reorthogonalization.[1] However, typical s-step bases lose their independence in finite-precision arithmetic after far fewer vectors. We study this problem because it often happens for basis lengths s we would like to use, from a performance perspective.

Loss of independence of the s-step basis in finite precision has two causes:

1. Breakdown of the Krylov subspace: the Krylov subspace cannot be extended further, even in exact arithmetic. This has nothing to do with the use of an s-step iterative method.

2. Dependence caused by the s-step basis itself, which has nothing to do with lucky breakdown.

Once we have determined that a particular s-step basis is numerically rank-deficient, there are many ways to distinguish between these two cases. For example, one may switch from an s-step iterative method to its corresponding standard iterative method for s iterations of the latter. If the standard iterative method breaks down, then the fault lies with the Krylov subspace itself and not with the s-step basis. (It may not even be a fault; for example, when solving $Ax = b$, if the initial guess is zero and b is an eigenvector of A, breakdown will result in one iteration, because the exact solution *is* b.) Otherwise, the s-step basis is to blame. We will not consider the first case (breakdown due to the Krylov subspace) further in this chapter.

Loss of independence of the s-step basis vectors may either cause a catastrophic convergence failure of the s-step iterative method, or slow down its convergence rate. This is because Krylov subspace methods converge as the dimension of their Krylov subspace increases. If the s-step basis is not numerically full rank, the Krylov subspace's dimension either does not increase, or increases more slowly than it should. Some postprocessing of the s-step basis, such as a stable rank-revealing decomposition and the use of pseudoinverses when factors of the basis are required, may help prevent a catastrophic failure of the method. Even simple orthogonalization of the basis helps somewhat, as the work of Swanson and Chronopoulos [220], who added MGS orthogonalization to their basis computation to improve stability and allow larger values of s. However, postprocessing cannot recover search directions lost in the s-step basis computation. There are two solutions for this problem:

1. Choosing a better-conditioned s-step basis

2. Using higher-precision arithmetic

[1] For a full discussion in the context of symmetric Lanczos, see e.g., [176].

If the matrix and vectors are stored, read, and written in higher precision, the memory traffic will at least double. Floating-point performance will be reduced, by some hardware- and software-dependent factor. More judicious use of higher-precision arithmetic may reduce these costs, while improving stability somewhat. Choosing a better basis is almost always a good idea. This is because it has little performance overhead (as shown in Section 2.1), yet often allows the basis length s to be chosen larger, which may improve performance (depending on the structure of the matrix).

We will define what we mean by "numerically full rank" and "well conditioned" more formally in Section 7.4. There, we will summarize known results about the growth of this condition number with respect to the basis length for different bases. We will also discuss how the condition number might be affected by orthogonalization against previously orthogonalized basis vectors, in our versions of Lanczos and Arnoldi iteration.

7.3 Three different bases

In this section, we present three bases – monomial, Newton, and Chebyshev – that have been used in s-step Krylov methods. We summarize why they were chosen by previous authors, and briefly mention advantages and disadvantages of each basis. We also show the change of basis matrix $\underline{B}$ for each. Here, $\underline{B}$ is the $s+1$ by s tridiagonal matrix satisfying

$$AV = \underline{VB},$$

for the unpreconditioned (or split-preconditioned) basis of $s+1$ vectors $\underline{V} = [V, v_{s+1}] = [v_1, v_2, \ldots, v_{s+1}]$. In the case of left preconditioning, $\underline{B}$ satisfies both

$$M^{-1}AV = \underline{VB}$$

and

$$AM^{-1}W = \underline{WB},$$

where $\underline{V} = [V, v_{s+1}]$ comprise the *left-hand basis* and $\underline{W} = [W, w_{s+1}]$ the *right-hand basis*. For details on the preconditioned s-step basis, see Section 2.2. Later, in Section 7.4, we will summarize known results on the condition number of each basis as a function of s.

7.3.1 Monomial basis

The very first s-step Krylov methods used the *monomial basis*

$$\begin{aligned} K_{s+1}(A, v)_{\text{Monomial}} &= [v_1, v_2, \ldots, v_{s+1}] \\ &= [v, Av, A^2v, \ldots, A^sv]. \end{aligned} \tag{7.1}$$

This has a change of basis matrix

$$\underline{B} = [e_2, e_3, \ldots, e_{s+1}], \tag{7.2}$$

where e_j represents the j-th column of the $s+1$ by $s+1$ identity matrix. Sometimes a *scaled monomial basis* was used, where $\sigma_k v_{k+1} = Av_k$ for $k = 1, 2, \ldots, s$ and for positive real scaling factors $\sigma_1, \ldots, \sigma_s$. In that case, the change of basis matrix is given by

$$\underline{B} = [\sigma_1 e_2, \sigma_2 e_3, \ldots, \sigma_s e_{s+1}]. \tag{7.3}$$

Previous work using the monomial basis, either in the original or in the scaled form, includes the s-step CG algorithms devised by Van Rosendale [228] and Chronopoulos and Gear [57], as well as Walker's Householder GMRES [238, 135, 239]. The motivation behind these algorithms was to break the data dependency in traditional Krylov methods between the result of the sparse matrix-vector product, and the result of one or more dot products and other vector operations. Breaking this dependency meant more freedom for humans and compilers to optimize the resulting computational kernels, and thus potentially more performance. As we discuss elsewhere in this thesis, other authors and finally we realized this potential by providing optimized kernels.

The monomial basis was a natural (though not necessary) choice for s-step Krylov methods, as it is the basis one commonly uses when notating a Krylov subspace. Several other reasons suggest it. First, computing the *Gram matrix* G of the monomial basis when A is Hermitian results in a "matrix of moments": $G_{ij} = v^* A^{(i-1)+(j-1)} v$, just as the $(i+j)^{\text{th}}$ moment of the monomial polynomials is $\int_\Omega x^{i+j}\, dx$ (where the domain of definition of the polynomials is some region Ω). This is not necessarily true for other bases. In general,

$$G_{ij} = v^* p_{j-1}(A) p_{i-1}(A) v,$$

and it may not be true that $p_{j-1}(z)p_{i-1}(z) = p_{j+i-2}(z)$. (In fact, this assumes that $p_{j+i-2}(z)$ has been defined, whereas the s-step basis only defines polynomials up to degree s.) Chronopoulos and Gear exploit this property of the monomial basis in their s-step CG algorithm. Computationally, the kernel represented by Equation (7.1) has other applications, such as multiple smoothing steps in multigrid (see [89, 219]), or lookahead in nonsymmetric Lanczos (see Parlett et al. [191]). This helps justify the effort of optimizing such a kernel. Finally, the monomial basis also requires no additional information to compute other than the sparse matrix A and the starting vector v. Other bases require spectral estimates or bounds; this introduces a chicken-and-egg problem, since one generally must compute such estimates with a Krylov subspace method. (We will talk about how that chicken-and-egg problem is solved in Sections 7.3.2 and 7.3.3.)

The cost of the monomial basis' simplicity is numerical stability. In exact arithmetic, any basis of the Krylov subspace in question will do, but finite-precision arithmetic changes this. Chronopoulos and Gear [57] found that for their test problems, basis lengths longer than $s = 5$ prevented their s-step CG from converging. Van Rosendale also noticed numerical stability problems due in part to his use of the monomial basis, but was unsure at the time how to correct them [229]. Hindmarsh and Walker [135] tried scaling the vectors in the monomial basis to avoid overflow and numerical instability in Householder GMRES, though as we will discuss in Section 7.4, this cannot help the latter much. Chronopoulos and his collaborators later developed ad-hoc techniques to circumvent stability problems. For example, they applied iterative refinement when solving linear systems to extract coefficients from the basis [55], or postprocessed the basis vectors by applying modified Gram-Schmidt

orthogonalization [60]. This latter step let Chronopoulos and Swanson increase the usable basis length to $s \leq 16$ for the problems of interest, but at the expense of significant additional computation and bandwidth requirements, that prevent their algorithm from meeting our goal of avoiding communication.

The monomial basis has another name: the *power method*, which is a simple iteration for finding the principal eigenvalue and corresponding eigenvector of a matrix. If the matrix and starting vector satisfy certain conditions, then repeatedly applying the matrix to a suitable starting vector causes the resulting vector to converge to the principal eigenvector of the matrix. The power method is used most famously by Google in their PageRank algorithm (see e.g., [39]), though it occurs in many contexts, such as the adaptive Chebyshev method for solving linear systems [172], and some ordinary differential equation solvers.

Intuitively, converging is not something one wants a basis to do. In exact arithmetic, any basis will do, but in finite-precision arithmetic, basis vectors should be maximally independent, ideally orthogonal. Instead, as the basis length s increases, the successive vectors of the monomial basis tend to get closer and closer together in direction. In informal language, once the matrix of basis vectors is no longer numerically full rank, they can no longer be called a basis. Once this happens, no postprocessing can recover the lost rank information. That has already been lost in the basis computation. No matter how accurately the postprocessing is done, computing the basis itself in finite-precision arithmetic loses that information. Of course, some techniques such as careful orthogonalization, or iterative refinement, or column scaling may help to some extent, depending on the details of the Krylov method.

This problem with the monomial basis reminds one of the analogous difficulty in using the monomial basis for polynomial interpolation. In fact, the two situations are closely related, and will enable more formal definitions of "numerically full rank" and "ill-conditioned." We will discuss this later in Section 7.4. The connection with polynomial interpolation also suggests other bases to try. One of these is the Newton basis, which will be shown below. However, an important point which we will examine later in more depth is that the ill conditioning of the monomial basis depends significantly on the matrix. In fact, there are matrices for which the monomial basis is better conditioned than the Newton basis, and vice versa.

7.3.2 Newton basis

The *Newton basis*

$$\begin{aligned} K_{s+1}(A, v)_{\text{Newton}} &= [v_1, v_2, \ldots, v_{s+1}] \\ &= \left[v, (A - \theta_1 I)v, (A - \theta_2 I)(A - \theta_1)v, \ldots, \prod_{i=1}^{s} (A - \theta_i I)v \right] \end{aligned} \tag{7.4}$$

in s-step Krylov methods corresponds intuitively to the Newton basis $p_0(z) = 1$, $p_1(z) = z - \theta_1$, $p_2(z) = (z-\theta_2)(z-\theta_1)$, $\ldots$, $p_s(z) = (z-\theta_s)\cdots(z-\theta_1)$ for polynomial interpolation at the nodes θ_1, θ_2, $\ldots$, θ_s. In the case of Krylov methods, the θ_i are called *shifts* rather than nodes. This (deliberately) recalls the shifts used in eigenvalue solvers to improve convergence near a specific eigenvalue. The intuition behind the Newton basis comes directly from polynomial interpolation: in many (but not all) cases, the Newton polynomial basis is better conditioned

than the monomial polynomial basis, for the same set of interpolation nodes. We will discuss this connection between the s-step basis and polynomial interpolation further in Section 7.4.

The Newton basis has an $s+1$ by s change of basis matrix

$$\underline{B} = \begin{pmatrix} \theta_1 & 0 & \dots & 0 \\ 1 & \theta_2 & \ddots & \vdots \\ 0 & 1 & \ddots & 0 \\ \vdots & \ddots & \ddots & \theta_s \\ 0 & 0 & \dots & 1 \end{pmatrix}. \tag{7.5}$$

The Newton basis may be scaled just like the monomial basis, so that $\sigma_k v_{k+1} = (A - \theta_k) v_k$ for $k = 1, 2, \dots, s$ and for positive real scaling factors $\sigma_1, \dots, \sigma_s$. In that case, the change of basis matrix is given by

$$\underline{B} = \begin{pmatrix} \theta_1 & 0 & \dots & 0 \\ \sigma_1 & \theta_2 & \ddots & \vdots \\ 0 & \sigma_2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \theta_s \\ 0 & 0 & \dots & \sigma_s \end{pmatrix}. \tag{7.6}$$

Previous work

Bai et al. [18, 19] were the first, as far as we know, to use the Newton basis in an s-step Krylov method.[2] The method was GMRES, and their goal was to avoid communication in the vector-vector operations (though they did not express it in terms of communication). Bai et al. observed that standard GMRES requires many vector-vector operations, which are slow relative to matrix-matrix operations on vector and parallel architectures. Walker's Householder QR version of GMRES [238, 239] replaces those vector-vector operations with a dense QR factorization. However, it uses the monomial basis, which makes the algorithm unstable for all but the smallest basis lengths s. Applying an analogy with polynomial interpolation, Bai et al. replaced this monomial basis with a modified version of the Newton basis. This improved numerical stability.

The Newton GMRES algorithm of Bai et al. allows the use of any QR factorization. This differs from standard GMRES, which requires column-oriented MGS. However, the column-by-column (unblocked) Householder QR factorization performs similarly to column-oriented MGS, because the cost of both is dominated by (the same number of) vector-vector operations. Bai et al. do suggest that one could use other QR factorization algorithms, including the blocked QR factorization of Pothen and Raghavan [198], which is a precursor of QR (even though Pothen and Raghavan's QR factorization does not keep the Q factor, so their algorithm cannot be used in s-step GMRES without modification). Erhel [94] implemented the same version of Newton-basis GMRES as Bai and his coauthors, though her implementation uses a new parallel QR factorization intended to improve performance over standard Householder QR.

[2] Joubert and Carey [144] suggested the Newton basis as a possibility, but they implemented a Chebyshev basis instead (see Section 7.3.3).

Choosing the shifts

All the authors we found chose the shifts $\theta_1, \ldots, \theta_s$ from among the eigenvalues of the upper Hessenberg projection matrix (the *Ritz values*) resulting from some number of steps of the iterative method: for example, (standard) Arnoldi iteration / GMRES for nonsymmetric A, or symmetric Lanczos / CG for SPD A. The Ritz values must be arranged in a particular ordering, either the *Leja ordering* for complex arithmetic or the *modified Leja ordering* for real arithmetic, in order to improve numerical stability. We will explain these ordering schemes in Section 7.3.2.

At least s Ritz values are required for the s shifts. Erhel found that for some problems, it improved numerical stability to compute $2s$ Ritz values (via $2s$ iterations of the standard iterative method), and then use the Leja ordering to sort and pick s of them. The authors treated these s or $2s$ steps of standard Arnoldi as a startup phase: first, perform that number of iterations of the standard method, then restart, and use the Ritz values to do restart cycles of the s-step method (with s steps in each restart cycle). One intuitive reason to choose Ritz values as shifts, is that (any version of) Arnoldi works by implicitly constructing an interpolating polynomial of the characteristic polynomial of the matrix A (assuming that A can be diagonalized) at the Ritz values. That makes the Ritz values natural choices for the interpolation nodes of the Newton basis polynomial. We will discuss more formal explanations when and why the Newton basis is a good choice in Section 7.4.

Avoiding complex arithmetic

Even if the linear system $Ax = b$ and the initial guess x_0 are all real, the current set of Ritz values may have imaginary components, if A is nonsymmetric. Complex-valued shifts will likely result in a complex-valued Newton basis. In exact arithmetic, this will not affect the minimum residual approximate solution (as produced by GMRES and mathematically equivalent algorithms), nor will it affect the minimum A-norm approximate solution (as produced by CG and mathematically equivalent algorithms). This is because these minimization problems always have a unique (real) solution, as long as the Krylov basis vectors do form a basis. Nevertheless, complex arithmetic doubles the storage requirement for all the vectors, along with the bandwidth requirements. It also more than doubles the number of floating-point operations. Due to rounding error, it likely introduces small imaginary components into the approximate solution, which may be nonphysical or unexpected. These imaginary components may be discarded.

The eigenvalues of real matrices that have imaginary components always occur in complex conjugate pairs. This is also true for the matrices' Ritz values. Bai et al. [18, 19] exploit this to avoid complex arithmetic, by arranging the shifts in a particular order and modifying the Newton basis. The order of the shifts does matter for numerical stability, as we will discuss below. However, the authors also arrange the shifts so that complex conjugate pairs occur consecutively, with the element of the pair with positive imaginary part occurring first. Suppose that θ_j, θ_{j+1} constitutes such a pair, with $\theta_{j+1} = \bar{\theta_j}$. Rather than choosing the corresponding basis vectors v_{j+1} and v_{j+2} as $v_{j+1} = (A - \theta_j I)v_j$, and $v_{j+2} = (A - \bar{\theta_j} I)v_{j+1}$, as the usual Newton basis would do, they pick

$$v_{j+1} = (A - \Re(\theta_j) I)v_j \tag{7.7}$$

and

$$v_{j+2} = (A - \Re(\theta_j)I)v_{j+1} + \Im(\theta_j)^2 v_j. \tag{7.8}$$

As a result,

$$\begin{aligned} v_{j+2} &= (A - \Re(\theta_j)I)^2 v_j + \Im(\theta_j)^2 v_j \\ &= A^2 v_j - 2\Re(\theta_j)Av_j + \Re(\theta_j)^2 v_j + \Im(\theta_j)^2 \\ &= A^2 v_j - 2\Re(\theta_j)Av_j + |\theta_j|^2 v_j \\ &= (A - \bar{\theta_j}I)(A - \theta_j)v_j. \end{aligned}$$

It is as if the basis "skips over" the offending complex shifts, since picking $v_{j+1} = (A-\theta_j I)v_j$, and $v_{j+2} = (A - \bar{\theta_j}I)v_{j+1}$ results in the same v_{j+2} (in exact arithmetic).

One way this approach might lose accuracy is if θ_{j-1} is real, $\Re(\theta_{j-1}) = \theta_j$, and θ_j and θ_{j+1} form a complex conjugate pair. Then,

$$\begin{aligned} v_j &= (A - \theta_{j-1}I)v_{j-1}, \\ v_{j+1} &= (A - \theta_{j-1}I)^2 v_{j-1}, \\ v_{j+2} &= (A - \theta_{j-1}I)^3 v_{j-1} + \Im(\theta_j)^2 (A - \theta_{j-1}I)v_{j-1}. \end{aligned}$$

The vectors v_j and v_{j+1} then form two steps of the monomial basis, with a possibly ill-conditioned matrix $A - \theta_{j-1}I$. Situations like this could arise if the Ritz values fit within an ellipse with a long vertical axis and a short horizontal axis. Otherwise, the ordering of the shifts discussed below will tend to maximize the distances between consecutive shifts that are not in the same complex conjugate pair.

It could be that such situations do not make the basis ill-conditioned enough to be of concern. If they are a problem, however, one might consider another choice for v_{j+1} other than $(A - \Re(\theta_j)I)v_j$, when $\theta_{j+1} = \bar{\theta_j}$. It's not clear what to pick; for example, $v_{j+1} = Av_j$ reintroduces the problem with a "zero shift." Resorting to complex arithmetic has its own problems, as discussed above. For this reason, one may wish to use a Chebyshev basis (see Section 7.3.3), which by construction produces real polynomials from a real matrix A.

The Leja and Modified Leja orderings

The order in which the shifts appear in the Newton basis affects numerical stability. For example, the effect of two nearly identical consecutive shifts $\theta_{j+1} \approx \theta_j$ is nearly the same as generating two monomial-basis vectors with a shifted matrix $A - \theta_j I$. The shifts are Ritz values of A, so the shifted matrix $A - \theta_j I$ is probably ill-conditioned. Several nearly identical shifts in a row could thus result in an ill-conditioned basis.

Bai et al. [19] avoid this problem by arranging the shifts according to the *Leja ordering* (see e.g., [202]). The Leja ordering of the shifts $\theta_1, \ldots, \theta_s$ is defined as follows. We define the set K_j for $j = 1, 2, \ldots, s$ as follows:

$$K_j \equiv \{\theta_{j+1}, \theta_{j+2}, \ldots, \theta_s\}.$$

We choose the first shift θ_1 as

$$\theta_1 = \text{argmax}_{z \in K_0} |z|. \tag{7.9}$$

Then, we choose subsequent shifts $\theta_2, \ldots, \theta_s$ using the rule

$$\theta_{j+1} = \text{argmax}_{z \in K_j} \prod_{k=0}^{j} |z - z_k|. \tag{7.10}$$

This sequence is not necessarily unique, but any such sequence that could be generated by such a procedure is called a sequence of *Leja points* for the set K_0. This procedure can be used for any set of points in the complex plane, even an infinite set (which results in an infinite sequence of Leja points). See Reichel [202] and Calvetti and Reichel [46] for details on this definition and on the efficient computation of Leja points for real intervals.

Equation (7.10) suggests the point of the Leja ordering: to maximize a particular measure of the "distance" between the current shift θ_j and the previous shifts $\theta_1, \ldots, \theta_{j-1}$. This makes the resulting Newton basis as "unlike" the monomial basis as possible. The Leja ordering improves the accuracy of operations on polynomials, such as Newton interpolation [202] and evaluating polynomial coefficients from their roots [46]. We will cite more formally stated results on this below. Given the connections between Krylov methods and polynomial interpolation, one should not be surprised that using the Newton s-step basis with the Leja ordering of the shifts can help slow condition number growth of the basis as a function of s.

Three deficiencies with the above definition suggest themselves:

1. If the matrix A is real but nonsymmetric, it may have complex Ritz values (that occur in complex conjugate pairs). The Leja ordering may separate complex conjugate pairs, thus defeating the scheme mentioned above (Equations (7.7) and (7.8)) for preserving real arithmetic.

2. Repeated shifts (i.e., $\theta_j = \theta_k$ for some $j \neq k$) may cause the product to maximize in Equation (7.10) to be zero for all possible choices of the next shift θ_{j+1}, so that there is no clear choice for the next shift. (Ritz values from a Krylov method may be unique in exact arithmetic, but need not be in finite-precision arithmetic.)

3. The product to maximize in Equation (7.10) may underflow (if the shifts are close together) or overflow (if the shifts are far apart) in finite-precision arithmetic. Underflow means that the product is zero in the arithmetic. Overflow makes the product `Inf` (floating-point infinity) for all choices of the next shift, which results in the same problem (not having a clear choice of the next shift).

The first problem can be solved by using the *modified Leja ordering* of Bai et al. [19]. This changes the Leja ordering so that complex conjugate pairs θ_j, $\theta_{j+1} = \overline{\theta}_j$ occur consecutively, arranged so that $\Im(\theta_j) > 0$ and $\Im(\theta_{j+1}) < 0$. As shown above (see Equations (7.7) and (7.8)), if complex conjugate pairs of shifts occur consecutively in this arrangement, and the matrix A and vector v are real, the Newton basis can be computed using only real arithmetic. Equation (7.10) is used to choose the first element θ_j of the complex conjugate pair (with positive imaginary part), and θ_{j+1} is chosen to be $\overline{\theta}_j$. The effects of the modified Leja ordering on numerical stability are unclear in theory, but our experience is that it is not generally harmful.

Reichel [202, Equation 1.5b, p. 341] describes how to solve the second problem of repeated shifts. Suppose that for a set of candidate shifts $K_0 = \{\theta_1, \theta_2, \theta_k\}$, each shift θ_j occurs μ_j times (where μ_j is a positive integer). We choose the first shift as before:

$$\theta_1 = \text{argmax}_{z \in K_0} |z|,$$

but we choose subsequent shifts $\theta_2, \ldots, \theta_k$ using the rule

$$\theta_{j+1} = \text{argmax}_{z \in K_j} \prod_{i=0}^{j} |z - z_i|^{\mu_i}. \tag{7.11}$$

Bai et al. [19] do not use this approach. Instead, they deal with this problem by randomly perturbing all the shifts slightly, and then repeating the computation from scratch. In our implementation of the Newton basis, we use Reichel's approach to handle multiple shifts. We only apply the random perturbation as a last resort, if the technique of Equation (7.11) and the solution to Problem 3 (described below) do not work.

Problem 3 can be solved by carefully scaling the points during the Leja ordering computation. Reichel [202] defines a quantity called the *capacity* of a subset K of the complex plane. We give the full definition of the capacity in Appendix D.2. What matters is that for the finite set $K_0 = \{\theta_1, \theta_2, \ldots, \theta_s\} \subset \mathbb{C}$, Reichel explains [202, Section 3, p. 340] that its capacity c can be estimated by the quantity

$$c_k \equiv \prod_{j=1}^{k-1} |\theta_k - \theta_j|^{1/k}. \tag{7.12}$$

Thus, when using Equation (7.11) to compute the Leja ordering, one should divide $\theta_1, \ldots, \theta_k$ by the capacity estimate c_k before computing the ordering. Reichel suggests computing the capacity estimate once. We instead update it by computing c_{k+1} after every time we use Equation (7.11) to compute θ_{k+1}, for each subsequent shift. Reichel does not explain how to extend this capacity estimate to the case of multiple shifts, but the formula is obvious:

$$c_k \equiv \prod_{j=1}^{k-1} |z_k - z_j|^{\mu_j/k}. \tag{7.13}$$

Bai et al. do not describe how to compute the capacity estimate for the shifts, but they do reference Reichel's work.

Algorithm 40 shows how to compute the modified Leja ordering. It contains the methods we use to solve Problems 1–3 above. In the form shown, it requires $\Theta(s^3)$ operations, but computing the Ritz values from an $s \times s$ upper Hessenberg matrix requires that many operations anyway, so the number of operations is not of concern.

7.3.3 Chebyshev basis

Definition

The *Chebyshev basis*

$$K_{s+1}(A, v) = [\tilde{T}_0(A)v, \tilde{T}_1(A)v, \ldots, \tilde{T}_s(A)v]$$

Algorithm 40 Modified Leja ordering

Input: n unique shifts $z_1, \ldots, z_n$, ordered so that any complex shifts only occur consecutively in complex conjugate pairs $z_k, z_{k+1} = \overline{z_k}$, with $\Im(z_k) > 0$ and $\Im(z_{k+1}) < 0$

Input: Each shift z_k has multiplicity μ_k (a positive integer)

Output: n unique shifts $\theta_1, \ldots, \theta_n$, which are the input shifts arranged in the modified Leja order

Output: `outList`: array of indices such that for $k = 1, \ldots, n$, $\theta_k = z_{\texttt{outList}(k)}$

Output: C: estimate of the capacity of $\{z_1, \ldots, z_n\}$

$C \leftarrow 1$ ▷ Initial capacity estimate
Let k be the least index j maximizing $|z_j|$
$\theta_1 \leftarrow z_k$, and `outList` $\leftarrow [k]$
if $\Im(z_k) \neq 0$ **then**
    If $\Im(z_k) < 0$ or $k = n$, error: Input out of order
    $\theta_2 \leftarrow z_{k+1}$, and `outList` $\leftarrow [\texttt{outList}, k+1]$
    $L \leftarrow 2$
else
    $L \leftarrow 1$
end if

while $L < n$ **do**
    $C' \leftarrow C$, and $C \leftarrow \prod_{j=1}^{L-1} |\theta_L - \theta_j|^{\mu_{\texttt{outList}(j)}/L}$ ▷ Update capacity estimate. `outList`(j) is the multiplicity of θ_j.
    for $j = 1, \ldots, n$ **do** ▷ Rescale all the shifts
        $z_j \leftarrow z_j/(C/C')$, and $\theta_j \leftarrow \theta_j/(C/C')$
    end for
    Let k be the least index k in $\{1, \ldots, n\} \setminus$ `outList` maximizing $\prod_{j=1}^{L-1} |z_k - \theta_j|^{\mu_{\texttt{outList}(k)}}$
    If the above product overflowed, signal an error.
    If the above product underflowed, randomly perturb $z_1, \ldots, z_n$ and start over from scratch. Keep track of the number of times this happens, and perturb by a larger amount each time. Signal an error if the number of restarts exceeds a predetermined bound.
    $\theta_{L+1} \leftarrow z_k$, and `outList` $\leftarrow [\texttt{outList}, k]$
    if $\Im(z_k) \neq 0$ **then**
        If $\Im(z_k) < 0$ or $k = n$, error: Input out of order
        $\theta_{L+2} \leftarrow z_{k+1}$, and `outList` $\leftarrow [\texttt{outList}, k+1]$
        $L \leftarrow L + 2$
    else
        $L \leftarrow L + 1$
    end if
end while

uses scaled and shifted *Chebyshev polynomials*.[3] The (unscaled, unshifted) Chebyshev polynomials are a sequence of polynomials $T_0(x)$, $T_1(x)$, ..., which satisfy the recurrence

$$\begin{aligned} T_0(x) &= 1, \\ T_1(x) &= x, \\ T_{k+1}(x) &= 2xT_k(x) - T_{k-1}(x) \text{ for } k > 0. \end{aligned} \tag{7.14}$$

If we restrict $x \in [-1, 1]$, these polynomials satisfy

$$T_k(x) = \cos(n \cos^{-1} x)$$

for all $k = 0, 1, \ldots$. The region of definition may be extended continuously to include complex numbers, if we write

$$T_k(z) = \cosh(k \cosh^{-1} z).$$

For the Krylov basis, the Chebyshev polynomials are scaled and shifted:

$$\tilde{T}_k(\lambda) = \frac{T_k(\frac{d-\lambda}{c})}{T_k(\frac{d}{c})} \tag{7.15}$$

where d and c may in general be complex, and $\tilde{T}_k(0) = 1$ for all k. If we define $\sigma_k = T_k(d/c)$ for $k = 1, 2, \ldots, s$, and set $\sigma_0 = 1$, then we obtain the following recurrence for the scaled and shifted Chebyshev polynomials:

$$\begin{aligned} \tilde{T}_0(\lambda) &= 1, \\ \tilde{T}_1(\lambda) &= \frac{1}{\sigma_1}\lambda, \\ \tilde{T}_{k+1}(\lambda) &= \frac{2\sigma_k}{\sigma_{k+1}}\lambda T_k(\lambda) - \frac{\sigma_{k-1}}{\sigma_{k+1}}T_{k-1}(\lambda) \text{ for } k > 0. \end{aligned} \tag{7.16}$$

The choice of d and c determines a family of ellipses $F(d, c)$ in the complex plane, with foci at $d \pm c$. One chooses the ellipse to be a good fit to the eigenvalues of the matrix A, for reasons we will explain below.

If we define $\sigma_k = T_k(d/c)$ for $k = 1, 2, \ldots, s$, and set $\sigma_0 = 1$, then the $s+1$ by s Chebyshev change of basis matrix is given by Equation (7.17):

$$\underline{B} = \begin{pmatrix} 0 & \frac{\sigma_0}{\sigma_1} & 0 & \ldots & 0 \\ \sigma_1 & 0 & \frac{\sigma_1}{\sigma_2} & \ddots & \vdots \\ 0 & \frac{\sigma_2}{2\sigma_1} & 0 & \ddots & 0 \\ \vdots & 0 & \frac{\sigma_3}{2\sigma_2} & \ddots & \frac{\sigma_{s-2}}{\sigma_{s-1}} \\ & & \ddots & \ddots & 0 \\ 0 & \ldots & & & \frac{\sigma_s}{2\sigma_{s-1}} \end{pmatrix}, \tag{7.17}$$

which comes from rearranging the three-term recurrence in Equation (7.16) to obtain

$$\lambda\tilde{T}_k(\lambda) = \frac{\sigma_{k+1}}{2\sigma_k}\tilde{T}_{k+1}(\lambda) + \frac{\sigma_{k-1}}{\sigma_k}\tilde{T}_{k-1}(\lambda).$$

[3] More specifically, it uses "Chebyshev polynomials of the first kind." There are also "Chebyshev polynomials of the second kind," which we will not reference further in this work.

Properties

Chebyshev polynomials have a unique "minimax" property. Over all (appropriately scaled and shifted) real polynomials of a specified degree on a specified real interval, the Chebyshev polynomial of that degree *mini*mizes the *max*imum absolute value on the interval. (Hence, the term "minimax.") When their region of definition is extended to ellipses in the complex plane, the Chebyshev polynomials still satisfy the minimax property asymptotically, though not (always) exactly (see e.g., [96]). The minimax property makes Chebyshev polynomials both theoretically and practically useful for Krylov subspace methods.[4] On the one hand, their minimax property makes them a powerful theoretical tool for bounding the convergence rate of iterations such as CG and GMRES (see e.g., Greenbaum [120]), which promise optimal reduction of an error or residual norm in each iteration. This error minimization phenomenon also inspired the use of Chebyshev polynomials in practical iterative methods. Direct use of these polynomials in a Krylov subspace method for solving linear systems, along with some technique for estimating a region containing the spectrum of the matrix, is called *Chebyshev iteration* (see e.g., Manteuffel [172]). (We discuss this iteration for solving linear systems in more detail in Section 1.6.3.) Chebyshev polynomials of matrices are also used in polynomial preconditioners, both for their error minimization property and because they can be computed without expensive inner products (see e.g., [207]). Finally, some versions of the "lookahead" process in nonsymmetric Lanczos use a basis of Chebyshev polynomials to advance the iteration past a temporary breakdown point (see e.g., [104]). Here the minimax property helps keep the basis vectors well-conditioned, without needing to orthogonalize each basis vector after creating it. We have the same goal in mind as in lookahead when choosing the Chebyshev basis.

Prior work

De Sturler [71] suggested using Chebyshev polynomials in an s-step method, but did not implement the idea at that time. He was inspired by their "minimax" property. Joubert and Carey [144, 145] developed a similar Chebyshev-basis s-step GMRES algorithm and implementation. Van der Vorst and de Sturler [73] later implemented an s-step version of GMRES using the Chebyshev basis. Even though they used neither an optimized matrix powers kernel nor an optimized QR factorization (such as TSQR, see Section 2.3), just rearranging GMRES in this way achieved a speedup of $1.81\times$ over standard GMRES on 196 processors of a 400-processor Parsytec Supercluster, even though that machine was known for having fast communication and slow computation relative to other parallel computers of its day.

Real and complex arithmetic

Performance reasons require avoiding complex arithmetic if the original matrix A (and right-hand side b, if solving $Ax = b$) is real. We have already discussed how to avoid complex arithmetic with the Newton basis in Section 7.3.2, when any complex shifts that occur do

[4] Koch and Liesen [157] concisely state this dual utility of Chebyshev polynomials in the context of Krylov subspace methods.

so in complex conjugate pairs (as they must if they are Ritz values of a real matrix A with a real starting vector). Similarly, we can avoid complex arithmetic with the Chebyshev basis. In the change of basis matrix $\underline{B}$ given by Equation (7.17), the scaling factors σ_k are given by

$$\sigma_k = T_k(d/c) = \cosh(k \cosh^{-1}(d/c)),$$

where d and c determines a family of ellipses $F(d, c)$ in the complex plane, with foci at $d \pm c$. These ellipses are chosen to be a good fit to the spectrum of the matrix A. If A is real, then d (the center of the ellipse) must be on the real axis, and c must be either real or pure imaginary. If d and c are both real, then σ_k is always real for all k, and the change of basis matrix is real. However, if c is imaginary, $\sigma_1 = T_1(d/c) = d/c$ will certainly be imaginary. Also, it is not hard to show that if z is pure imaginary, then $T_k(z)$ is pure imaginary if k is odd, and real if k is even. That means σ_k/σ_{k+1} and σ_{k+1}/σ_k must be pure imaginary for $k \geq 1$, if d/c is pure imaginary.

If d/c is pure imaginary, then the change of basis matrix $\underline{B}$ will also be pure imaginary: $\underline{B} = i\underline{B}'$, where $\underline{B}'$ is real. However, recall that the Chebyshev basis is shifted: for the (real) matrix A and real vector v,

$$\tilde{T}_k(A)v = T_k\left(\frac{1}{c}(dI - A)\right) / T_k(d/c).$$

Recall that if d is real and c is pure imaginary, then $T_k(d/c)$ must also be pure imaginary. Also, if c is pure imaginary and d and (the matrix) A are real, then $T_k\left(\frac{1}{c}(dI - A)\right)$ must also be pure imaginary. That means $\tilde{T}_k(A)v$ is real. In that case, we can replace c with c/i without changing the basis. This amounts to rotating the ellipse 90 degrees clockwise, so that its foci are on the real axis. It makes the change of basis matrix $\underline{B}$ real, without actually changing the basis.

Estimating ellipse parameters

Just like with the Newton basis, the Chebyshev basis requires some estimated information about the spectrum of A. Specifically, it requires a bounding ellipse. In the case of Chebyshev iteration, there are many adaptive schemes for estimating the center d and foci $d \pm c$ of the ellipse, especially when the matrix A is real. Elman et al. [92] show how to use the results of a few steps of Arnoldi iteration to estimate the d and c parameters. We may do the same, but instead of running Chebyshev iteration once we have d and c, we may run our s-step Arnoldi (or Lanczos in the case of symmetric A) algorithm.

Performance notes

The Chebyshev basis uses a three-term recurrence. This increases the computational requirements (the number of floating-point operations). It may also increase the amount of storage in fast memory, depending on how the matrix powers kernel is implemented (see Section 2.1, in particular the "implicit" SA3 algorithm). Note, however, that the the modified Newton basis, if there are complex conjugate pairs of shifts, requires the same amount of fast memory storage as the Chebyshev basis, and at worst an amount of computation intermediate between that of the Newton basis with all real shifts, and the Chebyshev basis.

Our experience in our parallel shared-memory implementation of CA-GMRES (see Section 3.6) is that the runtime of the Newton basis is nearly the same as the runtime of the monomial basis. Whether the Chebyshev basis' additional term in the vector recurrence matters depends on whether the matrix powers kernel performance is still bound by memory bandwidth. That additional term only affects the sequential part of the matrix powers kernel; it does not add to the number or size of the messages in parallel.

7.4 Basis condition number

Any s-step basis must be linearly independent in exact arithmetic, as long as the Krylov subspace can be extended. However, the basis vectors tend to lose their independence as the basis length s increases. At some point, the vectors become no longer full rank in finite-precision arithmetic. That may have many ill effects on the convergence of the iterative method. First, the s basis vectors may not be able to increase the dimension of the Krylov subspace by s. This will make the s-step method no longer mathematically equivalent to the conventional Krylov iteration from which it was derived, and could (at best) slow convergence. Second, depending on how the solution update at that outer iteration is computed, it may be inaccurate or even undefined. Finally, the inaccurate solution update may cause the Krylov method to diverge. Fixing divergence requires either backtracking to some previous outer iteration, or restarting entirely from some known good starting vector. Either way loses time solving a problem that a conventional Krylov iteration would not have had. Thus it is important to understand the factors that can cause loss of linear independence in the s-step basis vectors.

We begin with Section 7.4.1, where we define more formally what we mean by the "condition number" of the basis. That is, we state the mapping whose sensitivity to relative perturbations the condition number (which is a kind of derivative) measures. This will lead into Section 7.4.3, where we summarize results relating the s-step Krylov basis to polynomial interpolation on the eigenvalues of the matrix A. The condition numbers of these processes are also related. In Section 7.4.4, we cite known results concerning the condition number growth of the monomial, Newton, and Chebyshev bases. Finally, in Section 7.4.2, we talk about the influence of orthogonalizing each group of s (or $s + 1$, depending on the algorithm) basis vectors against the previously orthogonalized basis vectors, in our communication-avoiding versions of Lanczos or Arnoldi iteration.

7.4.1 Condition number

Many authors speak of the "condition number" of the s-step basis. The condition number measures the relative sensitivity of some computation's output to the input, but most of these authors do not define that computation explicitly. Defining it is important because different computations, and different ways of computing them, have different condition numbers. For example, suppose that A is a dense 10×10 Jordan block with eigenvalues all 1. If I compute a dense 10×10 random orthogonal matrix Q, then the condition number of QAQ^* with respect to solving linear systems is about 13. However, the condition numbers of each of its eigenvalues are all greater than 1.6×10^{13}. The first authors we found to make explicit what

they mean by the "condition number" of the s-step basis were Joubert and Carey [144], and the content of this section was inspired by them.

In this section, we speak of the condition number of the s-step basis, when we are using that basis in a Krylov method to solve linear systems. As the above Jordan block example illustrates, eigenvalue problems have their own entirely different condition numbers. The analysis of Joubert and Carey covers the $t = 1$ case of our CA-GMRES algorithm (Section 3.4). Here, we extend the analysis to the general $t \geq 1$ case. The analysis can also be extended to our CA-CG algorithm (Section 5.4), in a sense we will explain below. This section describes work in progress, and we will identify the steps which need completion.

GMRES

Each iteration of standard MGS-based GMRES requires solving a linear least-squares problem to find the solution update at that iteration. The least-squares problem uniquely determines the solution update (in exact arithmetic). It is the condition number of this problem which matters for the accuracy of the approximate solution to the linear system $Ax = b$. If we write $Q^{(j)}$ for the matrix of j basis vectors produced by j iterations of standard GMRES, x_0 for the initial guess, and $r_0 = b - Ax_0$ for the initial residual, the least-squares problem is

$$y_j := \text{argmin}_y \|r_0 - AQ^{(j)}y\|_2 \tag{7.18}$$

and the approximate solution x_j is

$$x_j := x_0 + Q^{(j)}y_j.$$

Of course, GMRES does not solve Equation (7.18) explicitly. However, all algorithms equivalent to standard GMRES in exact arithmetic solve this least-squares problem in some way, so determining the sensitivity of this problem to perturbations is generally applicable. The accuracy of y_j and therefore of x_j depends on various factors, here especially on the 2-norm condition number of the matrix $AQ^{(j)}$. We can look at the least-squares problem in two different ways:

1. As a monolithic problem, which gives the expected accuracy of the solution x_j, considering the entire basis $Q^{(j)}$ up to the current iteration j this point; or

2. As a QR factorization update, in which we assume that the columns of $Q^{(j-1)}$ are perfectly orthonormal resp. unitary, and only consider the accuracy of updating the basis (to get $Q^{(j)}$) by adding a new column.

Our CA-GMRES (Section 3.4) algorithm solves the same least-squares problem (in exact arithmetic) as does standard GMRES, but computes both the basis vectors and the solution in a different way (that matters for finite-precision arithmetic). Also, CA-GMRES only produces an approximate solution $x_{k+1} = x_0 + \Omega_k y_k$ once every outer iteration k. If we prefer the monolithic approach, we get

$$y_k := \text{argmin}_y \|r_0 - A[V_1, \ldots, V_k]y_k\|_2,$$

which suggests that the condition number of $A[V_1, \ldots, V_k]$ is of interest. If we choose to look at the least-squares problem as an update, we get

$$y_k := \text{argmin}_y \|r_0 - A[\mathfrak{Q}_{k-1}, V_k]y_k\|_2,$$

which suggests that the condition number of AV_k, or of $A[\mathfrak{Q}_{k-1}, V_k]$, is of interest.

CG

CG is equivalent in exact arithmetic to the Full Orthogonalization Method (FOM) for solving linear systems (see e.g., Saad [208]). FOM computes the same basis vectors and upper Hessenberg matrix as GMRES, except that FOM's approximate solution comes from solving a least-squares problem in the A norm:

$$y_k := \text{argmin}_y \|e_0 - [V_1, \ldots, V_k]y\|_A.$$

We can express this as a 2-norm least squares problem:

$$y_k := \text{argmin}_y \|A^{1/2}e_0 - A^{1/2}[V_1, \ldots, V_k]y\|_2. \tag{7.19}$$

Thus the condition number that matters is that of $A^{1/2}[V_1, \ldots, V_k]$. The matrix square root exists because we assume that A is positive definite.

Eigensolvers

We reserve discussion of the condition number of the basis with respect to various eigenvalue problems for future work.

7.4.2 The projection step

CA-GMRES (Section 3.4), CA-Arnoldi (Section 3.3), and CA-Lanczos (Section 4.2) all orthogonalize the current set of basis vectors $\underline{\acute{V}}_k$ against the previous basis vectors $\underline{\mathfrak{Q}}_{k-1}$, before making the updated s-step basis orthogonal with a QR factorization.[5] We call this the *projection step* of outer iteration k, because it involves projecting $\underline{V}_k$ orthogonally onto $\mathcal{R}\left(\underline{\mathfrak{Q}}_{k-1}\right)^{\perp}$, which is the orthogonal complement of the range of $\underline{\mathfrak{Q}}_{k-1}$. The basis condition number analysis of Joubert and Carey [144] does not include the projection step, because their algorithm is equivalent in exact arithmetic to Arnoldi(s, 1). We leave a full analysis of the influence of the projection step to future work, but we discuss a few possible directions of that analysis here.

In this section, we only consider CA-Arnoldi for simplicity. Let $\underline{\acute{V}}_k$ be the s-step basis generated at outer iteration k by the matrix powers kernel, and let $\underline{\acute{V}}_k'$ be the result of projecting $\underline{\acute{V}}_k$ orthogonally onto $\mathcal{R}\left(\underline{\mathfrak{Q}}_{k-1}\right)^{\perp}$. This is done by either the MGS or CGS variants

[5] The left-preconditioned algorithms derived from CA-Lanczos orthogonalize with respect to a different inner product, so they cannot use a QR factorization, but otherwise the process is similar. Also, depending on how the algorithms are implemented, $\underline{\mathfrak{Q}}_{k-1}$ may be replaced with $\mathfrak{Q}_{k-1}$ and $\underline{\acute{V}}_k$ may be replaced with $\underline{V}_k$. The details are not so important here.

of Block Gram-Schmidt, which we describe in Section 2.4. There are many variations on Block Gram-Schmidt and also many reorthogonalization options; we gloss over these here and try to address the issues that are independent of the choice of Block Gram-Schmidt scheme.

We discuss a general Block Gram-Schmidt reorthogonalization scheme in Section 2.4.8. That can be applied to any collection of vectors, even if they are dependent in exact arithmetic. Our vectors specifically are the Krylov basis vectors in a Krylov subspace method. This means both that the "Twice is Enough" observation of Kahan applies (cited in Parlett [189, Section 6.9]), and that detecting loss of linear independence is important. Furthermore, the scheme in Section 2.4.8 performs more computations and requires more communication than we would like. Most of the communication comes from computing the norms of columns both of $\underline{\acute{V}}_k$ (before the projection step) and $\underline{\acute{V}}_k'$ (after the projection step). Computing column norms defeats the communication-avoiding feature of the combination of TSQR and block Gram-Schmidt.

In our iterative methods, $\underline{\acute{V}}_k'$ may be numerically rank-deficient for two reasons:

1. Lucky breakdown of the iterative method
2. The s-step basis is poorly conditioned

It is very important to distinguish the two causes, as lucky breakdown means the Krylov subspace cannot be extended further and the iterative method should stop. A poorly conditioned s-step basis, in contrast, calls for adjusting the basis (by reducing the basis length s and/or improving the basis polynomials) and repeating the affected outer iteration.

The numerical rank-deficiency of an s-step basis in one outer iteration of CA-Arnoldi does not depend on the numerical rank deficiency of previous or future outer iterations; it only depends on the matrix A, the starting vector of that outer iteration, the basis length s, and the particular basis used. This means we can use a dynamic approach to distinguish the two causes of numerical rank deficiency in the s-step basis. If efforts to improve the conditioning of the s-step basis do not succeed, then we can revert to a standard Krylov subspace method from that point. This will detect a lucky breakdown accurately, albeit expensively. Lucky breakdowns are rare, though, so we do not expect to pay this price often.

Standard Arnoldi computes and orthogonalizes each basis vector one at a time. The above phenomenon could still occur, in the sense that the norm of a new basis vector Av_j might drop significantly after one round of orthogonalization. This is often associated with convergence of a particular Ritz value (in case reorthogonalization is desirable), although it may also signal lucky breakdown. However, in an s-step method like Arnoldi(s, t), the s-step basis complicates the analysis. For example, if the monomial basis is used and if the power method applied to A and q_{sk+1} would converge quickly, then the basis vectors may have a large q_{sk+1} component relative to their $\mathcal{R}\left(\underline{\mathfrak{Q}}_{k-1}\right)^{\perp}$ component. If the power method applied to A and q_{sk+1} would not converge quickly, then the basis vectors should have a significant $\mathcal{R}\left(\underline{\mathfrak{Q}}_{k-1}\right)^{\perp}$ component. This could explain the phenomenon we observed in experiments, that the monomial basis works poorly for symmetric matrices (on which the power method converges nicely) as compared with other bases, but better for highly nonnormal matrices (on which the power method may take many iterations to converge, or may not converge at all).

From the above argument, it seems that loss of linear independence in the columns of $\underline{\acute{V}}_k$, and loss of accuracy in the projection step, are correlated. However, we will have to do more experiments and analysis to confirm this.

7.4.3 Basis as polynomial interpolation

In this section, we illustrate the connections between the s-step Krylov basis, and polynomial interpolation of the eigenvalues of the matrix A. Much of the content in this section comes from Beckermann [25, 26].

We first introduce notation. We call the matrix

$$K_{s+1}(A, v) = [p_0(A)v, p_1(A)v, \ldots, p_s(A)v] \tag{7.20}$$

corresponding to a particular choice of s-step basis polynomials $p_0(z)$, $p_1(z)$, ..., $p_s(z)$ a *Krylov matrix*. Depending on the basis type, this may be either a *Krylov-monomial*, *Krylov-Chebyshev*, or *Krylov-Newton* matrix. (Some authors call this a "Krylov-like" matrix if the basis is not monomial.) The choice of polynomials is implicit in the above notation and understood from context. The matrix powers kernel at outer iteration k of CA-GMRES (Section 3.4), CA-Arnoldi (Section 3.3), or CA-Lanczos (Section 4.2) generates a Krylov matrix

$$\underline{V}_k = K_{s+1}(A, q_{sk+1}) = [p_0(A)q_{sk+1}, p_1(A)q_{sk+1}, \ldots, p_s(A)q_{sk+1}].$$

Suppose that the $n \times n$ matrix A has an eigenvalue decomposition $A = U\Lambda U^{-1}$, where the matrix Λ is the diagonal matrix of eigenvalues. Here, we do not assume that the matrix of eigenvectors U is orthogonal resp. unitary. Then the Krylov matrix $K_{s+1}(A, v)$ satisfies

$$K_{s+1}(A, v) = U[p_0(\Lambda)U^{-1}v, p_1(\Lambda)U^{-1}v, \ldots, p_s(\Lambda)U^{-1}v].$$

If we let $[z_1, \ldots, z_n]^T = U^{-1}v$, then

$$\begin{aligned} K_{s+1}(A, v) &= U \begin{pmatrix} p_0(\lambda_1)z_1 & p_1(\lambda_1)z_1 & \ldots & p_s(\lambda_1)z_1 \\ p_0(\lambda_2)z_2 & p_1(\lambda_2)z_2 & \ldots & p_s(\lambda_2)z_2 \\ \vdots & \vdots & \ddots & \vdots \\ p_0(\lambda_n)z_n & p_1(\lambda_n)z_n & \ldots & p_s(\lambda_n)z_n \end{pmatrix} \\ &= Ue^{i\Phi} \operatorname{diag}(e^{-i\phi_1}z_1, \ldots, e^{-i\phi_n}z_n) \begin{pmatrix} p_0(\lambda_1) & p_1(\lambda_1) & \ldots & p_s(\lambda_1) \\ p_0(\lambda_2) & p_1(\lambda_2) & \ldots & p_s(\lambda_2) \\ \vdots & \vdots & \ddots & \vdots \\ p_0(\lambda_n) & p_1(\lambda_n) & \ldots & p_s(\lambda_n) \end{pmatrix}, \end{aligned} \tag{7.21}$$

where we define the diagonal matrix

$$e^{i\Phi} \equiv \operatorname{diag}(e^{i\phi_1}, \ldots, e^{i\phi_n}). \tag{7.22}$$

In Equation (7.21), we chose each ϕ_j so that $e^{-i\phi_j}z_j$ is nonnegative and real.

We call matrices of the form

$$P_{s+1}(\lambda_1, \ldots, \lambda_n) = \begin{pmatrix} p_0(\lambda_1) & p_1(\lambda_1) & \ldots & p_s(\lambda_1) \\ p_0(\lambda_2) & p_1(\lambda_2) & \ldots & p_s(\lambda_2) \\ \vdots & \vdots & \ddots & \vdots \\ p_0(\lambda_n) & p_1(\lambda_n) & \ldots & p_s(\lambda_n) \end{pmatrix} \tag{7.23}$$

Vandermonde matrices. Most authors restrict the term "Vandermonde matrix" to the monomial basis $p_j(z) = z^j$, and call matrices in the form of Equation (7.23) "Vandermonde-like" if they correspond to a different basis (see e.g., Gautschi [108]). As with Krylov matrices, we say "Vandermonde-monomial matrix" for the monomial basis, "Vandermonde-Newton matrix" for the Newton basis, and so on for other bases.

Recall the definition of the Krylov basis condition number for GMRES (Equation (7.18)) and for CG (Equation (7.19)). Equation (7.21) suggests that two important factors in the basis condition number are the condition number of the eigenvectors of A (so, $\kappa_2(U^{-1}U)$), which is 1 when A is normal, and matrices of the form

$$\tilde{P}_{s+1} = \text{diag}(e^{-i\phi_1} z_1, \ldots, e^{-i\phi_n} z_n) P_{s+1}(\lambda_1, \ldots, \lambda_n). \tag{7.24}$$

Matrices of the form of $\tilde{P}_{s+1}$ are called *row-scaled Vandermonde matrices*. Beckermann uses row-scaled Vandermonde matrices in his analysis. A row-scaled Vandermonde matrix arises when doing polynomial interpolation in which the different observations (corresponding to different rows of the Vandermonde-like matrix and their right-hand side elements) have different weights. For example, one may trust certain observations more than others. (Beckermann requires that those weights be positive, which is why we include the additional diagonal matrix of $e^{-i\phi_j}$ terms.)

We see from the above that using the Krylov basis to compute the solution update in an iterative method is like doing weighted (i.e., row-scaled) polynomial interpolation. The interpolation nodes are the eigenvalues of the matrix A, the values to interpolate are the entries of the residual vector (for GMRES) or the error vector (for CG), and the weights are the absolute values of the components of that vector in the basis of eigenvectors of the matrix A. We made simplifications in the above arguments, but this decomposition can be generalized, for example when the matrix has repeated eigenvalues or is defective.

Before we continue the analysis, we first make some observations about the decomposition in Equation (7.21). First, if A is normal, then U is unitary and thus we can ignore the U factor in analyzing the condition number of $K_s(A; q_{sk+1})$. Second, the row scaling does affect the condition number, although measuring the error in a different norm removes the influence of the row scaling completely, as long as none of the z_j are zero. In this case, the norm is determined by the components of the starting vector in the eigenbasis of A. We do not know these components, but it may not be necessary if we can bound the condition number independently of the row scaling. Third, while we do not have U or $U^{-1}b$, we do have approximations of the eigenvectors of A, since we are performing the Arnoldi process.

7.4.4 Condition number bounds

Monomial basis

Theory confirms the experimental observation that the condition number of the monomial basis grows exponentially with the basis length s, when the eigenvalues of the matrix are real. The situation may be quite different for matrices with complex eigenvalues; in fact, we will show an example with condition number one. Furthermore, unpredictable behavior may result if the matrix A is nonnormal; the basis may be better conditioned than one would expect from the eigenvalues of A. However, the case of real eigenvalues covers all symmetric and Hermitian matrices, among others.

The choice of basis for the Krylov subspace corresponds intuitively to polynomial interpolation. When using the monomial basis of polynomials 1, z, z^2, ..., z^s to interpolate the nodes z_1, ..., z_n, an n by $s+1$ *Vandermonde matrix* results:

$$P_{s+1}(z_1, \ldots, z_n) := \begin{pmatrix} 1 & z_1 & z_1^2 & \ldots & z_1^s \\ 1 & z_2 & z_2^2 & \ldots & z_2^s \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & z_n & z_n^2 & \ldots & z_n^s \end{pmatrix}$$

Many authors have observed that for real nodes z_1, ..., z_n, Vandermonde matrices have a large condition number with respect to the least-squares polynomial interpolation problem

$$\min_{y \in \mathbb{R}^{s+1}} \|b - P_{s+1}(z_1, \ldots, z_n)y\|_2.$$

For example, Gautschi [110] showed that for any real nodes z_1, ..., z_n, the 2-norm condition number of $P_{s+1}(z_1, \ldots, z_n)$ grows no slower than exponentially in s. Note that this may not hold if the nodes are not all real. For example, if ω_s is the s^{th} complex root of unity, then the Vandermonde matrix $V_{s-1}(\omega_s^0 = 1, \omega_s^1, \omega_s^2, \ldots, \omega_s^{s-1})$ has orthogonal columns, all of length s, and $\kappa_2(V_{s-1}) = 1$. (This matrix is just a multiple of the discrete Fourier transform matrix.) Nevertheless, the case of real nodes leads us to consider the large class of matrices with real roots.

This connection between Vandermonde matrices and the monomial basis can be formalized. Beckermann [25, 26] does so by applying the theory of orthogonal polynomials to relate the matrix of monomial basis vectors to Vandermonde and Hankel matrices. He uses this to prove that the condition number of the monomial basis matrix grows no slower than exponentially in the number of basis vectors, when the matrix A is diagonalizable and has real eigenvalues.

Beckermann quantifies the intuitive observation above, that the condition numbers of the following matrices are related:

1. Krylov matrices $K_{s+1}(A, v) = [v, Av, A^2v, \ldots, A^s]$
2. Square Vandermonde matrices
3. Row-scaled Vandermonde matrices $DP_{s+1}(z_1, \ldots, z_n)$
4. Matrices of moments $H(\mu)_{ij} = \int_{\text{supp}(\mu)} x^{i+j}$

5. *Hankel matrices* (square matrices with constant antidiagonals)

The condition numbers of Hankel matrices tend to grow rapidly with their dimension. The best-known example is the *Hilbert matrix*

$$H_{ij} = \frac{1}{i+j-1},$$

though there are many other examples. Beckermann observes that for any real positive semidefinite Hankel matrix H, there exists some positive (possibly discrete) Borel measure μ with real support $\text{supp}(\mu) \subset \mathbb{R}$ such that

$$H_{ij} = \int_{\text{supp}(\mu)} x^{(i-1)+(j-1)}\, d\mu(x). \tag{7.25}$$

That means any real positive semidefinite Hankel matrix is also a matrix of moments. Furthermore, given any Vandermonde matrix $P = P_{s+1}(z_1, \ldots, z_n)$, the matrix P^*P is an $s+1$ by $s+1$ Hankel matrix.

Beckermann continues to define some notation for lower bounds. Given a real interval $I \subset \mathbb{R}$, he defines $\Gamma_s(I)^2$ as the (square of the) lower bound of the 2-norm condition number of all $s+1$ by $s+1$ Hankel matrices H_{s+1} (defined by Equation (7.25)):

$$\Gamma_s(I)^2 \equiv \inf\{\kappa_2(H_{s+1}(\mu)) : \text{supp}(\mu) \subset I\}. \tag{7.26}$$

Similarly, for all real Vandermonde matrices with $s+1$ columns and the same number of nodes $z_1, \ldots, z_{s+1}$:

$$\Gamma_s^P(I) \equiv \inf\{\kappa_2(P_{s+1}(z_1, \ldots, z_{s+1})) : z_1, \ldots, z_{s+1} \in I\}. \tag{7.27}$$

For all row-scaled Vandermonde matrices $DP_{s+1}(z_1, \ldots, z_n)$ with $s+1$ columns and n nodes:

$$\Gamma_s^D(I) \equiv \inf\{\kappa_2(D \cdot P_{s+1}(z_1, \ldots, z_n)) : z_1, \ldots, z_n \in I, D \text{ diagonal.}\} \tag{7.28}$$

Finally, for all n by $s+1$ Krylov matrices $K_{s+1}(A, v)$ with Hermitian A and $n > s+1$:

$$\Gamma_s^K(I) \equiv \inf\{\kappa_2(K_{s+1}(A, v) : \sigma(A) \subset I\}. \tag{7.29}$$

Beckermann [26, Theorem 2.1] proves that for each interval $I \subset \mathbb{R}$ and for all $s > 0$,

$$\Gamma_s(\mathbb{R}) \leq \Gamma_s(I) = \Gamma_s^K(I) = \Gamma_s^D(I) \leq \Gamma_s^P(I). \tag{7.30}$$

This shows the close relationship between the condition numbers of Vandermonde matrices, Krylov matrices, matrices of moments, and Hankel matrices, at least for the monomial basis. The author then defines the constant

$$\Gamma(\mathbb{R}) \equiv \exp(2 \cdot \text{Catalan}/\pi) \approx 1.792, \tag{7.31}$$

and finally proves [26, Theorem 3.6] two bounds, one for general real intervals, and one for nonnegative intervals:

$$\frac{\Gamma(\mathbb{R})^s}{4\sqrt{s+1}} \leq \Gamma_s(\mathbb{R}) \leq \frac{\Gamma(\mathbb{R})^{s+1}}{\sqrt{2}}, \tag{7.32}$$

and

$$\frac{\Gamma(\mathbb{R})^{2s}}{4\sqrt{s+1}} \leq \Gamma_s(\mathbb{R}) \leq \frac{\Gamma(\mathbb{R})^{2s+1}}{\sqrt{2}}, \tag{7.33}$$

If we combine these results with Equation (7.30), we can bound the condition number of Krylov-monomial matrices above and below, using Bound (7.32) for general Hermitian matrices, and Bound (7.33) for Hermitian positive semidefinite matrices. These bounds show that for Hermitian matrices at least, the growth of the condition number of the monomial basis is at least exponential, and the constant is at least $\Gamma(\mathbb{R}) \approx 1.792$ (Equation (7.31)).

Newton basis

In Section 7.4.3, we discuss the connections between the choice of basis and polynomial interpolation with that basis. Reichel's [202] analysis of polynomial interpolation with the Newton basis at Leja points inspired Bai, Hu, and Reichel [18, 19] to use this technique for their s-step GMRES algorithm. Reichel showed that for most compact sets K in the complex plane, the Vandermonde-like matrix resulting from Newton polynomial interpolation at Leja points of that compact set has a condition number that grows subexponentially in the number of interpolation points. That means, for normal matrices A at least, we can expect the same subexponential growth of the Krylov basis' condition number with respect to the length s, if we use the analogous procedure to generate the basis. There, the ideal choice of the set K is the set of eigenvalues of the matrix A, but one does not know the eigenvalues of A in practice. Instead, one has the Ritz values, which are eigenvalue estimates. This difference may cause the basis condition number to grow more quickly, but if the Ritz values are good approximations of the eigenvalues, it should still be well conditioned.

This observation hints at a possible source of trouble for the Newton basis. If the Ritz values are poor approximations of the matrix's eigenvalues, then the basis condition number may grow much faster than Reichel's theory predicts. Erhel [94] observed this phenomenon, in that convergence failed for some matrices unless she used s Ritz values from $2s$ steps of standard Arnoldi, rather than s steps as one would ordinarily do. If we take a large number ($\gg s$) of steps of the iterative method to converge, then this overhead is modest and pays off. Nonnormal matrices may have ill-conditioned eigenvalues that are difficult to approximate, and it could be that the condition number of the Newton basis for highly nonnormal matrices is more sensitive to the accuracy of the eigenvalue estimates than for more well-behaved matrices.

Chebyshev basis

Joubert and Carey [144] use a mix of theoretical and heuristic arguments to suggest that with the Chebyshev basis, when the spectrum is well approximated by the bounding ellipse and the matrix A is not severely nonnormal, the condition number of the s-step basis grows no faster than $s^{3/2}$. They also point out that the position and shape of the bounding ellipse affect the basis condition number more than the size of the ellipse.

We did not have a chance to evaluate their claim experimentally. Doing so would require a numerical method for computing a "good" bounding ellipse of a finite set of points. The Computational Geometry Algorithms Library [97] offers an algorithm for computing the

unique ellipse of minimum area enclosing a set of points in the plane. As mentioned above, though, the area of the bounding ellipse is not necessarily the metric to minimize. We also achieved good numerical results by choosing from the Newton and monomial bases, especially when we incorporated techniques for keeping the basis vectors well scaled (Section 7.5). Thus, we decided to leave evaluating the stability of the Chebyshev basis for future work.

7.5 Basis scaling

Sometimes the lengths of the basis vectors may increase or decrease rapidly with respect to the basis length s. This is generally caused by the matrix having norm significantly larger resp. smaller than one. For large enough s, we observed in our numerical experiments (see e.g., Section 3.5.4) that this often causes loss of accuracy or slower convergence in the approximate solutions computed by our communication-avoiding Krylov methods. Previous authors observed this phenomenon and suggested scaling each basis vector by its Euclidean norm as it is generated. Hindmarsh and Walker [135] proposed this solution in order to avoid overflow and improve numerical stability of the "Householder GMRES" algorithm of Walker [238], which uses the monomial basis. Bai et al. [19, Algorithm 3.2] also scale each basis vector in turn by its norm in their Newton-basis GMRES algorithm. Scaling each vector after it is computed requires a factor of $\Theta(s)$ additional communication in the iterative method, which would remove its asymptotic advantage over the non-communication-avoiding version of the Krylov method. Joubert and Carey [144] instead used a scaled Chebyshev basis. The scaling factor for each basis vector is computed in advance, without requiring communication during the computation of basis vectors. However, the quality of the scaling factor depends in a complicated way on the quality of the estimated eigenvalue bounds in the algorithm. Joubert and Carey do not address this relationship or discuss basis scaling much, which could be because their test problems (PDE discretizations on regular 2-D meshes) only produce matrices with norms close to one.

We propose as a general solution, scaling the rows and columns of the matrix A (and the preconditioner also, if applicable). This requires a communication cost equivalent to a few sparse matrix-vector multiplications. However, it is only done once, as a preprocessing step for the Krylov method. When solving linear systems $Ax = b$, equilibration should be used, since it is already a natural preprocessing step for improving accuracy of solving linear systems. For solving eigenvalue problems $Ax = \lambda x$, balancing (if the matrix is nonsymmetric), followed by a scaling $A \mapsto A/\|A\|$, should be used. We found equilibration effective in practice when solving $Ax = b$ using CA-GMRES, as our numerical experiments in Section 3.5.4 show. As far as we know, no other authors have proposed scaling the matrix to solve the s-step basis vector growth problem without additional communication. The mathematical analyses in this section, in particular those explaining why the scaling of the s-step basis vectors affect numerical stability (Section 7.5.1), are also unique as far as we know.

We begin in Section 7.5.1 by explaining why the scaling of the basis vectors matters for the accuracy of methods such as CA-GMRES. Then, Section 7.5.2 informally explains basis vector growth in the context of the three different bases mentioned in this thesis. Next, Section 7.5.3 shows the two different ways of scaling the matrix A: equilibration and

balancing. We explain when each is appropriate, and outline the costs of each for general sparse matrices. Section 7.5.4 summarizes our findings.

7.5.1 Why does column scaling matter

In this section, we explain why the scaling of the s-step basis vectors can affect the numerical stability of our communication-avoiding Krylov methods. We consider CA-GMRES (Section 3.4) in this section, since of all the Krylov methods described in this thesis, it should be the least susceptible to the scaling of the basis vectors. We present experimental evidence in Section 3.5.4 that the scaling of the basis vectors does matter in CA-GMRES, but we want to show here that the phenomena we observed are not necessarily just an artifact of our implementation. What we show in this section is that poor column scaling of the matrix of basis vectors translates to poor row scaling when solving the least-squares problem for the coefficients of the new approximate solution. This is true even if the QR factorization in CA-GMRES orthogonalizes the basis vectors to machine precision. Least-squares problems are insensitive to column scaling, but sensitive to row scaling, and require additional techniques in the latter case to ensure an accurate solution. Including those techniques in CA-GMRES and our other Krylov methods is future work.

Column scaling, QR, and least squares

The QR factorization of the s-step basis vectors $\underline{V} = [V, v_{s+1}] = [v_1, \ldots, v_s, v_{s+1}]$ in CA-GMRES (Section 3.4) is insensitive to column scaling. By this, we mean that if we use Householder QR to compute the QR factorization $\underline{V} = \underline{Q}\underline{R}$ in finite-precision arithmetic,

- The Q factor is always orthogonal to machine precision
- The normwise relative residual error $\|\underline{V} - \underline{Q}\underline{R}\|_2 / \|\underline{V}\|_2$ is always small (on the order of machine precision)

This holds regardless of the scaling of the columns of $\underline{V}$. See e.g., Higham [133, Theorem 19.4 and Equation (19.12), p. 360]. While CA-GMRES uses TSQR (Section 2.3) instead of Householder QR to orthogonalize the basis vectors, it is reasonable to assume that TSQR shows the same behavior.

We can see the invariance of QR to column scaling by a simple example. Suppose we are using CA-GMRES with $t = 1$ (restarting after every invocation of the matrix powers kernel) to solve $Ax = b$. Suppose also that the $n \times n$ matrix A consists of columns 2, 3, ..., n, 1 of the diagonal matrix $\mathrm{diag}(1, 8, 64, \ldots, 8^n)$, so that A is upper Hessenberg with n nonzero entries. Suppose also that the starting vector for the monomial basis is $v = e_1$. Then, the monomial s-step basis has the form

$$\underline{V} = [e_1, 8e_2, 64e_3, \ldots, 8^s e_{s+1}]. \tag{7.34}$$

The columns of this Krylov matrix are perfectly orthogonal. If we are using base-2 floating-point arithmetic, then the QR factorization of (7.34) in floating-point arithmetic is exactly the same as the QR factorization of the first $s + 1$ columns of the identity matrix $[e_1, e_2, \ldots, e_{s+1}]$, as long as s is not so large that overflow occurs. Nevertheless, $\kappa_2(\underline{V}) = 8^s$,

so that if we measure its numerical rank only using the singular values, $\underline{V}$ is numerically rank-deficient in IEEE 754 double-precision arithmetic when $s > 13$.

Even if the column scalings are not powers of two, the normwise accuracy of the Householder QR factorization does not significantly on the scaling of the columns of the input matrix, excepting only underflow or overflow. This also means that if we use QR to solve a least-squares problem, the accuracy of the solution is insensitive to the column scaling.

Standard GMRES and least squares

To see how the scaling of the basis vectors could affect CA-GMRES accuracy, we have to look not just at the basis vectors themselves, but at the least-squares problem which CA-GMRES solves in order to compute the solution update at each outer iteration. We discussed this in Section 7.4.1, and we summarize it briefly here. Suppose that we are solving $Ax = b$, and suppose we have executed s iterations of standard GMRES. This produces basis vectors $\underline{Q} = [Q, q_{s+1}] = [q_1, \ldots, q_s, q_{s+1}$ that have been made orthogonal in exact arithmetic using Modified Gram-Schmidt (MGS) orthogonalization, as well as an $s+1$ by s upper Hessenberg matrix $\underline{H}$ such that $AQ = \underline{Q}\underline{H}$, whose entries are the MGS coefficients. Standard GMRES computes the approximate solution

$$x_{\text{MGS-GMRES}} = x_0 + Qy_{\text{MGS-GMRES}},$$

where $y_{\text{MGS-GMRES}} = y_{\text{Exact GMRES}}$ in exact arithmetic. Here, $y_{\text{Exact GMRES}}$ satisfies

$$y_{\text{Exact GMRES}} \equiv \text{argmin}_y \|b - AQy\|_2 \tag{7.35}$$

in exact arithmetic. However, standard GMRES does not compute $y_{\text{MGS-GMRES}}$ in this way; rather it computes $y_{\text{MGS-GMRES}}$ as the solution of the least-squares problem

$$y_{\text{MGS-GMRES}} \equiv \text{argmin}_y \|\beta e_1 - \underline{H}y\|_2, \tag{7.36}$$

where $\beta = \|b - Ax_0\|_2$. We see from Equation (7.36) that the accuracy of $y_{\text{MGS-GMRES}}$ is connected to the accuracy of $\underline{H}$ as computed by MGS using the Arnoldi process. Basis vectors can never grow larger than $\|A\|$ in length, since A is only multiplied by unit-length vectors. Thus, as long as GMRES does not break down, the norms of the columns, or of the rows, of $\underline{H}$ are never more than a factor of $\|A\|$ apart.

CA-GMRES and least squares

CA-GMRES computes its vector of solution update coefficients $y_{\text{CA-GMRES}}$ by solving a different least-squares problem than standard GMRES. If

$$\underline{V} = [V, v_{s+1}] = [v_1, \ldots, v_s, v_{s+1}]$$

are the s-step basis vectors computed by CA-GMRES that satisfy $AV = \underline{V}\underline{B}$ for the $s+1$ by s change of basis matrix $\underline{B}$, and if $\underline{V} = \underline{Q}\underline{R}$ and $V = QR$ are the QR factorizations of $\underline{V}$ resp. V, then CA-GMRES computes $y_{\text{CA-GMRES}}$ via

$$y_{\text{CA-GMRES}} = \text{argmin}_y \|\beta e_1 - \underline{RB}R^{-1}y\|_2. \tag{7.37}$$

This is equivalent to Equation (7.35) in exact arithmetic, but may produce different results in finite-precision arithmetic. CA-GMRES solves Equation (7.37) in three steps:

1. Compute the $s+1$ by s upper Hessenberg matrix $\underline{C} = \underline{RB}$
2. Solve the $s+1$ by s least-squares problem $\min_z \|\beta e_1 - Cz\|_2$ for z
3. Compute $y_{\text{CA-GMRES}} = Rz$

If the columns of $\underline{V}$ are badly scaled, then the columns of $\underline{R}$ are badly scaled, which means that the *rows* of $\underline{C}$ are badly scaled. This of course depends on the structure of $\underline{B}$, which depends in turn on the s-step basis being used. With the monomial basis, scaling the rows of $\underline{B}$ is equivalent to scaling the columns. However, with the Newton or Chebyshev basis, scaling the rows of $\underline{B}$ is not in general equivalent to scaling its columns.

Row scaling, QR, and least squares

While the solution of linear least-squares problems with Householder QR (or similar factorizations, like TSQR (Section 2.3)) is not sensitive to column scaling, it is sensitive to row scaling. Higham [133, Section 19.4, pp. 362-363] shows that Householder QR alone cannot guarantee small backward errors; two additional techniques must be added to the Householder QR factorization:

1. Column pivoting during the QR factorization (see Golub and Van Loan [114, Section 5.4.1, pp. 248–9])
2. One of the following:
   - Row pivoting during the QR factorization (choosing the row with largest infinity norm as the pivot row, for each column of the input matrix)
   - Before the QR factorization, sorting the rows in decreasing order of their infinity norms

Linear least-squares problems with poor row scaling do come up in applications. For example, Van Loan [171] shows how to solve equality-constrained least-squares problems of the form

$$\hat{x} = \text{argmin}_{Bx=d} \|Ax - b\|_2 \tag{7.38}$$

by transforming them into a sequence of unconstrained least-squares problems

$$\hat{x}(\mu) = \lim_{\mu \to \infty} \text{argmin}_x \left\| \begin{pmatrix} \mu B \\ A \end{pmatrix} x - \begin{pmatrix} \mu d \\ b \end{pmatrix} \right\|_2 . \tag{7.39}$$

When A and B are sparse, Van Loan's formulation avoids fill. The problem with this technique is that Equation (7.38) can result in least-squares problems with a poor row scaling. As we mentioned above, solving such problems with Householder QR alone can result in large relative errors in the computed solution $x(\mu)$, even though the original problem is well conditioned. That is, solving Equation (7.38) directly using standard techniques results in an accurate solution, but solving the equality-constrained least squares problem by transformation into Equation (7.39) and ordinary Householder QR does not. However, augmenting Householder QR with both column pivoting, and row pivoting or sorting the rows, results in an accurate $x(\mu)$. Higham [133] demonstrates this by using a small test problem from Van Loan [171, Section 3, pp. 855-857].

An inexpensive fix?

One possible way to fix the above problem, is to compute the solution of Equation (7.37) in the way described by Higham: using careful column and row pivoting to solve the least-squares problem. It should be the case that the initial QR factorization of the basis vectors $\underline{V} = \underline{Q}\underline{R}$ does not affect the accuracy of the computed $\underline{R}$ factor itself, so that we can still use TSQR to factor $\underline{V}$. This is more or less the same thing as the Rank-Revealing TSQR (RR-TSQR) factorization (Section 2.4.8): the initial invocation of TSQR reduces the size of the problem without affecting accuracy. We leave investigating this possible solution for future work.

In this section, we did not consider the effects of Block Gram-Schmidt (BGS) on the least-squares problem. That is, we did not consider the $t > 1$ case in CA-GMRES. We leave such investigations for future work.

7.5.2 Analysis of basis vector growth

In this section, we give a part theoretical, part heuristic description of s-step basis vector length growth for the monomial and Newton bases, in terms of the spectrum of the matrix A. This is much easier to do when the matrix A is normal, as then $\|A\|_2 = \lambda_{max}(A)$. When $\|A\|_2 \gg \lambda_{max}(A)$, the spectrum is much less descriptive of the basis' behavior.

Case A: A is normal

Understanding the possible cause of a poorly scaled basis is much easier when the matrix A is normal. Then the eigenvalues corresponding to the eigenvectors of A directly indicate how much the matrix "pushes" a vector in the direction of those eigenvectors, since the eigenvectors are all orthogonal. The power method reflects this direct connection, in that it converges well on a normal matrix as long as the largest two eigenvalues in absolute value are sufficiently separated.

We begin by considering just one step $v_k \mapsto Av_k$ of the s-step basis computation. This suffices for describing the behavior for normal matrices. First, suppose that the vector v_k is close to an eigenvector of A, so that $Av_k \approx \lambda_j v_k$, where λ_j is the eigenvalue corresponding to that eigenvector. In that case, the Krylov method is close to a lucky breakdown, and no s-step basis will be a good basis. For example, for the monomial basis, further s-step basis vectors $v_{k+2} = A(Av_k), \ldots$, will be nearly parallel to v_k. This is also true for the Newton and Chebyshev bases. Since the monomial basis converges faster to the principal eigenvector of the matrix A, this behavior is more likely to occur with the monomial basis, but we exclude this case in the arguments that follow.

If we assume the iterative method is not close to lucky breakdown, then we can further assume that the vector v_k has no strong components in any particular eigenvectors of A. Since A is normal, the spectrum of A governs the worst-case growth behavior of the basis. There are three cases of interest:

1. If $\|A\|_2 = \lambda_{max}(A) \gg 1$, then the basis vectors *may* grow rapidly in length as s increases.

2. If $\|A\|_2 \ll 1$, the basis vectors *must* shrink rapidly in length (since $\|A\|_2$ is the largest possible value of $\|Ax\|_2/\|x\|_2$ for any nonzero vector x).

3. If $\|A\|_2 \approx 1$, then the behavior of the basis depends in a more complicated way on the eigenvalues and eigenvectors of A, as well as on the starting vector of the s-step basis.

If we have a good estimate of $\|A\|_2$, we can reduce Cases 1 and 2 to Case 3, either by scaling the matrix A by its norm estimate, or by scaling successive basis vectors by the norm estimate. Neither of these require any communication. However, Section 7.5.3 will show more effective techniques.

For the Newton basis, the shifts $\theta_1, \ldots, \theta_s$ change the maximum scaling factor for each basis vector. We assume the (nonmodified) Leja ordering for simplicity (see Section 7.3.2). This ordering ensures that the first shift, θ_1, has the largest absolute value among all the shifts. The resulting maximum value of $\|v_2\|_2/\|v_1\|_2$ is the spectral radius of the shifted matrix $A - \theta_1 I$. This is given by Equation (7.40):

$$\rho(A - \theta_1 I) = \max_{\lambda \in \Lambda(A)} |\lambda - \theta_1|. \tag{7.40}$$

If we assume that the shifts are good approximations of the eigenvalues, and approximate the set $\Lambda(A)$ by $\{\theta_2, \ldots, \theta_s\}$ in Equation (7.40), the Leja ordering ensures that

$$\max_{\lambda \in \{\theta_2, \ldots, \theta_s\}} |\lambda - \theta_1| = |\theta_2 - \theta_1|.$$

Thus, $|\theta_2 - \theta_1|$ approximates the maximum scaling factor of $\|v_2\|_2/\|v_1\|_2$.

The extremal Ritz values tend fastest to the extremal eigenvalues of A, so θ_1 is likely a good approximation of $\lambda_{max}(A)$, if s is sufficiently large. That means $v_2 = (A - \theta_1 I)v_1$ has at most a small component in the principal eigenvector of A, in terms of the basis of eigenvectors. This means we can ignore θ_1 when considering the maximum growth factor of $\|v_3\|_2/\|v_2\|_2$. By the same argument as above, the maximum growth factor is thus well approximated by $|\theta_3 - \theta_2|$. In general, the maximum growth factor of $\|v_{k+1}\|_2/\|v_k\|_2$ for the Newton basis is well approximated by $|\theta_{k+1} - \theta_k|$.

One possibility for scaling the Newton basis vectors without communication would be, therefore, to use the above growth factor approximations:

$$\sigma_k \equiv \begin{cases} 1 & \text{for } k = 1, \\ |\theta_{k+1} - \theta_k| & \text{for } k > 1, \end{cases}$$

$$v_k \equiv \begin{cases} v_1 & \text{for } k = 1, \\ \sigma_k^{-1} \sigma_{k-1} (A - \theta_{k-1} I) v_{k-1} & \text{for } k > 1. \end{cases}$$

For the Chebyshev basis, one already uses shifted and scaled polynomials. The scaling factor comes from the center and focal locations of the estimated bounding ellipse of the eigenvalues. The result of the shift and scaling is that the ellipse is transformed into the unit circle at the origin. Thus, as long as the ellipse does actually bound all the eigenvalues, and as long as the magnitude of the largest eigenvalue is not much smaller than the two-norm of the matrix, the lengths of the basis vectors should not grow fast. (This requirement includes many more matrices than just SPD matrices.)

Case B: A is not SPD

The situation when the matrix A is not SPD, and especially when A is not normal, is much more difficult to analyze. In the latter case, the power method may not converge, even if the first and second largest eigenvalues in magnitude are well separated. This is because the eigenvalues may be sensitive to perturbations. Of course, for us the power method not converging means that it may make a good basis. In that case, however, we found experimentally that other bases, such as the Newton basis, increase much faster in condition number with s than the monomial basis. This is likely also to happen to the Chebyshev basis, since the scaling and shifting only move the eigenvalues inside the unit circle, not the two-norm of the matrix. It is difficult to know *a priori* whether this will happen, and sometimes difficult to know whether the matrix is nonnormal.

The only thing we can recommend in that case is a cautious dynamic approach. First, try one kind of basis with very small s. Observe the scaling of the basis vectors, as well as their condition number. (Both of these can be monitored, at least indirectly, from a rank-revealing factorization of the R factor from TSQR of the basis vectors.) Estimate a scaling factor if necessary and try again, scaling each basis vector in turn. If that does not help, try a different basis. Once the right basis is found, increase s, monitoring the basis condition number and scaling the whole while, until the desired value of s for performance considerations is reached.

The dynamic approach to basis selection does sacrifice performance in a start-up phase of computation. However, the information gathered during that start-up phase can be recycled, either for the same matrix with a different right-hand side, or for similar matrices.

7.5.3 Scaling the matrix A

Scaling the rows and columns of the matrix A is another way to prevent rapid growth of the basis vectors. As far as we know, we are the first to address this problem by scaling the matrix. Other authors scaled the basis vectors instead, sometimes at an additional communication cost of one reduction per vector.

We consider two different types of scaling:

1. Balancing: replacing A by $A' = DAD^{-1}$ with D diagonal
2. Equilibration: replacing A by $A' = D_r A D_c$ with D_r and D_c diagonal

Balancing preserves the eigenvalues of the matrix, which is useful when solving $Ax = \lambda x$. Equilibration may change the eigenvalues, but it may also reduce the condition number when solving linear systems $Ax = b$. Both balancing and equilibration can be computed and applied inexpensively to sparse matrices, and an approximate balancing can be computed even if the matrix A is only available as an operator $x \mapsto A \cdot x$ that performs matrix-vector multiplications. In our numerical experiments solving $Ax = b$ with CA-GMRES, we found equilibration to be effective in practice. A full theoretical and experimental comparison of the two techniques to scaling the basis vectors is future work.

Balancing

Balancing is a way of scaling the rows and columns of a square, nonsymmetric matrix A that may reduce $\|A\|_2$, without affecting the eigenvalues of A. It entails applying a similarity transform $A \mapsto A' = DAD^{-1}$ with a diagonal matrix D, where D is chosen to minimize the Frobenius norm (and equivalently, the two-norm) of A approximately:

$$D = \text{argmin}_{D \text{ diagonal}} \|DAD^{-1}\|_F.$$

If A is symmetric, then the above minimum is attained when $D = I$, so balancing is unnecessary. Balancing often improves accuracy when solving $Ax = \lambda x$ for nonsymmetric A, as typical eigensolvers can only guarantee accuracy of an eigenvalue relative to the two-norm of the matrix. See e.g., [5, "Balancing and Conditioning"].

Balancing alone does not suffice to prevent rapid growth of the s-step basis vector lengths. One must also scale the balanced matrix $A' = DAD^{-1}$ by a constant so that the absolute values of the entries of A' are not much larger than 1. For example, one may scale A' by its maximum entry in absolute value, resulting in a matrix A'':

$$A'' = \frac{1}{\max_{i,j} |A'_{ij}|} A'. \tag{7.41}$$

If we are using a Krylov subspace method to solve $Ax = \lambda x$, then we need to undo this scaling of the approximate eigenvalues after they are computed, since dividing the entire matrix by a constant also divides the eigenvalues by that constant.

LAPACK's nonsymmetric eigenvalue solver routine `xGEEVX` [5] uses the balancing algorithm of Parlett and Reinsch [190] to preprocess general dense matrices before finding their eigenvalues (see e.g., [5, "Balancing and Conditioning"] and [52]). The solver first attempts to permute the matrix so it is more nearly upper triangular. After that, the balancing algorithm iterates over pairs of corresponding rows and columns of the matrix, searching for a power of two scaling factor which makes their one-norms (not including the diagonal elements of the matrix) approximately equal. (The one-norm is chosen not for accuracy, but to save floating-point arithmetic. This procedure may be done in any norm (see e.g., [52]).) The algorithm continues passing over the matrix until the one-norms of corresponding rows and columns are equal to within a certain tolerance. Three or four passes often suffice to reduce the two-norm of the matrix sufficiently. Powers-of-two scaling factors are chosen because they do not change the significands of the entries of the matrix, in binary floating-point arithmetic. (Physicists and engineers approximate this procedure intuitively when picking the best SI units for a decimal floating-point computation.)

The balancing procedure of Parlett and Reinsch may also be applied to sparse matrices. In that case, only the nonzero values of the matrix are examined. Typical sparse matrix data structures do not support efficient iteration over both rows and columns of the matrix, though there are data structures which do. Chen [52] describes a specialized modification of the Compressed Sparse Column data structure which makes computing column and row sums (without the diagonal elements) more efficient. Depending on the data structure, the total cost is equivalent to reading the entries of the matrix 6–8 times.

A sparse matrix A can be balanced even if it is only available as an operator $x \mapsto A \cdot x$. Chen [52] describes so-called *Krylov-based balancing algorithms* that use only sparse matrix-vector products with A and A^* ("two-sided"), or just with A ("one-sided"). Two-sided

algorithms tend to be more effective, and five iterations (each of which involve computing $A \cdot x$ and $A^* \cdot y$ for some vectors x and y) often suffice to compute a good approximate balancing. Here, the vectors are chosen from a certain random distribution. Chen's two-sided algorithm gives $D = I$ if A happens to be symmetric, so it does not make a symmetric problem nonsymmetric.

Note that balancing is often defined to include an initial permutation step, that attempts to put A in upper triangular or block upper triangular form. See [5, "Balancing and Conditioning"] and [52]. The permutation step may save computation when computing the eigenvalue decomposition of a dense matrix, but offers no such benefits for iterative methods. Also, the matrix powers kernel may require another permutation for performance reasons (see Section 2.1).

Equilibration

Equilibrating a matrix A means applying diagonal row and column scalings D_r resp. D_c, so that the norms of each row and column of the equilibrated matrix $D_r A D_c$ are all one. (This is not possible when A has a zero row or column, so we assume this is not the case.) Equilibration may be done in any of various norms. Typically the norm used is the infinity norm $\|\cdot\|_\infty$, which means that the largest element in each row or column of $D_r A D_c$ has magnitude one. The 1-norm $\|\cdot\|_1$ may also be used, in which case, if A contains only nonnegative entries, then $D_r A D_c$ is *doubly stochastic* (all the entries are nonnegative, and each row or column sums to one).

Equilibrating the matrix A helps prevent excessive growth or decay of the s-step basis vector lengths as a function of s. For example, if A is $n \times n$ and $e = [1, 1, \ldots, 1]^T$ is length n, then equilibrating A in the 1-norm makes $\|A^k e\|_1 = 1$ for $k = 1, 2, 3, \ldots$. Equilibrating A in the infinity norm makes $\|A^k e_j\|_\infty = 1$ for $k = 1, 2, 3, \ldots$, where e_j is column j of the $n \times n$ identity matrix. Equilibration also improves the accuracy of solving linear systems. For example, if D is the diagonal of the $n \times n$ SPD matrix A, then van der Sluis [225, Theorem 4.1] showed that the equilibration $A \mapsto D^{-1/2} A D^{-1/2}$ minimizes the 2-norm condition number of A (with respect to solving linear systems) over all such symmetric scalings, to within a factor of the dimension n of A. Consequently, equilibration is not just useful for s-step Krylov methods, but useful in general.

There are many options for equilibrating sparse matrices. For example, for our numerical experiments with nonsymmetric matrices, we adapted the LAPACK subroutine `DGEEQU` for dense matrices (see [5]), which equilibrates in the infinity norm. There are a variety of options for nonsymmetric matrices. See e.g., Riedy [203]. For symmetric matrices, the algorithm based on `DGEEQU` does not preserve symmetry. It first computes row scaling factors, and then computes column scaling factors of the row-scaled matrix; those two sets of factors may be different. However, it is possible to equilibrate in a way that preserves symmetry. If A is symmetric positive definite, then $A \mapsto D^{-1/2} A D^{-1/2}$ with $D_{ii} = A_{ii}$ results in an equilibrated SPD matrix. If A is symmetric but indefinite, there are a number of alternatives. Bunch [42] gives an algorithm for equilibrating dense symmetric indefinite matrices. It makes one left-to-right pass in a left-looking fashion over the lower triangle of the matrix, and produces an equilibrated matrix satisfying the same criteria as the output of LAPACK's `DGEEQU`. MC30 [66] in the HSL library [139] produces a scaling $A \mapsto DAD$ that makes all the entries as

close to one in absolute value as possible. It does this by minimizing the sum of squares of the natural logarithms of the magnitudes of the entries of the scaled matrix. Ruiz [206] presents an iterative equilibration algorithm for sparse matrices that respects symmetry. It converges linearly at a rate of at least 1/2 per pass over the matrix. The algorithm has been implemented as MC77 in the HSL 2004 numerical library [139]. Livne and Golub [170] develop a different iterative algorithm, BIN (for "binormalization"), that respects symmetry. In the SPD case, it sometimes produces a scaled matrix with a lower condition number than scaling symmetrically by the diagonal. Duff and Pralet [91] compare a number of equilibration schemes for symmetric indefinite matrices.

The cost of equilibration depends both on the equilibration algorithm, and how the matrix is stored. Our translation for sparse matrices of LAPACK's dense nonsymmetric equilibration `DGEEQU` requires two read passes over the entries of the sparse matrix, once for the row scalings and once for the column scalings. (Even if the matrix is stored in CSR format (as a row-oriented adjacency list), the second pass for the column scalings still only requires iterating over the nonzero values and their column indices once.) Bunch's symmetric indefinite equilibration [42] requires reading and writing each row of the lower triangle of the matrix once, and writing each column of the lower triangle once. If the matrix is sparse, that amounts to reading the nonzeros twice and writing the nonzeros once. However, the algorithm constrains the read and write order in a way that could hinder an efficient implementation, if the sparse matrix is stored in either a row-oriented or a column-oriented format. Ruiz's iterative scaling method [206] has a similar cost per iteration, but imposes fewer constraints on the order of reading the matrix entries. The maximum weighted matching step in the scaling method of Duff and Pralet [91] requires in the worst case, for a nonsymmetric matrix, $\Theta(\text{nnz} \cdot n \log n)$ operations, where nnz is the number of nonzeros and n is the dimension of the matrix.

Nonsymmetric equilibration is less expensive than balancing, when the entries of the sparse matrix A can be accessed directly. For a general nonsymmetric sparse matrix, equilibration only requires reading the nonzero values of the matrix twice: once when iterating over rows, and once again when iterating over columns. Balancing may require several passes over the rows and columns of the matrix. When the matrix A is only available as an operator $x \mapsto A \cdot x$ (and possibly $y \mapsto A^* \cdot y$), equilibration is more difficult. One could use a randomized approach to estimate the largest nonzero absolute value of each row and column of A. However, balancing seems more natural in this case than equilibration. Accompanied by a constant scaling of the entire matrix, it also achieves the desired effect of restricting s-step basis length growth. In that case, one may use one of the randomized Krylov-based balancing algorithms of Chen [52]. We do not investigate this case further in this thesis.

Row scalings and residual norms

Scaling the rows of the matrix A affects the residual $r_k = b - Ax_k$, whose norm is part of the stopping criterion for many iterative methods. The pre-scaling residual norm may be computed without communication, though it does require an additional division per element of the vector. Suppose $A \mapsto D_r A D_c$ is the row and column scaling (which may be an equilibration or a balancing), and that x_k is the current approximate solution in the

iterative method running on the scaled matrix. Let r_k be the residual norm before scaling:

$$r_k = b - Ax_k,$$

and let $\tilde{r}_k$ be the residual norm after scaling:

$$\tilde{r}_k = D_r b - D_r A D_c (D_c^{-1} x_k) = D_r r_k.$$

The iterative method normally only works with the scaled matrix $D_r A D_c$, so the residual it computes is the post-scaling residual $\tilde{r}_k$. However, as long as the diagonal row scaling D_r is available, computing r_k from $\tilde{r}_k$ only requires elementwise division. Furthermore, Equation (7.42) bounds how much the pre-scaling relative residual $\|r_k\|/\|r_0\|$ may differ from the post-scaling $\|\tilde{r}_k\|/\|\tilde{r}_0\|$:

$$\frac{\min(D_r)}{\max(D_r)} \leq \frac{\frac{r_k}{r_0}}{\frac{\tilde{r}_k}{\tilde{r}_0}} \leq \frac{\max(D_r)}{\min(D_r)} \tag{7.42}$$

These lower and upper bounds may be used to avoid computing the pre-scaling residual, if the post-scaling residual norm is sufficiently large that the iterative method could not have converged to within the desired tolerance yet.

Experimental note

Scaling the matrix is a special case of preconditioning, so we should expect that it affect the convergence of the iterative method, even in exact arithmetic. This is why, when comparing s-step iterative methods with their conventional counterparts, one should use the same matrix scaling for each.

7.5.4 Conclusions

In our experiments solving nonsymmetric linear systems with CA-GMRES (Section 3.4), we found that for practical problems and basis lengths s that achieve good performance, equilibration almost made the basis type irrelevant. That is, either the monomial or the Newton basis could be used without affecting numerical stability. Section 3.5.4 in particular gives examples of sparse matrices from applications, some very badly scaled, for which equilibration meant the difference between CA-GMRES not converging at all, and CA-GMRES converging at the same rate as standard GMRES (which it should do in exact arithmetic). Of course, equilibration by itself is not enough to ensure stability of CA-GMRES, if the matrix is badly conditioned. Our CA-GMRES numerical experiments with diagonal matrices in Section 3.5.2 illustrate that when the matrix is badly conditioned, the monomial basis may not converge once s is sufficiently large, whereas the Newton basis often does converge. Equilibration makes scaling the basis vectors by their norms (a communication latency - intensive procedure) unnecessary for practical problems. Furthermore, computing an equilibration requires the equivalent of only 2 passes over the nonzero entries of a sparse matrix. Computing a balancing requires the equivalent of a few more passes, and is appropriate when solving $Ax = \lambda x$.

7.6 Future work

Much of the basis condition number analysis in Section 7.4 requires refinement. First, the influence of the projection step (as discussed in Section 7.4.2) is not well understood. The projection step allows the basis length s to be smaller than the restart length, which has advantages both for performance and accuracy, so it is important to understand how projecting the current basis vector onto the orthogonal complement of the previous basis vectors affects their condition number. It is also important to develop inexpensive tests for loss of independence of the s-step basis vectors. The rank-revealing version of TSQR may be used in that case, as we discuss in Section 2.4.8.

There are also unresolved questions with the Newton basis. The influence of the modified Leja ordering and the real-arithmetic variant of the Newton basis on the condition number is not understood. We also observed in experiments that the Newton basis often does much worse than the monomial basis on highly nonnormal matrices. This may have something to do with the power method being unstable for highly nonnormal matrices, so that the monomial basis does not converge. Alternately, it could have to do with the Ritz values being poor approximations to the eigenvalues of the matrix A, so that the shifts in the Newton basis just make the condition number worse. Both experiments and theoretical analysis are required to address these issues.

There are also questions left to answer about the Chebyshev basis. The condition number of the Chebyshev basis is affected by how closely the chosen bounding ellipse approximates the spectrum of the matrix A, but this relationship requires more study. It's known that for Chebyshev iteration [172], the estimates of the field of values must be fairly accurate in order to ensure that the method converges. It's not clear whether this observation carries over to the condition number of a Chebyshev basis for a general s-step Krylov method, but intuition suggests this. This is because the minimax property of Chebyshev polynomials follows from their regular alternation in sign within the minimax region. The exhaustion of all roots and inflection points within that region means that Chebyshev polynomials grow fast in absolute value outside of it. If the bounding region is too small, then the Krylov basis effectively evaluates Chebyshev polynomials at eigenvalues of the matrix A which lie outside the region. (This is a problem for symmetric Lanczos, for which the extremal Ritz values converge outwards to the extremal eigenvalues.) However, if the bounding region is too large, then the resulting basis may not have a suboptimal condition number. Furthermore, the bounding region may unnecessarily contain forbidden points, like the origin.

One way to solve this problem is to allow more general bounding regions than ellipses. The resulting polynomials are called *Faber polynomials*, of which the shifted and scaled Chebyshev polynomials are a special case. Koch and Liesen [157] address this with their delightfully named "conformal bratwurst maps," which are easily computable Riemann maps (in the sense of the Riemann Mapping Theorem – see e.g., [167, Theorem 8.1, p. 322]) from parametrically distorted ellipses to the unit disk.

One ultimate goal would be to have complete backward stability proofs for all of our iterative methods. Such proofs have been developed in great detail for standard GMRES. See e.g., Greenbaum et al. [121] and the extensive bibliography in Paige et al. [186].

Appendix A

Experimental platforms

Most of the performance results in this thesis refer to shared-memory parallel benchmarks, run on two experimental platforms: an Intel Nehalem node and an Intel Clovertown node. This includes performance results both for individual kernels – such as the matrix powers kernel (Section 2.1), TSQR (Section 2.3), and BGS (Section 2.4) – and for the CA-GMRES solver built of these kernels (Section 3.6). Table A.1 compares their architectural features of these two platforms.

We sketch the architecture of each node pictorially in Figures A.1 (for the Nehalem) resp. A.2 (for the Clovertown). The chief difference between the two platforms is that the Nehalem has more memory bandwidth than the Clovertown, both actually (especially for writes) and effectively (due to NUMA, so that the two sockets do not need to share a single connection to memory). The result of more memory bandwidth is that our CA-GMRES has a somewhat smaller speedup over standard GMRES, as one would expect.

There are two other features which the Nehalem processor has that the Clovertown does not. First, the Nehalem supports *simultaneous multithreading* (SMT). SMT is a technique allowing a single processor core to share its resources between multiple threads of execution. We found that SMT did not improve the performance of either standard GMRES or CA-GMRES, so we did not exploit it; we pinned one thread to each processor core. Second, the Nehalem has a "TurboMode" feature, in which each processor core is allowed to control its clock rate automatically and independently of the other processors. This lets one core run faster if the other cores are not busy, but throttles back the frequency of all the cores if they are all busy. The purpose of TurboMode is to control the temperature of the processor, since higher frequencies use more power and thus dissipate more heat. We disabled TurboMode in order to make the timings regular, since otherwise the CPU may change the frequency automatically and without programmer control. While this biases the timings to favor using more cores, TurboMode does not affect the memory bandwidth, which is the more important hardware feature for these tests.

Manufacturer	Intel	Intel
Nickname	Nehalem	Clovertown
Number	X5550	X5345
# threads / core	2	1
# cores / socket	4	4
# sockets	2	2
CPU freq. (GHz)	2.66	2.33
Peak Gflop/s	85.3	74.66
Caches	256 KB L2 per core, 8 MB L3 per socket	4 MB L2, shared by every 2 cores
Peak memory bandwidth (GB/s)	2×25.6 (read or write)	21.33 (read) 10.66 (write)
Memory type	1066 MHz DDR3	667 MHz FBDIMM
DIMMs	12×1 GB	8×2 GB
NUMA?	Yes	No

Table A.1: Characteristics of the two hardware platforms used for our shared-memory parallel performance experiments. "# threads / core" refers to the simultaneous multithreading (SMT) feature; if 1, the processor does not have SMT. "CPU freq." is the CPU frequency, and "Peak Gflop/s" the peak double-precision floating-point performance. "Peak memory bandwidth" is a shared resource among all cores, unless shown as $S \times B$, in which case each of S sockets has B memory bandwidth. "DIMMs" is the number N and capacity C (in GB) of DRAM memory units, shown as "$N \times C$." "NUMA?" is "Yes" if the memory is NUMA and "No" otherwise.

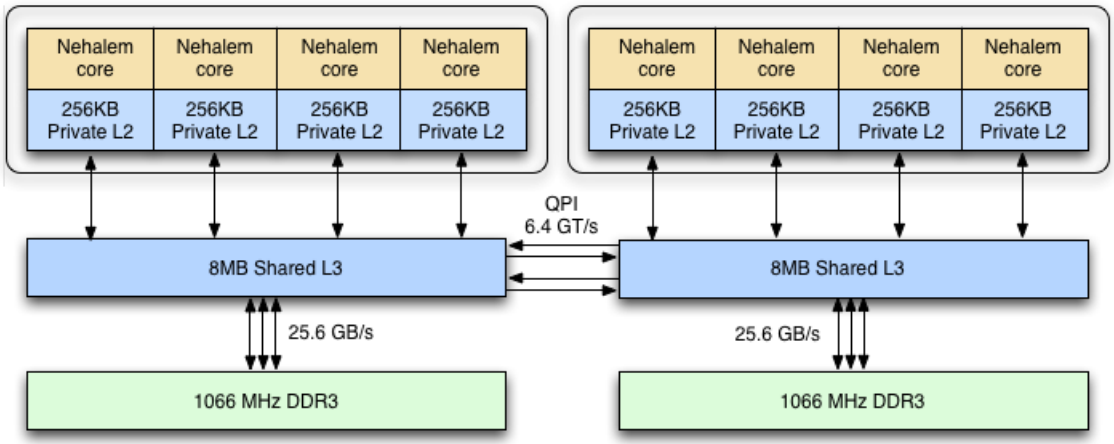


Figure A.1: Architecture of the Intel Nehalem processor used in our performance experiments. Each socket (shown as a white box with rounded corners) contains four processor cores ("Nehalem core"), and each core has its own private L2 cache. "QPI" refers to the "Quick-Path Interconnect," which replaces the Front-Side Bus (FSB) in earlier Intel architectures. It can handle 6.4 billion transactions (GT) per second.

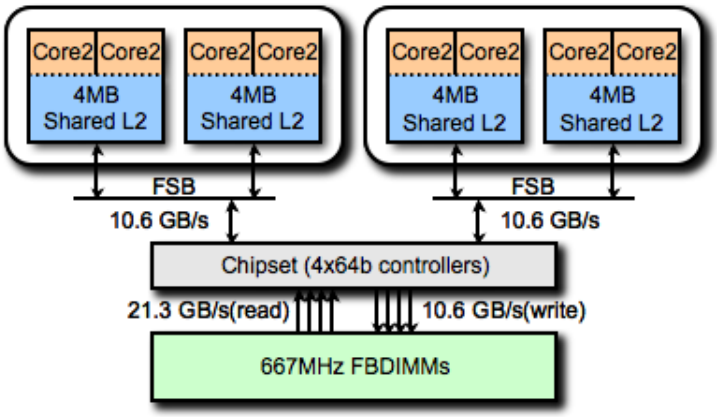


Figure A.2: Architecture of the Intel Clovertown processor used in our performance experiments. Each socket (shown as a white box with rounded corners) contains two dies. Each die has two cores (each is shown as "Core2," which is the brand name of the core type, and does not mean that there are two cores there), which share a single shared L2 cache. "FSB" refers to the "Front-Side Bus," which is a single bus which all processors must use in order to read resp. write data from resp. to main memory.

Appendix B

Test problems

B.1 Test matrices

We tested our iterative methods on several different classes of matrices. Some of them come from real applications, and others are contrived problems. We also show performance results for some of the matrices from applications in Section 3.6, comparing optimized versions of CA-GMRES (see Section 3.4) and standard GMRES. The contrived test problems are not as interesting from a performance standpoint, but they do a better job than the "real-life" matrices of exercising the techniques we use to improve numerical stability.

The first class of matrices tested are positive diagonal matrices with varying condition numbers. We discuss these test problems in detail in Appendix B.3. When the condition number is relatively large, the Newton basis is more stable than the monomial basis.

The second class of test problems come from existing literature on s-step methods, in particular s-step versions of GMRES by Bai et al. [19]. We chose this class of problems in order to verify our methods against previous work. All of these test problems are linear convection-diffusion PDEs on a 2-D square domain, discretized with a regular mesh. The problems have parameters which can be adjusted to increase the asymmetry of the matrix in a physically plausible way (by increasing the ratio of convection to diffusion, for example). This is important because some s-step bases require estimates of the eigenvalues of the matrix, and nonsymmetric matrices have more interesting spectra.

The third class of matrices tested come from physical applications. We first chose them in Demmel et al. [79] to illustrate the performance benefits of the matrix powers kernel and block orthogonalization optimizations, on "real-life" problems. The applications represented include structural engineering, computational fluid dynamics, and computational chemistry. Some are symmetric, and some are not. In addition, some of the matrices have entries which range widely in magnitude. This tests the use of equilibration in order to avoid under- or overflow in the s-step basis (see Section 7.5). In fact, we found that for this class of matrices, the choice of basis often matters much less than the use of equilibration. This was not true for some other classes of matrices we tested. Also, for some of these problems, the Newton basis is less stable than the monomial basis.

The fourth class of matrices tested includes only one matrix, WATT1, a sparse matrix from petroleum applications that was suggested by Erhel [94] as a nonsymmetric test problem

for s-step GMRES. The spectral properties of WATT1 make it especially interesting and challenging to construct a well-conditioned s-step basis. As we show in Section 3.5.5, it also justifies our innovation in CA-GMRES of choosing the s-step basis length s smaller than the restart length $r = s \cdot t$.

B.2 Exact solution and right-hand side

Whenever we solve a linear system $Ax = b$ with a test matrix A, we choose the right-hand side b by first picking a known exact solution x, and then computing $b = A \cdot x$. For all test problems, we choose the exact solution x as the sum of both a "white noise" component and a more slowly varying periodic component: for $x \in \mathbb{R}^n$, the k^{th} component of x is given by

$$x(k) = u(k) + \sin(2\pi k/n),$$

where the scalar $u(k)$ is chosen from a random uniform $[-1, 1]$ distribution. We choose x in this way because a completely random solution is usually nonphysical, but a highly nonrandom solution (such as a vector of all ones) might be near an eigenvector of the matrix (which would result in artificially rapid convergence of the iterative method).

B.3 Diagonal matrices

We first test each new Krylov method in this thesis with positive diagonal matrices. In exact arithmetic at least, they suffice to investigate symmetric matrices completely, because symmetric matrices are equivalent to diagonal matrices via orthogonal transformation. They are also a useful "sanity check" for nonsymmetric Krylov methods.

We choose the matrices to have dimensions 10000×10000, and vary their condition numbers. Their diagonal entries have values logarithmically spaced between 1 and $1/\kappa$, where κ (the only parameter) is the 2-norm condition number of the matrix. Since these diagonal test matrices have a positive diagonal and their maximum element is always 1, they need no further equilibration or scaling in order to avoid underflow or overflow in the matrix powers kernel (see Section 7.5). (Equilibration would make any diagonal matrix the identity, and balancing does not change a symmetric matrix.) This means that the only factor affecting convergence of our iterative methods on these matrices is the distribution of their eigenvalues.

While these are the only diagonal matrices we test in this thesis, in general diagonal matrices can be used to generate a wider range of test problems. For example, one may choose the diagonal matrix to have norm $\gg 1$ or $\ll 1$, in order to investigate the effects of a poorly scaled s-step basis (see Section 7.5). Diagonal matrices also let us study effects related to the distribution of eigenvalues and singular values, independently of other properties such as normality. For many possible choices of diagonal matrices for testing the convergence of CG, see also D'Azevedo et al. [70].

Constructing more general matrices with a given set of eigenvalues or singular values is a challenging problem in itself, which we did not take the time to solve in this thesis. For examples of such "inverse eigenvalue problems," their practical applications, and how

to solve them, see e.g., Chu [61]. A simple way to construct a nonsymmetric matrix with a given set of singular values is to make the singular values the diagonal of a diagonal matrix, and then permute the rows and columns of that matrix nonsymmetrically. The row resp. column permutation matrix forms the left resp. right singular vectors, which are orthogonal by construction (since they are a permutation).

B.4 Convection-diffusion problem

In this section, we describe a partial differential equation (PDE) boundary-value problem with three parameters, and its discretization, resulting in a sparse linear system $Ax = b$. Bai et al. [19] use this problem and discretization to test their Newton-basis s-step GMRES algorithm. Typical values of the three parameters result in a nonsymmetric but nonsingular matrix. This suits the problem for testing iterative methods that work on nonsymmetric matrices, such as variations of GMRES or BiCGSTAB. We use the discretization to test the convergence of our CA-GMRES algorithm. Of the many such PDE discretization test problems available, we chose that of Bai et al. because they used it to investigate convergence of an s-step GMRES method.

B.4.1 The problem is representative

PDE discretizations on 2-D domains are popular test problems for Krylov subspace methods. We describe examples from three different authors, in order to show that the test problem of Bai et al. are representative of other authors' test problems for nonsymmetric iterative solvers.

Saad and Schultz [209] compare the convergence of GMRES, GCR, and ORTHOMIN(k) using the five-point discretization of a second-order linear 2-D PDE on the unit square with Dirichlet boundary conditions, with variable coefficients and a forcing term chosen so that the solution is known and infinitely differentiable. The two first-derivative terms involve multiplicative real parameters β resp. γ. They do not affect the solution of the PDE, but they do affect the symmetry of the discretized linear system. Choosing the test problem so that the underlying PDE has infinitely differentiable coefficients and solution on its domain could be a way to isolate the effects of the matrix itself on the iterative method, from the potential effects of a difficult-to-solve PDE.

Another example comes from Elman et al., in which they test their hybrid Chebyshev-Krylov subspace method [92]. The test involves the following 2-D second-order PDE:

$$-(e^{-xy}u_x)_x - (e^{xy}u_y)_y + \gamma\left((x+y)u_y + ((x+y)u)_y\right) + \frac{u}{1+x+y} = f(x,y) \tag{B.1}$$

on the unit square $\Omega = [0,1]^2$, with homogeneous Dirichlet boundary conditions $u(x,y) = 0$ on $\partial\Omega$. In Equation (B.1), γ is a real scalar parameter, and the right-hand side $f(x,y)$ is chosen to make the exact solution $u(x,y)$ as follows:

$$u(x,y) = xe^{xy}\sin(\pi x)\sin(\pi y). \tag{B.2}$$

Elman et al. discretize this PDE using the usual five-point second-order centered finite difference stencil on a regular mesh. They vary γ (choosing $\gamma = 5$ and $\gamma = 50$) in order to vary the nonsymmetry of the matrix.

A third example is Sonneveld's [214] "Problem 1," which he uses to compare Bi-CG, ORTHOMIN, and CGS. The 2-D second-order PDE is as follows:

$$-\epsilon \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) + \cos(\alpha) \frac{\partial u}{\partial x} + \sin(\alpha) \frac{\partial u}{\partial y} = 0 \tag{B.3}$$

again on the unit square $\Omega = [0, 1]^2$, and again with homogeneous Dirichlet boundary conditions $u(x, y) = 0$ on $\partial\Omega$.[1] In Equation (B.3), $\epsilon > 0$ and α are real-valued parameters. Sonneveld chooses to discretize the problem on a square regular mesh with $N = (1/\Delta x - 1)^2$ interior points, with a five-point hybrid difference scheme: upwind differencing is done only if necessary to maintain diagonal dominance, otherwise central differencing is done. The α parameter lets users make the matrix progressively less symmetric (as it varies away from $\pi/4$). Decreasing the ϵ parameter decreases the amount of diffusion in the problem, and therefore decreases the influence of a positive definite additive term on the matrix. Thus, the two parameters govern the two matrix attributes of interest in the comparison study: symmetry, and positive definiteness.

B.4.2 The partial differential equation

The test problem of Bai et al. is a linear convection-diffusion PDE on the unit square $\Omega = [0, 1]^2$, with homogeneous Dirichlet boundary conditions. It is given by Equation (B.4).

$$-\Delta u + 2p_1 u_x + 2p_2 u_y - p_3 u = f \tag{B.4}$$

on $\Omega = [0, 1]^2$, with homogeneous Dirichlet boundary conditions $u(x, y) = 0$ on $\partial\Omega$. In Equation (B.4), p_1, p_2, and p_3 are constant parameters. The right-hand side function $f(x, y)$ is chosen so that the exact analytic solution $u(x, y)$ is given by

$$u(x, y) = xe^{xy} \sin(\pi x) \sin(\pi y). \tag{B.5}$$

Note that the parameters p_1, p_2, and p_3 do not go into the exact solution $u(x, y)$; rather, they go into the PDE and the forcing function $f(x, y)$. We do not show the latter function here because it is long, and also because the derivation is not informative. (We used a computer algebra system to check our derivation.)

B.4.3 Its discretization

Bai et al. discretize Equation (B.4) using centered finite differences on a regular $n + 2$ by $n + 2$ grid (including boundary points) with grid spacing $h = 1/(n + 1)$ in each dimension. This results in an $n^2 \times n^2$ system of linear equations $Ax = b$. Bai et al. do not precondition this system. They use different values of the parameters and different restart lengths to construct three different test problems, which are illustrated in Table B.1.

[1] The paper omits the Dirichlet boundary condition, though this is easy to infer from the fact that Sonneveld only discretizes on interior mesh points.

Experiment #	Matrix dimension	p_1	p_2	p_3	Restart length
1	3969	1	1	20	20
2	3969	1	1	20	30
3	3969	2	4	30	25

Table B.1: Matrix dimensions, parameter values, and GMRES restart lengths used by Bai et al. [19] in their numerical experiments. Matrix dimension 3969 corresponds to $n = 63$ grid points in each dimension, not including boundary points.

Bai et al. use this family of test problems to test their Newton-basis s-step GMRES algorithm. They helpfully show plots of the 2-norm of the residual error over several iterations of the method, for both their Newton basis and for a monomial basis. It is important to note that Bai et al. scale their s-step basis vectors so that all of them have unit length, before computing their QR factorization. They do this for the same reason that we equilibrate the matrix: in order to avoid excessive growth in the basis vector lengths that causes inaccuracy in the QR factorization, or even overflow or underflow. See Section 7.5 for details. We choose to equilibrate instead of scaling the basis vectors, because computing the length of each basis vector separately requires additional communication. In parallel, it requires $\Theta(\log P)$ messages per vector, and sequentially, it requires reading and writing each basis vector. Part of the comparison between the results of Bai et al. and our results, therefore, involves the effects of equilibration, so we show results both with and without equilibration.

B.5 Sparse matrices from applications

In Section 3.5, we perform numerical experiments comparing CA-GMRES and standard GMRES, using a collection of test matrices from applications. In Sections 2.1 and 3.6, we show performance results for nearly the same set of matrices. Here, we describe the structural and numerical properties of those matrices, and list their sources. We introduced these test problems in Demmel et al. [79] to investigate the performance both of the matrix powers kernel by itself, and of CA-GMRES as a whole. We also make brief reference to numerical results in that paper, but this thesis is where we first study the numerical properties of CA-GMRES and other communication-avoiding iterative methods in detail. For some of those matrices, only the nonzero pattern and not the nonzero values were available at the time we submitted [79]. For those matrices for which we have the nonzero values as well, we present numerical results in this thesis.

Table B.2 lists the matrices and some of their structural properties, including dimensions and number of nonzero entries. That table includes numbers which refer to the sources of the matrices. Those numbers correspond to those in the list of sources below. In this list, "UF" refers to the University of Florida Sparse Matrix Collection [68]. Within the UF collection are named various "groups," which help to identify the source of the matrix in the collection. All but one of the matrices are available from the UF collection.

1. UF, `GHS_psdef` group: finite element method (FEM) stiffness matrix with two strongly connected components, from a car model

Short name	Source name	Dim	nnz	Sym?	SPD?	Source
bmw	bmw7st_1	141K	3.7M	Y	Y	(1)
cant	Cant_A	62K	4.0M	Y	Y	(2)
cfd	cfd2	123K	3.1M	Y	Y	(3)
pwtk	pwtk	218K	11M	Y	Y	(4)
shipsec	shipsec1	141K	3.6M	Y	Y	(5)
xenon	xenon2	157K	3.9M	N	N	(6)

Table B.2: Test matrices in [79] for which we perform numerical experiments in this thesis. "Short name" refers to the label used in [79], and "Source name" refers to the label used in the source of the matrix. "Dim" is the matrix dimension (number of rows – all matrices in the table are square). "nnz" (number of nonzero elements in the sparse matrix) includes both numerical nonzero elements, and explicitly stored zeros (if there are any). "Sym?" is Y if the matrix is symmetric, N if not. "SPD?" is Y if the matrix is SPD, N if it is not. The source number refers to the list of sources in this section (Section B.5).

Short name	Source name	Dim	nnz	Sym?	SPD?	Source
1d3pt	1d3pt	1M	3M	Y	Y	(7)
1d5pt	1d5pt	1M	5M	Y	Y	(7)
2d9pt	2d9pt	1M	9M	Y	Y	(7)
marcat	TCOMM	548K	2.7M	N	N	(8)
mc2depi	TwoD	526K	2.1M	N	N	(9)

Table B.3: Test matrices in [79] for which we do not perform numerical experiments in this thesis. "Short name" refers to the label used in [79], and "Source name" refers to the label used in the source of the matrix. "Dim" is the matrix dimension (number of rows – all matrices in the table are square). "nnz" (number of nonzero elements in the sparse matrix) includes both numerical nonzero elements, and explicitly stored zeros (if there are any). "Sym?" is Y if the matrix is symmetric, N if not. "SPD?" is Y if the matrix is SPD, N if it is not. The source number refers to the list of sources in this section (Section B.5).

2. Adams [2]: FEM stiffness matrix, cantilever elasticity model, with large jumps in material coefficients and incompressible materials. I received this matrix via personal communication, as it is no longer available at its former web address (`http://www.columbia.edu/~ma2325/femarket/`).

3. UF, `Rothberg` group: symmetric pressure matrix from computational fluid dynamics

4. UF, `Boeing` group: pressurized wind tunnel stiffness matrix

5. UF, `DNVS` group: ship section / detail (structural problem) with three strongly connected components

6. UF, `Ronis` group: analysis of elastic properties of complex zeolite, sodalite crystals

Table B.3 lists those matrices for which performance experiments are included in [79], but on which we do not perform numerical experiments in this thesis. Either the matrices

are too simple (such as the 1-D Laplacian) to be interesting, or their nonzero values are not available and the matrix cannot be reconstructed from its supposed source. The matrices in the latter category were collected in a test suite for measuring the performance of tuned sparse matrix-vector multiply implementations. Vuduc [235] cites Im [140] as the source for some these matrices, and Im cites both Davis [68] and Li [168] as the source. Li does not include the `marcat` or `mc2depi` matrices in her thesis, so their ultimate provenance is unclear. We have traced them to a collection of Markov chain test problems by Stewart [218], but there these are presented as parametrized problems, and the parameters which produced the matrices with the attributes we have are unclear. Table B.3 includes source numbers, which correspond to the numbers in the list below.

7. $\langle n \rangle$d$\langle k \rangle$pt refers to a sparse matrix constructed from a regular discretization of the Poisson equation on the domain $[0, 1]^n$ (n dimensions), using a k-point stencil. The nonzero elements of the matrix are stored explicitly, even though the discretization does not require this.

8. UF, `Rothberg` group: Aircraft flap actuator model

9. Stewart [218]: A queueing network model of impatient telephone customers on a computerized telephone exchange. The model is parametrized, but the values of the original parameters which produced this test matrix have been lost. See also Philippe et al. [196], where the authors use this model (with parameters resulting in smaller dimensions and fewer nonzeros) to test various numerical methods (including Arnoldi) to find the steady-state vector of the Markov chain.

10. Stewart [218]: general 2D Markov chain model – in this case, a Ridler-Rowe model of an epidemic. The model is parametrized, but the original parameters which produced this test matrix have been lost. See also Pollett and Stewart [197].

Finally, for the matrices for which we perform CA-GMRES numerical experiments in Section 3.5, we show properties relating to equilibration in Tables B.4 and B.5. In particular, we perform symmetric equilibration by scaling symmetrically by the absolute value of the diagonal entries. If some of the diagonal entries in absolute value are greater than one, and some are less than one, we scale symmetrically only those rows and columns corresponding to the diagonal elements with absolute value greater than one.

B.6 WATT1 test matrix

In this section, we describe WATT1, a sparse matrix suitable for testing nonsymmetric s-step iterative methods, such as CA-GMRES and CA-Arnoldi. The spectral properties of WATT1 make it especially interesting and challenging to construct a well-conditioned s-step basis. This makes it a good test problem for comparing the numerical stability of s-step Krylov methods with their corresponding standard Krylov methods. Erhel [94] used WATT1 to test her implementation of the Newton-basis s-step GMRES algorithm of Bai et al. [19]. As we showed in Section 3.5.5, it also justifies our innovation in CA-GMRES of choosing the

Matrix name	$\min\{\lvert A_{ij}\rvert \neq 0\}$	$\max(\lvert A_{ij}\rvert)$	$\min\{\lvert A_{ii}\rvert \neq 0\}$	$\max \lvert A_{ii}\rvert$
bmw7st_1	5.0487×10^{-29}	5.33×10^{15}	5.05×10^{-29}	5.33×10^{15}
Cant_A	5.5511×10^{-17}	7.05×10^{3}	5.55×10^{-17}	7.05×10^{3}
cfd2	6.6582×10^{-9}	1.00×10^{0}	1.00×10^{0}	1.00×10^{0}
e20r5000	3.2526×10^{-19}	9.40×10^{1}	N/A	N/A
e40r5000	8.5665×10^{-20}	4.11×10^{1}	N/A	N/A
pwtk	4.0390×10^{-28}	3.67×10^{7}	4.04×10^{-28}	3.67×10^{7}
shipsec1	2.8422×10^{-14}	1.42×10^{13}	2.84×10^{-14}	1.42×10^{13}
xenon2	5.4251×10^{23}	3.17×10^{28}	N/A	N/A

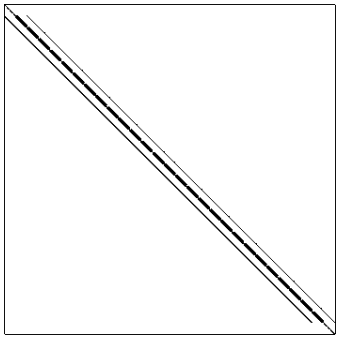
Table B.4: For the matrices from applications described in this section: the minimum and maximum nonzero elements in absolute value of each matrix, along with the minimum and maximum diagonal elements in absolute value of each matrix. The latter matters for symmetric positive definite equilibration (symmetrically scaling by the absolute value of the diagonal), so if the matrix is nonsymmetric, we show "N/A" (not applicable) there. All of these values are pre-equilibration, if applicable.

Matrix name	$\min\{\lvert A_{ij}\rvert : A_{ij} \neq 0\}$	$\max(\lvert A_{ij}\rvert)$	Equilibrated?
bmw7st_1	7.1693×10^{-35}	1.0000×10^{0}	Y
Cant_A	1.5737×10^{-20}	1.0000×10^{0}	Y
cfd2	6.6582×10^{-9}	1.0000×10^{0}	N
e20r5000	3.2526×10^{-19}	9.3981×10^{1}	N
e40r5000	8.5665×10^{-20}	4.1058×10^{1}	N
pwtk	1.2682×10^{-32}	1.0000×10^{0}	Y
shipsec1	9.7502×10^{-22}	1.0000×10^{0}	Y
xenon2	2.3895×10^{-7}	1.0000×10^{0}	Y

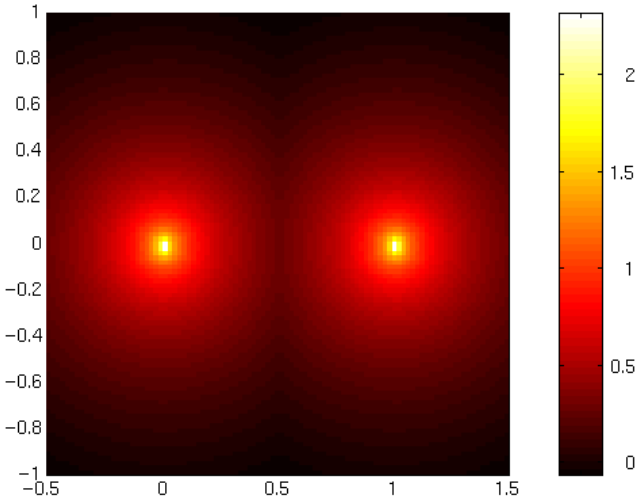
Table B.5: For the matrices from applications described in this section: the minimum and maximum nonzero elements in absolute value of each matrix after equilibration, if applicable. "Equilibrated?" is "Y" if equilibration (whether symmetric or nonsymmetric) was done, else "N." Only xenon2 (which is nonsymmetric) was equilibrated nonsymmetrically.

Matrix name	Dimensions	# nonzeros	$\kappa_2(A)$
WATT1	1856×1856	11360	4.3596×10^{9}
WATT2	1856×1856	11550	1.3628×10^{11}

Table B.6: Properties of the two matrices in the WATT set, in the Harwell-Boeing collection of sparse matrices. Although WATT1 and WATT2 are both sparse, they are small enough that their 2-norm condition numbers may be computed using the dense SVD.



(a) Nonzero structure ("spyplot") of the WATT1 matrix.



(b) Spectral portrait of the WATT1 matrix. The image illustrates the ϵ-pseudospectrum of WATT1, as a function of ϵ (see below for explanation).

Figure B.1: These two figures were copied without permission from the Matrix Market [35] interactive collection of test matrices. The *ϵ-pseudospectrum* of a matrix A is the set $\Lambda_\epsilon = \{z \in \mathbb{C} : z \text{ is an eigenvalue of } A + E \text{ with } \|E\|_2 \leq \epsilon \|A\|_2\}$. The spectral portrait shows the mapping $z \mapsto \log_{10} \|(A - zI)^{-1}\|_2 \|A\|_2$ as a heat plot on a subset of the complex plane. The color scale shows the inverse base-10 logarithm of the perturbation ϵ, so for example 2 indicates $\epsilon = 10^{-2}$. The region enclosed within a particular ϵ contour on the plot indicates the set of all eigenvalues of all matrices $A + E$, where $\|E\|_2 \leq \epsilon \|A\|_2$.

s-step basis length s smaller than the restart length $r = s \cdot t$. For details on our CA-GMRES experiments, see Section 3.5.

The WATT1 matrix belongs to the WATT collection of real nonsymmetric sparse matrices from petroleum engineering applications. Some properties of the two matrices in this collection, including their dimensions and condition numbers, are shown in Table B.6. The WATT matrices belong to the Harwell-Boeing test suite [90], to which they were added in December 1983. The submitter noted that "[t]he values of the coefficients can vary by as much as 10^{14} and the use of automatic scaling routines were giving singular matrices. Fill-in was also higher than expected." We show the nonzero structure of WATT1 in Figure B.1a.

The WATT matrices are small (by current sparse matrix standards), so they can be analyzed accurately and in detail using dense factorizations, yet they are nonsymmetric and ill-conditioned. They also come from applications, rather than being contrived examples. This makes them valuable as test problems for iterative methods. Most importantly for our purposes, their largest eigenvalues are densely clustered. For example, Matlab's dense eigensolver finds that 1 is the largest eigenvalue of WATT1, the first matrix in the WATT set, and it occurs 128 times. We can see this and more from the spectral portrait of WATT1, as provided by the Matrix Market [35] interactive collection of test matrices, and shown here in Figure B.1b. The *spectral portrait* is a heat plot of the ϵ-pseudospectrum of the matrix. It is

explained further in the caption. The spectral portrait illustrates both that the eigenvalues of WATT1 have two clusters in the complex plane (one around 1, and one around 0), and that these eigenvalues of the matrix can be located accurately within the cluster.

The dense clustering of the eigenvalues of WATT1, and the clusters' relative insensitivity to perturbations, explain the observation that a small number of Arnoldi iterations on these matrices produce a set of Ritz values with repetitions or near-repetitions. This is of specific interest to s-step methods, especially with the Newton basis. Repeated shifts require special handling when computing their Leja or modified Leja ordering for the Newton basis. Also, nearly identical shifts may cause underflow when computing this ordering. One way to prevent underflow is to perturb the shifts randomly, which we found does prevent underflow in the case of the WATT1 matrix. (See Section 7.3.2 for details on computing the Leja ordering for the Newton basis, and how we prevent possible underflow when doing so.) Finding matrices from applications with these properties justifies the effort spent getting that special handling correct.

Erhel [94] used the first matrix WATT1 in the WATT set as a test problem for her implementation of the Newton-basis s-step GMRES of Bai et al. [19]. (She implements the same Newton-basis GMRES algorithm, but uses a different, though equally accurate, QR factorization in order to improve performance.) Her paper includes plots of the 2-norm of the residual error at each iteration of the method, for standard GMRES (with restart length s), monomial-basis s-step GMRES, and Newton-basis s-step GMRES. Thus, we can easily compare the convergence of the algorithm she implements with the convergence of our CA-GMRES algorithm.

When solving linear systems $Ax = b$ with the WATT1 matrix, we choose the exact solution x and right-hand side b according to the procedure described in Chapter B, as we do for all the other numerical experiments in this thesis.

Appendix C

Derivations

This appendix contains algebraic derivations of various algorithms and formulae that appear throughout this thesis. We put derivations in this appendix that we think would interrupt the flow if left in the main text, for one or more of the following reasons:

- They only summarize known results, without presenting new results.
- They give so much detail that most readers may find them excessively tedious, yet they may be of interest to some readers.

Section C.1 gives a full derivation of CG3, the three-term recurrence variant of CG. Section C.2 provides a simple example of how Block Gram-Schmidt with reorthogonalization by blocks is not sufficient to ensure orthogonality of the computed vectors in finite-precision arithmetic, even when the block orthogonalization phase is accurate to machine precision and the QR factorization of the current block is performed in exact arithmetic.

C.1 Detailed derivation of CG3

In this section, we derive CG3 – the three-term recurrence variant of CG. We derive it in extreme detail, and this section contains no new results. We first showed CG3 as Algorithm 32 in Section 5.4. We show it again here as Algorithm 41, with a slightly different notation to facilitate our explanations in this section.

We learned CG3 from Saad [208, Algorithm 6.19, p. 192]. The algorithm shown there contains a small error: Line 5 of the algorithm should read

$$x_{j+1} = \rho_j(x_j + \gamma_j r_j) + (1 - \rho_j)x_{j-1}$$

instead of

$$x_{j+1} = \rho_j(x_j - \gamma_j r_j) + (1 - \rho_j)x_{j-1}.$$

We also index our algorithm differently than he does: we use k as the index here instead of j, and we start with $k = 1$ instead of $j = 0$ (which we do in order to avoid negative indices).

CG3 is mathematically equivalent to CG. In particular, the residual vectors r_k in CG3 are equal (in exact arithmetic) to their corresponding residual vectors in CG. CG3 computes and stores only residual vectors and solution vectors, unlike standard CG, which stores these

Algorithm 41 CG3 (3-term recurrence variant of CG)

Input: $Ax = b$: $n \times n$ SPD linear system
Input: x_1: initial guess
Output: Approximate solution x_j of $Ax = b$

1: $x_0 := 0$, $r_0 := 0$
2: $r_1 := b - Ax_1$ ▷ Initial residual
3: **for** $k = 1, 2, \ldots,$ until convergence **do**
4: Compute $w_k := Ar_k$
5: Compute $\langle r_k, r_k \rangle$ and $\langle Ar_k, r_k \rangle$. If either or both are zero, the algorithm has broken down.
6: $\gamma_k := \frac{\langle r_k, r_k \rangle}{\langle Ar_k, r_k \rangle}$
7: **if** $j = 1$ **then**
8: $\rho_k := 1$
9: **else**
10: $\rho_k := \left(1 - \frac{\gamma_k}{\gamma_{k-1}} \cdot \frac{\langle r_k, r_k \rangle}{\langle r_{k-1}, r_{k-1} \rangle} \cdot \frac{1}{\rho_{k-1}}\right)^{-1}$
11: **end if**
12: $x_{k+1} := \rho_k(x_k + \gamma_k r_k) + (1 - \rho_k)x_{k-1}$
13: $r_{k+1} := \rho_k(r_k - \gamma_k w_k) + (1 - \rho_k)r_{k-1}$
14: **end for**

as well as A-conjugate vectors p_k. This makes CG3 attractive for adaptation into an s-step method, since standard CG has data dependencies between the r_k residual vectors and the p_k A-conjugate vectors, whereas CG3 has no such data dependencies.

CG3 can be derived using the following three observations, which apply in exact arithmetic:

1. The residual vectors r_k in CG (and CG3) are nonzero scalar multiples of the symmetric Lanczos basis vectors q_k. (See symmetric Lanczos, Algorithm 24 in Section 4.1.)

2. As a result, the residual vectors r_k must satisfy a three-term recurrence, as the basis vectors q_k in symmetric Lanczos do.

3. Also as a result, the residual vectors r_k in CG (and CG3) are orthogonal (in exact arithmetic): $\langle r_j, r_i \rangle = 0$ if $i \neq j$, otherwise it is nonzero.

We now show this derivation here. We follow the pattern of Saad's derivation [208, pp. 190–192] in this section, with one correction and a few additional explanations. In particular, we take care to show that the coefficients γ_k and ρ_k are both always real and always defined in exact arithmetic, as long as CG has not broken down (i.e., as long as $\{r_1, r_2, \ldots, r_k\}$ continues to be a basis of the Krylov subspace $\mathcal{K}_k(A, r_1)$).

We begin the derivation in Section C.1.1 with the three-term recurrence for the residual vectors r_k (Line 13 of Algorithm 41). We continue in Section C.1.2 with the three-term recurrence for the solution vectors x_k (Line 12 of Algorithm 41). Sections C.1.3 and C.1.4 show how the formulae for the γ_k resp. ρ_k coefficients in CG3 are derived. In those two sections, we show that γ_k and ρ_k are defined for all $k = 1, 2, 3, \ldots$, as long as CG3 itself does not break down.

C.1.1 Three-term r_k recurrence

The residual vectors r_k in CG3 satisfy a three-term recurrence. We may write this in terms of vectors and matrices:

$$r_{k+1} = \rho_k \left(r_k - \gamma_k A r_k \right) + \delta_k r_{k-1}, \tag{C.1}$$

where ρ_k, γ_k, and δ_k are scalar coefficients whose values we will determine for all k. Note that as long as $\{r_1, r_2, \ldots, r_{k+1}\}$ is a basis for the Krylov subspace

$$\mathcal{K}_{k+1}(A, r_1) = \text{span}\{r_1, Ar_1, A^2 r_1, \ldots, A^k r_1\},$$

then both ρ_k and γ_k must be nonzero. We will use this fact many times below. We may also write the three-term recurrence as a polynomial recurrence, in which we use t for the placeholder variable in the polynomials:

$$r_{k+1}(t) = \rho_k \left(r_k(t) - \gamma_k t \cdot r_k(t) \right) + \delta_k r_{k-1}(t). \tag{C.2}$$

In Equation (C.2), $r_k(t)$ is an overloaded notation referring to the residual vector r_k as a polynomial with scalar coefficients μ_j in a monomial polynomial basis:

$$r_k(t) = \sum_{j=0}^{k-1} \mu_j t^j \cdot r_1(t). \tag{C.3}$$

We will not refer to the μ_j coefficients further, but we will refer to this mapping implicitly in the arguments that follow. The mapping $r_k(t) \mapsto r_k$ (from polynomial to vector) is accomplished by replacing t^j in Equation (C.3) with the matrix A^j, and replacing $r_1(t)$ with the initial vector r_1. The initial residual polynomial $r_1(t)$ maps to the initial residual $r_1 = b - Ax_1$, which is a function of A; thus, $r_1(0)$ maps to the vector $b - 0x_1 = b$. We can see from this that $r_k(0)$ for $k = 1, 2, \ldots$ is some nonzero constant C (as long as b is nonzero).

We can already simplify the three-term recurrence for the residual vectors using the observation that for $k = 1, 2, \ldots$, $r_k(t) = C$, a nonzero constant (as long as $b \neq 0$). Thus, setting $t = 0$ in Equation (C.2), we obtain

$$C = \rho_k \left(C - \gamma_k 0 \cdot C \right) + \delta_k C,$$

and thus $\delta_k = 1 - \rho_k$. We see also from this argument that the value of the constant C does not matter, as long as it is nonzero, so we choose $C = 1$.

We now may state the three-term recurrence in its familiar form as Equation (C.4):

$$r_{k+1} = \rho_k \left(r_k - \gamma_k A r_k \right) + (1 - \rho_k) r_{k-1}. \tag{C.4}$$

C.1.2 Three-term x_k recurrence

The approximate solution vectors x_k in CG3 also satisfy a three-term recurrence. We can see this using an argument of Saad [208, p. 191]. Saad begins with the polynomial version of the three-term recurrence for the residual vectors:

$$r_{k+1}(t) = \rho_k \left(r_k(t) - \gamma_k t \cdot r_k(t) \right) + (1 - \rho_k) r_{k-1}(t).$$

Suppose we also express the solution vectors x_k in polynomial form as $x_k(t)$. (Formally, this also happens via the mapping expressed in Equation (C.3).) Since $r_{k+1} = b - Ax_{k+1}$ in vector form, $r_{k+1}(t) = 1 - tx_{k+1}(t)$ in polynomial form. (Recall that b in vector form maps to $r_1(0) = 1$ in polynomial form. Saad then solves this polynomial equation for $x_{k+1}(t)$ to obtain

$$x_{k+1}(t) = \frac{1 - r_{k+1}(t)}{t}. \tag{C.5}$$

Dividing through by t in polynomial form is equivalent to multiplying both sides of the vector form on the left by A^{-1}. The matrix A is invertible because we are running a form of CG, which assumes that A is SPD.

We may now substitute into Equation (C.5) the recursion formula for the residual polynomials $r_{k+1}(t)$:

$$\begin{aligned}
x_{k+1}(t) &= \frac{1 - r_{k+1}(t)}{t} \\
&= t^{-1}\left(1 - \rho_k(r_k(t) - \gamma_k t \cdot r_k(t)) - (1 - \rho_k)r_{k-1}(t)\right) \\
&= t^{-1}\left(1 - \rho_k\left(1 - tx_k(t) - \gamma_k t \cdot r_k(t)\right) - (1 - \rho_k)(1 - tx_{k-1}(t))\right) \\
&= t^{-1}\left((1 - \rho_k) + \rho_k t(x_k(t) + \gamma_k r_k(t)) - (1 - \rho_k) + (1 - \rho_k)tx_{k-1}(t)\right) \\
&= \rho_k(x_k(t) + \gamma_k r_k(t)) + (1 - \rho_k)x_{k-1}(t).
\end{aligned}$$

The last line of the above derivation translates directly into a vector recurrence for the approximate solution x_k:

$$x_{k+1} = \rho_k(x_k + \gamma_k r_k) + (1 - \rho_k)x_{k-1}. \tag{C.6}$$

Note the difference in sign between this recurrence and the three-term recurrence for the residual vectors, Equation (C.4).

C.1.3 A formula for γ_k

We now derive a formula for the γ_k coefficient in CG3. The γ_k coefficient is defined for $k = 1, 2, \ldots$, as long as CG3 has not broken down. We begin with the newly simplified recurrence for the residual vectors, Equation (C.4):

$$r_{k+1} = \rho_k\left(r_k - \gamma_k A r_k\right) + (1 - \rho_k)r_{k-1},$$

and multiply both sides on the left by r_k^*. Doing so and applying the orthogonality of the residual vectors r_{k-1}, r_k, and r_{k+1} we obtain:

$$\begin{aligned}
r_k^* r_{k+1} &= \rho_k\left(r_k^* r_k - \gamma_k r_k^* A r_k\right) + (1 - \rho_k)r_k^* r_{k-1}, \\
0 &= \rho_k\left(\langle r_k, r_k\rangle - \gamma_k\langle Ar_k, r_k\rangle\right) + (1 - \rho_k)\cdot 0, \\
\rho_k\gamma_k\langle Ar_k, r_k\rangle &= \rho_k\langle r_k, r_k\rangle.
\end{aligned}$$

Since $\rho_k \neq 0$ (unless CG has reached a lucky breakdown), we obtain the following formula for γ_k:

$$\gamma_k = \frac{\langle r_k, r_k\rangle}{\langle Ar_k, r_k\rangle}. \tag{C.7}$$

Note that if there is a lucky breakdown and $Ar_k \in \text{span}\{r_1, r_2, \ldots, r_k\}$, then there are two possibilities:

1. Ar_k has no components in the r_k direction, in which case $\langle Ar_k, r_k \rangle = 0$. Therefore, since A is SPD, $r_k = 0$ as well.

2. Ar_k has some component ν_k in the r_k direction, in which case $\langle Ar_k, r_k \rangle = \nu_k > 0$.

We will discuss how to detect breakdown in Section C.1.5.

C.1.4 A formula for ρ_k

We now derive a formula for the ρ_k coefficient in CG3. The ρ_k coefficient is defined for $k = 1, 2, \ldots$, as long as CG3 has not broken down. We begin by showing that $\rho_1 = 1$. We first recall that since the CG3 residual vectors are orthogonal, $\langle r_2, r_1 \rangle = 0$. The formula for x_2 (recall that $x_0 = 0$) in Algorithm 41 is

$$x_2 = \rho_1(x_1 + \gamma_1 r_1).$$

Multiply both sides on the left by $-A$ and add b to both sides:

$$\begin{aligned} b - Ax_2 &= b - \rho_1 Ax_1 - \rho_1\gamma_1 Ar_1, \\ b - Ax_2 &= b - Ax_1 + Ax_1 - \rho_1 Ax_1 - \rho_1\gamma_1 Ar_1, \\ r_2 &= r_1 + (1 - \rho_1)Ax_1 - \rho_1\gamma_1 Ar_1, \end{aligned}$$

multiply both sides on the left by r_1^*:

$$\begin{aligned} r_1^* r_2 &= r_1^* r_1 + (1 - \rho_1) r_1^* Ax_1 - \rho_1\gamma_1 r_1^* Ar_1, \\ 0 &= \langle r_1, r_1 \rangle + (1 - \rho_1)\langle Ax_1, r_1 \rangle - \rho_1\gamma_1 \langle Ar_1, r_1 \rangle, \\ 0 &= \langle r_1, r_1 \rangle + (1 - \rho_1)\langle Ax_1, r_1 \rangle - \rho_1\gamma_1 \langle Ar_1, r_1 \rangle, \end{aligned}$$

and apply the formula $\gamma_1 = \langle r_1, r_1 \rangle / \langle Ar_1, r_1 \rangle$:

$$\begin{aligned} 0 &= \langle r_1, r_1 \rangle + (1 - \rho_1)\langle Ax_1, r_1 \rangle - \rho_1 \langle r_1, r_1 \rangle, \\ 0 &= (1 - \rho_1)\langle r_1, r_1 \rangle + (1 - \rho_1)\langle Ax_1, r_1 \rangle. \end{aligned} \tag{C.8}$$

Since x_1 is the starting guess and may be arbitrary, and since we assume A is nonsingular, Ax_1 may be arbitrary and independent of r_1. In order for Equation (C.8) to hold for any starting guess x_1 and any right-hand side b, it must be the case that $\rho_1 = 1$.

We now derive a recursion formula for ρ_k with $k = 2, 3, \ldots$. We begin with the recurrence for the residual vectors:

$$r_k = \rho_{k-1}\left(r_{k-1} - \gamma_{k-1} Ar_{k-1}\right) + (1 - \rho_{k-1}) r_{k-2}.$$

When $k = 2$, $r_{k-2} = r_0 = 0$. Regardless, $\rho_{k-1} = \rho_1 = 1$ and therefore the r_0 term disappears anyway. Thus, we do not need to handle the case $k = 2$ separately; it is allowed in the

arguments below. If we multiply both sides of the above equation on the left by r_k^*, we obtain the following:

$$\begin{aligned}
r_k &= \rho_{k-1}\left(r_{k-1} - \gamma_{k-1}Ar_{k-1}\right) + (1-\rho_{k-1})r_{k-2},\\
r_k^* r_k &= \rho_{k-1}\left(r_k^* r_{k-1} - \gamma_{k-1}r_k^* Ar_{k-1}\right) + (1-\rho_{k-1})r_k^* r_{k-2},\\
\langle r_k, r_k\rangle &= \rho_{k-1}\left(0 - \gamma_{k-1}\langle Ar_{k-1}, r_k\rangle\right) + (1-\rho_{k-1})\cdot 0,\\
\langle r_k, r_k\rangle &= -\rho_{k-1}\gamma_{k-1}\langle Ar_{k-1}, r_k\rangle.
\end{aligned}$$

Since the terms $\langle r_k, r_k\rangle$, ρ_{k-1}, and γ_{k-1} are all nonzero, we may simplify this further to obtain Equation (C.9):

$$\langle Ar_{k-1}, r_k\rangle = -\frac{\langle r_k, r_k\rangle}{\rho_{k-1}\gamma_{k-1}}. \tag{C.9}$$

Now we use the residual vector recurrence for r_{k+1}:

$$r_{k+1} = \rho_k\left(r_k - \gamma_k Ar_k\right) + (1-\rho_k)r_{k-1},$$

and multiply both sides of it on the left by r_{k-1}^*, obtaining:

$$\begin{aligned}
r_{k+1} &= \rho_k\left(r_k - \gamma_k Ar_k\right) + (1-\rho_k)r_{k-1},\\
r_{k-1}^* r_{k+1} &= \rho_k\left(r_{k-1}^* r_k - \gamma_k r_{k-1}^* Ar_k\right) + (1-\rho_k)r_{k-1}^* r_{k-1},\\
0 &= \rho_k\left(0 - \gamma_k\langle Ar_k, r_{k-1}\rangle\right) + (1-\rho_k)\langle r_{k-1}, r_{k-1}\rangle,\\
0 &= -\rho_k\gamma_k\langle Ar_k, r_{k-1}\rangle + (1-\rho_k)\langle r_{k-1}, r_{k-1}\rangle,\\
\rho_k\gamma_k\langle Ar_k, r_{k-1}\rangle &= (1-\rho_k)\langle r_{k-1}, r_{k-1}\rangle.
\end{aligned}$$

Since the terms $\langle r_{k-1}, r_{k-1}\rangle$, ρ_k, and γ_k are all nonzero, we may simplify this further to obtain Equation (C.10):

$$\langle Ar_k, r_{k-1}\rangle = \frac{1-\rho_k}{\rho_k}\cdot\frac{\langle r_{k-1}, r_{k-1}\rangle}{\gamma_k}. \tag{C.10}$$

The left-hand side of Equation (C.10) is almost, but not quite the same as the left-hand side of Equation (C.9). In real arithmetic, they are equal, but in complex arithmetic, they are complex conjugates. We will assume the latter for maximum generality:

$$\langle Ar_{k-1}, r_k\rangle = \overline{\langle Ar_{k-1}, r_k\rangle}$$

and thus, if we take conjugates of both sides of Equation (C.9), we obtain Equation (C.11):

$$\begin{aligned}
\langle Ar_k, r_{k-1}\rangle &= -\overline{\left(\frac{\langle r_k, r_k\rangle}{\rho_{k-1}\gamma_{k-1}}\right)}\\
&= -\frac{\langle r_k, r_k\rangle}{\overline{\rho}_{k-1}\gamma_{k-1}}.
\end{aligned} \tag{C.11}$$

The simplification on the second line of Equation (C.11) follows from $\langle r_k, r_k\rangle$ and γ_{k-1} both being real. Now we may eliminate the $\langle Ar_k, r_{k-1}\rangle$ term in Equations (C.10) and (C.11), by setting their right-hand sides equal to each other:

$$\begin{aligned}
-\frac{\langle r_k, r_k\rangle}{\overline{\rho}_{k-1}\gamma_{k-1}} &= \frac{1-\rho_k}{\rho_k}\cdot\frac{\langle r_{k-1}, r_{k-1}\rangle}{\gamma_k},\\
-\frac{\langle r_k, r_k\rangle}{\langle r_{k-1}, r_{k-1}\rangle}\cdot\frac{\gamma_k}{\gamma_{k-1}}\cdot\frac{1}{\overline{\rho}_{k-1}} &= \frac{1-\rho_k}{\rho_k}.
\end{aligned}$$

Given that $\rho_k \neq 0$ (as long as symmetric Lanczos has not broken down), we conclude that

$$\rho_k = \left(1 - \frac{\langle r_k, r_k \rangle}{\langle r_{k-1}, r_{k-1} \rangle} \cdot \frac{\gamma_k}{\gamma_{k-1}} \cdot \frac{1}{\overline{\rho}_{k-1}}\right)^{-1} \tag{C.12}$$

for $k > 1$. The inverse on the right-hand side is possible by observing that if

$$\frac{1-y}{y} = -x$$

and $y \neq 0$, then $x \neq 1$ (otherwise $1 - y = -y$ and $1 = 0$, which is impossible). Thus

$$y = \frac{1}{1-x}.$$

We can remove the conjugation $\overline{\rho_{k-1}}$ in Equation (C.12) using induction. By definition, $\rho_1 = 1$, which is real. Also, we know that all of the terms in Equation (C.12) are real, except perhaps for ρ_k and $\overline{\rho_{k-1}}$. If we take complex conjugates of both sides, and inductively assume that ρ_{k-1} is real, then $\overline{\rho_k}$ must be real and therefore ρ_k is real for all k. This gives us the final formula for ρ_k:

$$\rho_k = \begin{cases} 1 & \text{for } k = 1, \\ \left(1 - \frac{\langle r_k, r_k \rangle}{\langle r_{k-1}, r_{k-1} \rangle} \cdot \frac{\gamma_k}{\gamma_{k-1}} \cdot \frac{1}{\rho_{k-1}}\right)^{-1} & \text{for } k = 2, 3, \ldots. \end{cases} \tag{C.13}$$

C.1.5 Detecting breakdown

We mentioned at the end of Section C.1.3 that if there is a lucky breakdown and $Ar_k \in \text{span}\{r_1, r_2, \ldots, r_k\}$, then there are two possibilities for the value of $\langle Ar_k, r_k \rangle$:

1. Ar_k has no components in the r_k direction, in which case $\langle Ar_k, r_k \rangle = 0$. Therefore, since A is SPD, $r_k = 0$ as well.

2. Ar_k has some component ν_k in the r_k direction, in which case $\langle Ar_k, r_k \rangle = \nu_k > 0$.

The first case makes it easy to detect breakdown before computing γ_k. However, the second case makes breakdown harder to detect. If γ_k may be well defined and nonzero even when $Ar_k \in \text{span}\{r_1, r_2, \ldots, r_k\}$, then how do we detect the breakdown before trying to compute a nonsensical value of ρ_k?

In exact arithmetic, if CG3 first breaks down in iteration k, the residual r_{k+1} should be exactly zero. If we substitute this into the three-term residual recurrence, we obtain

$$\begin{aligned} 0 &= \rho_k(r_k - \gamma_k A r_k) + (1 - \rho_k) r_{k-1}, \\ \rho_k \gamma_k A r_k &= \rho_k r_k + (1 - \rho_k) r_{k-1}. \end{aligned} \tag{C.14}$$

The case $\rho_k = 0$ is impossible since that would make $r_{k-1} = 0$, but CG3 did not break down in that iteration so $r_{k-1} \neq 0$. We see, therefore, that we have nothing to worry about; ρ_k is well defined, and we will detect at the next iteration $k + 1$ that r_{k+1} is zero (in exact arithmetic; in finite-precision arithmetic it will have very small norm). We will compute $\langle r_{k+1}, r_{k+1} \rangle$ in the next iteration anyway, so we will detect it then. Such detection takes place in Line 5 of Algorithm 41.

C.2 Naïve Block Gram-Schmidt may fail

In Section 2.4.7, we referred to an observation by Stewart [217], in which he points out how Block Gram-Schmidt with reorthogonalization by blocks is not sufficient to ensure orthogonality of the computed vectors in finite-precision arithmetic, even when the block orthogonalization phase is accurate to machine precision and the QR factorization of the current block is performed in exact arithmetic. The following simple example illustrates this phenomenon. Suppose we want to orthogonalize the columns of the matrix

$$V = [V_1, V_2, \ldots, V_k, \ldots, V_M],$$

where each V_k is a block of vectors. We have already orthogonalized blocks V_1, ..., V_{k-1}, and thus computed the corresponding blocks Q_1, ..., Q_{k-1}. Say also we have already computed the orthogonal projection

$$Y_k = \left(I - [Q_1, \ldots, Q_{k-1}][Q_1, \ldots, Q_{k-1}]^*\right) V_k.$$

We skip any possible block reorthogonalization step for simplicity. Suppose that Y_k, and thus V_k, have two columns: $Y_k = [y_1, y_2]$. In addition, suppose that $y_1 \perp \mathcal{R}\left([Q_1, \ldots, Q_{k-1}]\right)$ already, and that $y_2 = y_1 + \epsilon w$, where ϵ is a tiny scalar and $\|w\|_2 = 1$. The "noise term" ϵw may be used to represent in a simplified way the effects of block reorthogonalization, which guarantee orthogonality to machine precision. Alternately, y_1 and y_2 may represent two vectors that are nearly dependent in exact arithmetic. The effects will be the same, regardless of the cause.

If we now compute the thin QR factorization of Y_k *in exact arithmetic* (not considering the effects of rounding error), we obtain $Y_k = Q_k R_{kk}$ with

$$R_{kk} = \begin{pmatrix} \|y_1\|_2 & \|y_1\|_2 + \epsilon\theta \\ 0 & |\epsilon| \left\| w - \frac{\theta}{\|y_1\|_2} y_1 \right\|_2 \end{pmatrix}, \tag{C.15}$$

where $\theta = \langle w, y_1/\|y_1\|_2 \rangle$ is a scalar in $[-1, 1]$ (because it is the inner product of two unit-length vectors), and the Q factor is given by

$$\begin{aligned} Q_k &= [q_1, q_2] \\ &= \left[\|y_1\|_2^{-1} y_1,\ R_{kk}(2,2)^{-1} \epsilon \left(w - \theta \|y_1\|_2^{-1} y_1 \right) \right]. \end{aligned} \tag{C.16}$$

We may further simplify q_2 as follows:

$$q_2 = \operatorname{sign}(\epsilon) \left\| w - \frac{\theta}{\|y_1\|_2} y_1 \right\|_2^{-1} \cdot \left(w - \frac{\theta}{\|y_1\|_2} y_1 \right), \tag{C.17}$$

Depending on the direction of the unit-length noise term w, q_2 may consist mainly of noise. Furthermore, that noise may not be at all orthogonal to $\mathcal{R}\left([Q_1, \ldots, Q_{k-1}]\right)$, and reorthogonalizing Q_k against those blocks will only promote the noise term in q_2. Note that this problem does not come from the particular orthogonalization procedure, since all QR factorizations result in the same Q factor, up to a unitary column scaling (see e.g., Golub and Van Loan [114]).

Appendix D

Definitions

This appendix contains definitions and discussions of terms that appear throughout this thesis. We put definitions in this appendix that we think would interrupt the flow if left in the main text, for one or more of the following reasons:

- They only summarize known results, without presenting new results.
- They give so much detail that most readers may find them excessively tedious, yet they may be of interest to some readers.

Section D.1 defines the *departure from normality* of a square matrix A, which is useful for understanding the numerical behavior of s-step bases. Section D.2 defines the *capacity* of a subset of the complex plane satisfying certain criteria. We refer to the capacity and use estimates of the capacity in Section 7.3.2, when computing the Leja or modified Leja ordering of shifts for the Newton s-step basis.

D.1 Departure from normality

The "departure from normality" of a square matrix A measures the "nonnormality" of A. The matrix A is *normal* when it commutes with its adjoint, i.e., when $A^*A = AA^*$. Otherwise, A is *nonnormal*. The *departure from normality* $\Delta(A)$ of the matrix A is the Frobenius norm of the strict upper triangle (not including the diagonal) of the upper triangular Schur factor of A. That is, if $A = UTU^*$ is the Schur decomposition of A, with U unitary and T upper triangular, and $T = D + M$ with D diagonal (its entries being the eigenvalues of A) and M upper triangular with a zero diagonal), then

$$\Delta(A)^2 \equiv \|M\|_F^2 = \|T\|_F^2 - \|D\|_F^2. \tag{D.1}$$

Henrici [127] defines the above quantity, and points out that while the off-diagonal elements of the Schur triangular factor are not unique, the departure from normality itself is unique in exact arithmetic. It is also within a factor of $\sqrt{n}$ (where the matrix A is $n \times n$) of the distance from A to the nearest normal matrix (Higham and Knight [134, p. 347]):

$$\frac{\Delta(A)}{\sqrt{n}} \leq \min_E \{\|E\|_F : A + E \text{ is normal}\} \leq \Delta(A). \tag{D.2}$$

The departure from normality of a matrix A is useful for at least two reasons, relevant to the content of this thesis. First, it expresses the sensitivity of the eigenvalues of A to additive perturbations; see e.g., Wilkinson [240, pp. 167–8], quoting a result by Henrici [127] (see also Golub and Van Loan [114, p. 312]). All the s-step bases we present in this thesis, except for the monomial basis, require estimates of the spectrum of A. Without foreknowledge of the spectrum, these estimates are taken from the Ritz values computed using the Krylov method itself. The Newton and Chebyshev s-step bases are able to keep the condition number of the basis small, by constructing the bases using polynomials that are known to be bounded or to have slow growth as a function of s on the approximate spectrum. Outside of the approximate spectrum, the polynomials may grow rapidly. Thus, if it is hard to approximate the eigenvalues of the matrix A, the s-step bases may not be accurate.

The departure from normality also affects the relationship between the *spectral radius* λ_A of A (the maximum eigenvalue of A in absolute value), and the 2-norm $\|A^r\|_2$ of a positive integer power r of A. For normal matrices, $\|A^r\|_2 = \lambda_A^r$; for nonnormal matrices, this is not necessarily true. Henrici [127, Theorem 2] gives an upper bound on $\|A^r\|_2$, which shows that $\|A^r\|_2$ can grow much faster than λ_A^r, depending on the departure from normality $\Delta(A)$. This relates directly to the convergence of the power method in finite-precision arithmetic, as discussed by Higham and Knight [134]. They find that if the matrix A is sufficiently nonnormal, even if $\lambda_A < 1$, repeated powers of the matrix may not converge to zero. For the s-step monomial basis, of course, the intention is that the basis vectors *not* converge as a function of s, or at least converge slowly. This could explain what we observed, for example, with the test problems of Bai et al. [19] in Section 3.5.3, namely that CA-GMRES with the monomial basis sometimes converges better than with the Newton basis for sufficiently nonnormal matrices.

D.2 Capacity

Reichel [202] defines the "capacity" of a subset K of the complex plane $\mathbb{C}$. He uses it as a theoretical tool in order to prove a bound on the growth of the condition number of the Newton basis for polynomial interpolation. The capacity also plays an important role in computing the Leja ordering of points in K (see Section 7.3.2), since running estimates of the capacity can be used to scale the points in order to avoid underflow or overflow in finite-precision arithmetic. The Leja ordering matters to us because it often makes the Newton s-step Krylov basis better conditioned than the monomial s-step Krylov basis.

We paraphrase Reichel's [202, p. 332] definition of the capacity here. Let $K \subset \mathbb{C}$ be compact and possibly infinite. Identify $\mathbb{C}$ with $\mathbb{R}^2$ in the usual way: $z = x + \imath y \mapsto (x, y)$. Suppose also that the complement $(\mathbb{C} \cup \{\infty\}) \setminus K$ of K is connected and has a Green's function $G(x, y)$ with a pole at infinity. The following three requirements uniquely determine $G(x, y)$:

1. $\Delta G(x, y) = 0$ in $\mathbb{C} \setminus K$ (it satisfies Laplace's equation)
2. $G(x, y) = 0$ on the boundary ∂K of K (Laplace's equation for G has Dirichlet boundary conditions)
3. $\frac{1}{2\pi} \int_{\partial K} \frac{\partial G}{\partial n}(x, y)\, ds = 1$ (where $\frac{\partial}{\partial n}$ denotes the normal derivative directed into $\mathbb{C} \setminus K$)

The *capacity* c of the set K is a nonnegative constant defined by

$$c \equiv \lim_{|z|\to\infty} |z| \exp(-G(x,y)). \tag{D.3}$$

For example, the circle of radius r in the complex plane has capacity r. If K has capacity c, and $\alpha > 0$, then the set $\alpha K = \{\alpha z : z \in K\}$ has capacity αc. Thus, the capacity of K relates to its scaling. Estimating the capacity c of K and working with $c^{-1}K$ helps avoid underflow or overflow when computing the Leja or modified Leja ordering of points in K, as we explain in Section 7.3.2.

Bibliography

[1] N. N. Abdelmalek, *Round off error analysis for Gram–Schmidt method and solution of linear least squares problems*, BIT Numerical Mathematics, 11 (1971), pp. 345–368.

[2] M. Adams, *Multigrid equation solvers for large scale nonlinear finite element simulations*, Tech. Rep. UCB/CSD-99-1033, EECS Department, University of California, Berkeley, Jan 1999.

[3] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, *LogGP: Incorporating long messages into the LogP model – one step closer towards a realistic model for parallel computation*, in Seventh Annual Symposium on Parallel Algorithms and Architecture (SPAA'95), July 1995.

[4] G. Amdahl, *Validity of the single processor approach to achieving large-scale computing capabilities*, in AFIPS Conference Proceedings, vol. 30, 1967, pp. 483–485.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. W. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, 1999.

[6] ——, *LAPACK – Linear Algebra PACKage*, 2010. Available online at `http://www.netlib.org/lapack/` (last accessed 08 Feb 2010).

[7] M. Andrews, T. Leighton, P. T. Metaxas, and L. Zhang, *Automatic methods for hiding latency in high bandwidth networks (extended abstract)*, in STOC: ACM Symposium on Theory of Computing (STOC), 1996.

[8] W. E. Arnoldi, *The principle of minimized iterations in the solution of the matrix eigenvalue problem*, Q. Appl. Maths, 9 (1951), pp. 17–29.

[9] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, *The Landscape of Parallel Computing Research: A View from Berkeley*, Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[10] K. Asanovic, R. Bodik, J. W. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. A. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. A. Yelick, *A View of the Parallel Computing Landscape*, Communications of the ACM, 52 (2009), pp. 56–67.

[11] S. Ashby, *CHEBYCODE: A Fortran implementation of Manteuffel's adaptive Chebyshev algorithm*, Tech. Rep. UIUC DCS-R-85-1203, University of Illinois Urbana-Champaign, 1985.

[12] J. Backus, *Can programming be liberated from the von Neumann style? a functional style and its algebra of programs*, Communications of the ACM, 21 (1978).

[13] J. Baglama, D. Calvetti, and L. Reichel, *Algorithm 827: irbleigs: A MATLAB program for computing a few eigenpairs of a large sparse Hermitian matrix*, ACM Trans. Math. Softw., 29 (2003), pp. 337–348.

[14] J. Bahi, S. Contasset-Vivier, and R. Couturier, *Parallel Iterative Algorithms: From Sequential to Grid Computing*, Chapman and Hall / CRC, 2007.

[15] Z. Bai and D. Day, *Block Arnoldi method*, in Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, Z. Bai, J. W. Demmel, J. J. Dongarra, A. Ruhe, and H. van der Vorst, eds., Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000, pp. 196–204.

[16] Z. Bai, J. W. Demmel, J. J. Dongarra, A. Ruhe, and H. van der Horst, eds., *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, SIAM, Philadelphia, 2000.

[17] Z. Bai, D. Hu, and L. Reichel, *An implementation of the GMRES method using QR factorization*, in Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing, 1991, pp. 84–91.

[18] ——, *A Newton basis GMRES implementation*, Tech. Rep. Research Report 91-03, Dep't of Mathematics, University of Kentucky, Apr. 1991.

[19] ——, *A Newton basis GMRES implementation*, IMA Journal of Numerical Analysis, 14 (1994), pp. 563–581.

[20] A. H. Baker, J. M. Dennis, and E. R. Jessup, *On improving linear solver performance: A block variant of GMRES*, SIAM J. Sci. Comp., 27 (2006), pp. 1608–1626.

[21] C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq, and H. K. Thornquist, *Anasazi webpage*. `http://trilinos.sandia.gov/packages/anasazi/`.

[22] G. Ballard, J. W. Demmel, O. Holtz, and O. Schwartz, *Minimizing communication in linear algebra*, Tech. Rep. UCB/EECS-2009-62, University of California Berkeley EECS Department, May 2009. Accepted by *SIAM Journal on Matrix Analysis and Applications*.

[23] R. Barrett, M. Berry, T. F. Chan, J. W. Demmel, J. Donato, J. J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Horst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, second ed., 1994.

[24] G. Baudet, *Asynchronous iterative methods for multiprocessors*, J. Assoc. Comput. Mach., 25 (1978), pp. 226–244.

[25] B. Beckermann, *On the numerical condition of polynomial bases: Estimates for the condition number of Vandermonde, Krylov and Hankel matrices.* Habilitationsschrift, Universität Hannover, April 1996.

[26] ——, *The condition number of real Vandermonde, Krylov and positive definite Hankel matrices*, Numerische Mathematik, 85 (2000), pp. 553–577.

[27] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari, *Optimal sparse matrix dense vector multiplication in the I/O-model*, in Proceedings of the Nineteenth Annual ACM symposium on Parallel Algorithms and Architectures (SPAA 2007), San Diego, California, ACM, 2007, pp. 61–70.

[28] M. Bern, J. R. Gilbert, B. Hendrickson, N. Nguyen, and S. Toledo, *Support-graph preconditioners*, SIAM Journal on Matrix Analysis and Applications, 27 (2006), pp. 930–951.

[29] G. Bilardi and F. P. Preparata, *Processor–time tradeoffs under bounded-speed message propagation: Part I, upper bounds*, Theory of Computing Systems, 30 (1997).

[30] ——, *Processor–time tradeoffs under bounded-speed message propagation: Part II, lower bounds*, Theory of Computing Systems, 32 (1999).

[31] R. H. Bisseling and W. Meesen, *Communication balancing in parallel sparse matrix-vector multiplication*, Electronic Transactions on Numerical Analysis, 21 (2005), pp. 47–65.

[32] Å. Björck, *Solving linear least squares problems by Gram-Schmidt orthogonalization*, BIT Numerical Mathematics, 7 (1967), pp. 1–21.

[33] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. W. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, SIAM, Philadelphia, 1997.

[34] O. A. R. Board, *OpenMP Application Program Interface, Version 3.0.* Available online at `http://www.openmp.org/mp-documents/spec30.pdf`. Last accessed 10 March 2010., May 2008.

[35] R. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. J. Dongarra, *Matrix Market: a web resource for test matrix collections*, in The Quality of Numerical Software: Assessment and Enhancement, R. Boisvert, ed., Chapman and Hall, London, 1997, pp. 125–137. The Matrix Market online interactive collection of test problems is available at `http://math.nist.gov/MatrixMarket/index.html`.

[36] S. Börm and L. Grasedyck, *Hlib package.* `http://www.hlib.org/hlib.html`, 2006.

[37] S. Börm, L. Grasedyck, and W. Hackbusch, *Hierarchical matrices.* `http://www.mis.mpg.de/scicomp/Fulltext/WS_HMatrices.pdf`, 2004.

[38] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial*, SIAM, 2000.

[39] S. Brin and L. Page, *The anatomy of a large-scale hypertextual web search engine*, in Seventh International World-Wide Web Conference (WWW 1998), April 14-18, 1998, Brisbane, Australia, April 1998.

[40] R. Bru, L. Elsner, and M. Neumann, *Models of parallel chaotic iteration methods*, Linear Algebra Appl., (1989).

[41] A. M. Bruckstein, D. L. Donoho, and M. Elad, *From sparse solutions of systems of equations to sparse modeling of signals and images*, SIAM Review, 51 (2009), pp. 34–81.

[42] J. R. Bunch, *Equilibration of symmetric matrices in the max-norm*, Journal of the ACM, 18 (1971), pp. 566–572.

[43] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Tech. Rep. UT-CS-07-600, University of Tennessee, Sept. 2007. LAWN #191.

[44] ——, *Parallel tiled QR factorization for multicore architectures*, Tech. Rep. UT-CS-07-598, University of Tennessee, July 2007. LAWN #190.

[45] D. Calvetti, G. H. Golub, and L. Reichel, *An adaptive Chebyshev iterative method for nonsymmetric linear systems based on modified moments*, Numer. Math., 67 (1994), pp. 21–40.

[46] D. Calvetti and L. Reichel, *On the evaluation of polynomial coefficients*, Numerical Algorithms, (2003).

[47] E. Carson, *A communication-avoiding biconjugate gradient algorithm.* Unpublished semester project for the fall 2009 semester of Math 221 at the University of California, Berkeley, Dec 2009.

[48] Ü. Çatalyürek and C. Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Trans. Parallel Dist. Systems, 10 (1999), pp. 673–693.

[49] S. Chandrasekaran, M. Gu, and W. Lyons, *A fast and stable adaptive solver for hierarchically semi-separable representations*, May 2004.

[50] S. Chandrasekaran, M. Gu, and T. Pals, *Fast and stable algorithms for hierarchically semi-separable representations*, 2004.

[51] D. Chazan and W. Miranker, *Chaotic relaxation*, Linear Algebra Appl., 2 (1969), pp. 199–222.

[52] T.-Y. Chen and J. W. Demmel, *Balancing sparse matrices for computing eigenvalues*, Linear Algebra Appl., 309 (2000), pp. 261–287.

[53] E. Chow, *A priori sparsity patterns for parallel sparse approximate inverse preconditioners*, SIAM Journal on Scientific Computing, 21 (2000), pp. 1804–1822.

[54] ——, *Parallel implementation and practical use of sparse approximate inverses with a priori sparsity patterns*, Int. J. High Perf. Comput. Appl., 15 (2001), pp. 56–74.

[55] A. T. Chronopoulos, *s-step iterative methods for (non)symmetric (in)definite linear systems*, SIAM Journal on Numerical Analysis, 28 (1991), pp. 1776–1789.

[56] A. T. Chronopoulos and C. W. Gear, *Implementation of preconditioned s-step conjugate gradient methods on a multiprocessor system with memory hierarchy*, Parallel Comput., 11 (1989), pp. 37–53.

[57] ——, *s-step iterative methods for symmetric linear systems*, J. Comput. Appl. Math., 25 (1989), pp. 153–168.

[58] A. T. Chronopoulos and S. K. Kim, *s-step Orthomin and GMRES implemented on parallel computers*, Tech. Rep. Technical Report No. 90/43R, UMSI, Minneapolis, MN, 1990.

[59] A. T. Chronopoulos and D. Kincaid, *On the Odir iterative method for nonsymmetric indefinite linear systems*, Numerical Linear Algebra with Applications, 8 (2001), pp. 71–82.

[60] A. T. Chronopoulos and C. D. Swanson, *Parallel iterative s-step methods for unsymmetric linear systems*, Parallel Computing, 22 (1996), pp. 623–641.

[61] M. T. Chu and G. H. Golub, *Inverse Eigenvalue Problems: Theory, Algorithms, and Applications*, Oxford University Press, New York, 2005.

[62] C. Click, *A crash course in modern hardware.* Sun 2009 JVM Language Summit, lecture available in video form at `www.infoq.com/presentations/click-crash-course-modern-hardware` (last accessed January 2010), 2009.

[63] E. Cohen, *Estimating the size of the transitive closure in linear time*, in 35th Annual Symposium on Foundations of Computer Science, Santa Fe, NM, November 20–22, 1994, 1994.

[64] J. Cullum and W. E. Donath, *A block Lanczos algorithm for computing the q algebraically largest eigenvalues and a corresponding eigenspace of large, sparse, real symmetric matrices*, in Proc. of 1974 IEEE Conf. on Decision and Control, Phoenix, 1974.

[65] J. K. Cullum and R. A. Willoughby, *Lanczos Algorithms for Large Symmetric Eigenvalue Computations: Volume 1, Theory*, Birkhäuser Boston, 1985.

[66] A. R. Curtis and J. K. Reid, *On the automatic scaling of matrices for Gaussian elimination*, J. Inst. Math. Appl., 10 (1972), pp. 118–124.

[67] R. D. da Cunha, D. Becker, and J. C. Patterson, *New parallel (rank-revealing) QR factorization algorithms*, in Euro-Par 2002. Parallel Processing: Eighth International Euro-Par Conference, Paderborn, Germany, August 27–30, 2002, 2002.

[68] T. A. Davis, *The University of Florida Sparse Matrix Collection*, 2009. Submitted to ACM Transactions on Mathematical Software.

[69] D. Day, *An efficient implementation of the nonsymmetric Lanczos algorithm*, SIAM Journal on Matrix Analysis and Applications, 18 (1997), pp. 566–589.

[70] E. D'Azevedo, V. Eijkhout, and C. Romine, *LAPACK Working Note 56: Reducing communication costs in the conjugate gradient algorithm on distributed memory multiprocessor*, Tech. Rep. CS-93-185, Computer Science Department, University of Tennessee, Knoxville, 1993.

[71] E. de Sturler, *A parallel variant of GMRES(m)*, in Proceedings of the 13th IMACS World Congress on Computation and Applied Mathematics, J. J. H. Miller and R. Vichnevetsky, eds., Dublin, Ireland, 1991, Criterion Press.

[72] ——, *Truncation strategies for optimal Krylov subspace methods*, SIAM Journal on Numerical Analysis, 36 (1999), pp. 864–889.

[73] E. de Sturler and H. A. van der Vorst, *Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers*, Applied Numerical Mathematics, 18 (2005), pp. 441–459.

[74] J. W. Demmel, *Applied Numerical Linear Algebra*, SIAM, Philadelphia, 1997.

[75] J. W. Demmel, L. Grigori, M. Hoemmen, and J. Langou, *Communication-optimal parallel and sequential QR factorizations: theory and practice*, Tech. Rep. UCB/EECS-2008-89, University of California Berkeley, 2008. Also appears as LAPACK Working Note #204 (see `http://www.netlib.org/lapack/lawns/downloads/`).

[76] J. W. Demmel, M. Hoemmen, Y. Hida, and E. J. Riedy, *Non-negative diagonals and high performance on low-profile matrices from Householder QR*, SIAM Journal on Scientific Computing, 31 (2009), pp. 2832–2841.

[77] J. W. Demmel, M. Hoemmen, M. Mohiyuddin, and K. A. Yelick, *Avoiding communication in computing Krylov subspaces*, Tech. Rep. UCB/EECS-2007-123, EECS Department, University of California, Berkeley, Oct 2007.

[78] ——, *Avoiding communication in sparse matrix computations*, in IEEE International Parallel and Distributed Processing Symposium, Apr 2008.

[79] ——, *Minimizing communication in sparse matrix solvers*, in Proceedings of the 2009 ACM/IEEE Conference on Supercomputing (New York, NY, USA), Nov 2009.

[80] W. Deren, *On the convergence of the parallel multisplitting AOR algorithm*, Linear Algebra Appl., 154–156 (1991), pp. 473–486.

[81] K. D. Devine, E. G. Boman, R. Heaphy, R. H. Bisseling, and U. V. Catalyurek, *Parallel hypergraph partitioning for scientific computing*, in Proc. of IPDPS'06, Rhodos, Greece, April 2006.

[82] C. H. Q. Ding and Y. He, *A ghost cell expansion method for reducing communication in solving PDE problems*, in Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (New York, NY, USA), ACM Press, Nov. 2001.

[83] J. J. Dongarra, *Performance of various computers using standard linear equations software*, Tech. Rep. CS-89-85, Computer Science Department, University of Tennessee, Knoxville TN, 37996, November 2009. The most recent version of this report, which is updated every six months, can be found at `http://www.netlib.org/benchmark/performance.ps`.

[84] J. J. Dongarra, J. Bunch, C. Moler, and P. Stewart, *LINPACK home page*, 2010. Available online at `http://www.netlib.org/linpack/` (last accessed 08 Feb 2010).

[85] J. J. Dongarra, J. D. Croz, I. S. Duff, and S. Hammarling, *Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 16 (1990), pp. 18–28.

[86] ——, *A set of Level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 16 (1990), pp. 1–17.

[87] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson, *Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subprograms*, ACM Trans. Math. Soft., 14 (1988), pp. 18–32.

[88] ——, *An extended set of FORTRAN Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 14 (1988), pp. 1–17.

[89] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss, *Cache optimization for structured and unstructured grid multigrid*, Electronic Transactions on Numerical Analysis, (2000).

[90] I. S. Duff, R. Grimes, and J. Lewis, *User's guide for the Harwell-Boeing Sparse Matrix Collection (release i)*, Tech. Rep. RAL-92-086, Rutherford Appleton Laboratory, 1992.

[91] I. S. Duff and S. Pralet, *Strategies for scaling and pivoting for sparse symmetric indefinite problems*, SIAM Journal on Matrix Analysis and Applications, 27 (2005), pp. 313–340.

[92] H. C. Elman, Y. Saad, and P. E. Saylor, *A hybrid Chebyshev Krylov subspace algorithm for solving nonsymmetric systems of linear equations*, SIAM Journal on Scientific and Statistical Computing, 7 (1986), pp. 840–855.

[93] E. Elmroth and F. Gustavson, *Applying recursion to serial and parallel QR factorization leads to better performance*, IBM Journal of Research and Development, 44 (2000), pp. 605–624.

[94] J. Erhel, *A parallel GMRES version for general sparse matrices*, Electronic Transactions on Numerical Analysis, 3 (1995), pp. 160–176.

[95] V. Faber and T. Manteuffel, *Necessary and sufficient conditions for the existence of a conjugate gradient method*, SIAM J. Numer. Anal., 21 (1984), pp. 352–361.

[96] B. Fischer and R. Freund, *Chebyshev polynomials are not always optimal*, Journal of Approximation Theory, 65 (1991), pp. 261–272.

[97] K. Fischer, B. Gärtner, T. Herrmann, M. Hoffmann, and S. Schönherr, *Bounding volumes*, in CGAL User and Reference Manual, CGAL Editorial Board, 3.5 ed., 2009.

[98] B. Fisher and R. Freund, *On adaptive weighted polynomial preconditioning for Hermitian positive definite matrices*, SIAM Journal on Scientific Computing, 15 (1994), pp. 408–426.

[99] R. Fletcher, *Conjugate gradient methods for indefinite systems*, in Proc. Dundee Biennial Conference on Numerical Analysis, G. A. Watson, ed., Berlin, New York, 1975, Springer-Verlag.

[100] G. E. Forsythe, *On the asymptotic directions of the s-dimensional optimum gradient method*, Numerische Mathematik, 11 (1968), pp. 57–76.

[101] S. Fortune and J. Wyllie, *Parallelism in random access machines*, in STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing, New York, NY, USA, 1978, ACM, pp. 114–118.

[102] R. Freund, *A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems*, SIAM Journal on Scientific Computing, 14 (1993), pp. 470–482.

[103] R. Freund and N. Nachtigal, *QMR: A quasi-minimal residual method for non-Hermitian linear systems*, Numer. Math., 60 (1991), pp. 315–339.

[104] R. W. Freund, M. H. Gutknecht, and N. M. Nachtigal, *An implementation of the look-ahead Lanczos algorithm for non-Hermitian matrices*, SIAM Journal on Scientific Computing, 14 (1993), pp. 137–158.

[105] R. W. Freund and M. Malhotra, *A block QMR algorithm for non-Hermitian linear systems with multiple right-hand sides*, Linear Algebra and its Applications, 254 (1997), pp. 119–157. Proceedings of the Fifth Conference of the International Linear Algebra Society (Atlanta, GA, 1995).

[106] A. Frommer and D. B. Szyld, *On asynchronous iterations*, Journal of Computational and Applied Mathematics, 123 (2000), pp. 201–216.

[107] A. Ganapathi, K. Datta, A. Fox, and D. Patterson, *A case for machine learning to optimize multicore performance*, in First USENIX Workshop on Hot Topics in Parallelism, 2009.

[108] W. Gautschi, *The condition of Vandermonde-like matrices involving orthogonal polynomials*, Linear Algebra Appl., 52/53 (1983), pp. 293–300.

[109] ——, *Orthogonal Polynomials: Computation and Approximation*, Oxford University Press, 2004.

[110] W. Gautschi and G. Inglese, *Lower bounds for the condition number of Vandermonde matrices*, Numer. Math., 52 (1988), pp. 241–250.

[111] L. Giraud and J. Langou, *A robust criterion for the Modified Gram-Schmidt algorithm with selective reorthogonalization*, SIAM Journal on Scientific Computing, 25 (2003), pp. vii–765.

[112] L. Giraud, J. Langou, and M. Rozložník, *On the loss of orthogonality in the Gram-Schmidt orthogonalization process*, Computers and Mathematics with Applications, 50 (2005), pp. 1069–1075.

[113] G. H. Golub and W. Kahan, *Calculating the singular values and pseudo-inverse of a matrix*, J. SIAM Numer. Anal., 2 (1965).

[114] G. H. Golub and C. F. V. Loan, *Matrix Computations*, The Johns Hopkins University Press, third ed., 1996.

[115] G. H. Golub, R. J. Plemmons, and A. Sameh, *Parallel block schemes for large-scale least-squares computations*, in High-Speed Computing: Scientific Applications and Algorithm Design, R. B. Wilhelmson, ed., University of Illinois Press, Urbana and Chicago, IL, USA, 1988, pp. 171–179.

[116] G. H. Golub and R. R. Underwood, *The block Lanczos method for computing eigenvalues*, in Mathematical Software III, J. R. Rice, ed., Academic, New York, NY, 1977.

[117] K. Goto and R. A. van de Geijn, *Anatomy of high-performance matrix multiplication*, Transactions on Mathematical Software, 34 (2008).

[118] ——, *Anatomy of high-performance matrix multiplication*, ACM Transactions on Mathematical Software, 34 (2008).

[119] S. L. Graham, M. Snir, and C. A. Patterson, *Getting Up to Speed: The Future of Supercomputing*, National Academies Press, Washington, D.C., USA, 2005.

[120] A. Greenbaum, *Iterative Methods for Solving Linear Systems*, SIAM, Philadelphia, PA, 1997.

[121] A. Greenbaum, M. Rozložník, and Z. Strakoš, *Numerical behavior of the modified Gram-Schmidt GMRES implementation*, BIT Numerical Mathematics, 37 (1997).

[122] K. D. Gremban, G. L. Miller, and M. Zagha, *Performance evaluation of a new parallel preconditioner*, Tech. Rep. CMU-CS-94-205, Carnegie Mellon University, 1994.

[123] L. Grigori, J. W. Demmel, and H. Xiang, *Communication avoiding Gaussian elimination*, in Proceedings of the IEEE/ACM SuperComputing SC08 Conference, November 2008. Also available as INRIA Technical Report 6523.

[124] B. Gunter and R. van de Geijn, *Parallel out-of-core computation and updating of the QR factorization*, ACM Transactions on Mathematical Software, 31 (2005), pp. 60–78.

[125] M. Gutknecht and Z. Strakoš, *Accuracy of two three-term and three two-term recurrences for Krylov space solvers*, SIAM Journal on Matrix Analysis and Applications, 22 (2000), pp. 213–229.

[126] W. Hackbusch, *Hierarchische Matrizen – Algorithmen und Analysis*. `http://www.mis.mpg.de/scicomp/Fulltext/hmvorlesung.ps`, last accessed 22 May 2006, Jan. 2006.

[127] P. Henrici, *Bounds for iterates, inverses, spectral variation and fields of values of non-normal matrices*, Numerische Mathematik, 4 (1962), pp. 24–40.

[128] V. Hernandez, J. E. Roman, and A. Tomas, *Parallel Arnoldi solvers with enhanced scalability via global communication rearrangement*, Parallel Computing, 33 (2007), pp. 521–540.

[129] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, *An overview of the Trilinos project*, ACM Transactions on Mathematical Software, 31 (2005), pp. 397–423.

[130] ——, *The Trilinos Project*. Available online at `http://trilinos.sandia.gov/`. Last accessed 10 March 2010., 2010.

[131] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, Journal of Research of the National Bureau of Standards, 49 (1952).

[132] U. Hetmaniuk and R. B. Lehoucq, *Basis selection in LOBPCG*, Journal of Computational Physics, 218 (2006), pp. 324–332.

[133] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, PA, USA, second ed., 2002.

[134] N. J. Higham and P. A. Knight, *Matrix powers in finite precision arithmetic*, SIAM Journal on Matrix Analysis and Applications, 16 (1995), pp. 343–358.

[135] A. C. Hindmarsh and H. F. Walker, *Note on a Householder implementation of the GMRES method*, Tech. Rep. UCID-20899, Lawrence Livermore National Laboratory, 1986.

[136] R. W. Hockney, *The Science of Computer Benchmarking*, SIAM, 1996.

[137] J.-W. Hong and H. T. Kung, *I/O complexity: The red-blue pebble game*, in Proc. 13th Ann. ACM Symp. on Theory of Computing (May 11-13, 1981), 1981, pp. 326–333.

[138] A. S. Householder, *Principles of Numerical Analysis*, McGraw-Hill, New York, 1953.

[139] HSL, *A collection of fortran codes for large scale scientific computation*, 2004.

[140] E.-J. Im, *Optimizing the performance of sparse matrix-vector multiplication*, PhD thesis, University of California Berkeley, May 2000.

[141] F. Irigoin and R. Triolet, *Supernode partitioning*, in Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 1988, pp. 319–329.

[142] D. Irony, S. Toledo, and A. Tiskin, *Communication lower bounds for distributed-memory matrix multiplication*, Journal of Parallel and Distributed Computing, 64 (2004), pp. 1014–1026.

[143] W. Jalby and B. Philippe, *Stability analysis and improvement of the block Gram-Schmidt algorithm*, SIAM J. Sci. Stat. Comput., 12 (1991), pp. 1058–1073.

[144] W. D. Joubert and G. F. Carey, *Parallelizable restarted iterative methods for non-symmetric linear systems, Part I: Theory*, International Journal of Computer Mathematics, 44 (1992), pp. 243–267.

[145] ——, *Parallelizable restarted iterative methods for nonsymmetric linear systems, Part II: Parallel implementation*, International Journal of Computer Mathematics, 44 (1992), pp. 269–290.

[146] W. Kahan, *Out-of-core stencil code in the 1950s.* Personal communication, 2007.

[147] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. A. Yelick, *Implicit and explicit optimizations for stencil computations*, in Memory Systems Performance and Correctness, San Jose, CA, Oct. 2006.

[148] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. A. Yelick, *Impact of modern memory subsystems on cache optimizations for stencil computations*, in 3rd Annual ACM SIGPLAN Workshop on Memory Systems Performance, Chicago, IL, 2005.

[149] G. Karypis and V. Kumar, *Metis 4.0: Unstructured graph partitioning and sparse matrix ordering system*, tech. rep., Department of Computer Science, University of Minnesota, 1998. Available online at `http://glaros.dtc.umn.edu/gkhome/views/metis`. Last accessed 10 March 2010.

[150] A. Kiełbasiński, *Analiza numeryczna algorytmu ortogonalizacji Grama–Schmidta*, Seria III: Matematyka Stosowana II, (1974), pp. 15–35.

[151] S. K. Kim and A. T. Chronopoulos, *A class of Lanczos-like algorithms implemented on parallel computers*, Parallel Computing, 17 (1991), pp. 763–778.

[152] ——, *An efficient Arnoldi method implemented on parallel computers*, in Proceedings of the 1991 International Conference on Parallel Processing, D. K. So, ed., vol. III, CRC Press, Aug. 1991, pp. 167–170.

[153] ——, *An efficient nonsymmetric Lanczos method on parallel vector computers*, Journal of Computational and Applied Mathematics, 42 (1992), pp. 357–374.

[154] ——, *An efficient parallel algorithm for extreme eigenvalues of sparse nonsymmetric matrices*, International Journal of High Performance Computing Applications, 6 (1992).

[155] A. V. Knyazev, *BLOPEX webpage*. `http://www-math.cudenver.edu/~aknyazev/software/BLOPEX/`.

[156] A. V. Knyazev, M. Argentati, I. Lashuk, and E. E. Ovtchinnikov, *Block locally optimal preconditioned eigenvalue xolvers (BLOPEX) in HYPRE and PETSc*, Tech. Rep. UCDHSC-CCM-251P, University of California Davis, 2007.

[157] T. Koch and J. Liesen, *The conformal 'bratwurst' maps and associated Faber polynomials*, Numerische Mathematik, 86 (2000).

[158] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. A. Yelick, *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*, tech. rep., DARPA Information Processing Techniques Office, September 2008. Available online at `http://users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/ECS_reports.htm`.

[159] A. N. Krylov, *On the numerical solution of the equation by which in technical questions frequencies of small oscillations of material systems are determined*, Izvestiya Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestvennykh Nauk, 7 (1931), pp. 491–539. In Russian.

[160] J. Kurzak and J. J. Dongarra, *QR factorization for the CELL processor*, Tech. Rep. UT-CS-08-616, University of Tennessee, May 2008. LAWN #201.

[161] C. Lanczos, *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*, J. Res. Nat. Bur. Standards, 45 (1950), pp. 255–282.

[162] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, *Basic Linear Algebra Subprograms for FORTRAN usage*, ACM Transactions on Mathematical Software, 5 (1979), pp. 308–323.

[163] R. B. Lehoucq and K. Maschhoff, *Block Arnoldi method*, in Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, Z. Bai, J. W. Demmel, J. J. Dongarra, A. Ruhe, and H. van der Vorst, eds., Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000, pp. 185–187.

[164] R. B. Lehoucq, D. C. Sorensen, and C. Yang, *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*, SIAM, Philadelphia, PA, 1998.

[165] C. Leiserson, S. Rao, and S. Toledo, *Efficient out-of-core algorithms for linear relaxation using blocking covers*, in Foundations of Computer Science, Nov. 1993.

[166] R. W. Leland, *The effectiveness of parallel iterative algorithms for solution of large sparse linear systems*, PhD thesis, University of Oxford, 1989.

[167] N. Levinson and R. M. Redheffer, *Complex Variables*, McGraw-Hill, Inc., 1970.

[168] X. S. Li, *Sparse Gaussian Elimination on High Performance Computers*, PhD thesis, University of California Berkeley, Sep 1996.

[169] J. Liesen, *Computable convergence bounds for GMRES*, SIAM Journal on Matrix Analysis and Applications, 21 (2000), pp. 882–903.

[170] O. Livne and G. H. Golub, *Scaling by binormalization*, Numer. Algorithms, 35 (2004), pp. 97–120.

[171] C. V. Loan, *On the method of weighting for equality-constrained least-squares problems*, SIAM Journal on Numerical Analysis, 22 (1985).

[172] T. A. Manteuffel, *The Tchebychev iteration for nonsymmetric linear systems*, Numerische Mathematik, 28 (1977), pp. 307–327.

[173] O. Marques, *BLZPACK webpage*. `http://crd.lbl.gov/~osni/`.

[174] J. McCalpin and D. Wonnacott, *Time skewing: A value-based approach to optimizing for memory locality*, Tech. Rep. DCS-TR-379, Department of Computer Science, Rutgers University, 1999.

[175] G. Meurant, *The block preconditioned conjugate gradient method on vector computers*, BIT, 24 (1984), pp. 623–633.

[176] G. A. Meurant, *The Lanczos and Conjugate Gradient Algorithms: From Theory to Finite Precision Computations*, SIAM, 2006.

[177] R. B. Morgan, *Implicitly restarted GMRES and Arnoldi methods for nonsymmetric systems of equations*, SIAM Journal on Matrix Analysis and Applications, 21 (2000), pp. 1112–1135.

[178] R. Nishtala, G. Almási, and C. Caşcaval, *Performance without pain = productivity: Data layout and collective communication in UPC*, in Proceedings of the ACM SIGPLAN 2008 Symposium on Principles and Practice of Parallel Programming, 2008.

[179] R. Nishtala, R. Vuduc, J. W. Demmel, and K. A. Yelick, *Performance modeling and analysis of cache blocking in sparse matrix vector multiply*, Tech. Rep. UCB/CSD-04-1335, University of California Berkeley, Computer Science Department, Jun 2004.

[180] R. Nishtala and K. A. Yelick, *Optimizing collective communication on multicores*, in First USENIX Workshop on Hot Topics in Parallelism, 2009.

[181] J. O'Connor and E. Robertson, *Jorgen pedersen gram*, in MacTutor History of Mathematics Archive, University of St Andrews, 2001. Available online at `http://www-history.mcs.st-andrews.ac.uk/Biographies/Gram.html`. Last accessed 15 Feb 2010.

[182] D. P. O'Leary, *The block conjugate gradient algorithm and related methods*, Linear Algebra Appl., 29 (1980), pp. 293–322.

[183] ——, *The block conjugate gradient algorithm and related methods*, Linear Algebra and its Applications, 29 (1980), pp. 293–322.

[184] P. S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann, 1997.

[185] C. C. Paige, *The Computation of Eigenvalues and Eigenvectors of Very Large Sparse Matrices*, PhD thesis, London University, London, England, 1971.

[186] C. C. Paige, M. Rozložník, and Z. Strakoš, *Modified Gram-Schmidt (MGS), least squares, and backward stability of MGS-GMRES*, SIAM Journal on Matrix Analysis and Applications, 28 (2006), pp. 264–284.

[187] C. C. Paige and M. A. Saunders, *Solution of sparse indefinite systems of linear equations*, SIAM Journal on Numerical Analysis, 12 (1975), pp. 617–629.

[188] H. Pan, B. Hindman, and K. Asanović, *Lithe: Enabling efficient composition of parallel libraries*, in First USENIX Workshop on Hot Topics in Parallelism, 2009.

[189] B. N. Parlett, *The Symmetric Eigenvalue Problem*, SIAM, 1998.

[190] B. N. Parlett and C. Reinsch, *Balancing a matrix for calculation of eigenvalues and eigenvectors*, Numer. Math., 13 (1969), pp. 293–304.

[191] B. N. Parlett, D. R. Taylor, and Z. A. Liu, *A look-ahead Lanczos algorithm for unsymmetric matrices*, Mathematics of Computation, 44 (1985), pp. 105–124.

[192] J.-K. Peir, *Program partitioning and synchronization on multiprocessor systems*, PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Mar. 1986.

[193] M. Penner and V. K. Prasanna, *Cache-friendly implementations of transitive closure*, in Proceedings of the International Conference on Parallel Architectures and Compiler Techniques, Barcelona, Spain, September 2001.

[194] ——, *Cache-friendly implementations of transitive closure*, Journal of Experimental Algorithmics, 11 (2006).

[195] C. J. Pfeifer, *Data flow and storage allocation for the PDQ-5 program on the Philco-2000*, Communications of the ACM, 6 (1963), pp. 365–366.

[196] B. Philippe, Y. Saad, and W. Stewart, *Numerical methods in Markov chain modelling*, Operations Research, 40 (1992), pp. 1156–1179.

[197] P. K. Pollett and D. E. Stewart, *An efficient procedure for computing quasi-stationary distributions of Markov chains with sparse transition structure*, Advances in Applied Probability, 26 (1994), pp. 68–79.

[198] A. Pothen and P. Raghavan, *Distributed orthogonal factorization: Givens and Householder algorithms*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 1113–1134.

[199] A. Pothen, H. Simon, and K.-P. Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM Journal on Matrix Analysis and Applications, 11 (1990), pp. 430–452.

[200] G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, F. G. V. Zee, and R. A. van de Geijn, *Scheduling of QR factorization algorithms on SMP and multi-core architectures*, in Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, Toulouse, France, Feb. 2008. FLAME Working Note #24.

[201] E. Rabani and S. Toledo, *Out-of-core SVD and QR decompositions*, in Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing, Norfolk, Virginia, SIAM, Mar. 2001.

[202] L. Reichel, *Newton interpolation at Leja points*, BIT Numerical Mathematics, 30 (1990), pp. 332–346.

[203] E. J. Riedy, *Making static pivoting scalable and dependable*, PhD thesis, University of California Berkeley EECS, 2010. In progress.

[204] E. J. Rosser, *Fine-grained analysis of array computations*, PhD thesis, Dept. of Computer Science, University of Maryland, Sept. 1998.

[205] A. Ruhe, *The two-sided Arnoldi algorithm for nonsymmetric eigenvalue problems*, in Matrix Pencils, B. Kågström and A. Ruhe, eds., vol. 973 of Lecture Notes in Mathematics, Springer-Verlag, Berlin / Heidelberg, 1983, pp. 104–120.

[206] D. Ruiz, *A scaling algorithm to equilibrate both rows and columns norms in matrices*, Tech. Rep. RAL-TR-2001-034, Rutherford Appleton Laboratory, October 2001.

[207] Y. Saad, *Practical use of polynomial preconditionings for the conjugate gradient method*, SIAM J. Sci. Stat. Comput., 6 (1985), pp. 865–881.

[208] ——, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, second ed., 2003.

[209] Y. Saad and M. H. Schultz, *GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., 7 (1986).

[210] H. A. Schwarz, *Über einen Grenzübergang durch alternierendes Verfahren*, Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich, 15 (1870), pp. 272–286.

[211] A. Smoktunowicz, J. Barlow, and J. Langou, *A note on the error analysis of Classical Gram-Schmidt*, Numerische Mathematik, 105 (2006), pp. 299–313.

[212] F. Song, A. YarKhan, and J. J. Dongarra, *Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems*, Tech. Rep. UT-CS-09-638, Computer Science Department, University of Tennessee, Knoxville, April 2009. Also available as LAPACK Working Note #221.

[213] Y. Song and Z. Li, *New tiling techniques to improve cache temporal locality*, in Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, GA, 1999.

[214] P. Sonneveld, *CGS, a fast Lanczos-type solver for nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 36–52.

[215] A. Stathopoulos, *PRIMME webpage.* `http://www.cs.wm.edu/~andreas/software/`.

[216] A. Stathopoulos and K. Wu, *A block orthogonalization procedure with constant synchronization requirements*, SIAM Journal on Scientific Computing, 23 (2002), pp. 2165–2182.

[217] G. W. Stewart, *Block Gram-Schmidt orthogonalization*, SIAM Journal on Scientific Computing, 31 (2008), pp. 761–775.

[218] W. J. Stewart, *MARCA Models: A collection of Markov chain models*, 2009. Available online at `http://www4.ncsu.edu/~billy/MARCA_Models/MARCA_Models.html`. Last accessed 19 Dec 2009.

[219] M. M. Strout, L. Carter, and J. Ferrante, *Rescheduling for locality in sparse matrix computations*, Lecture Notes in Computer Science, 2073 (2001), pp. 137–146.

[220] C. D. Swanson and A. T. Chronopoulos, *Orthogonal s-step methods for nonsymmetric linear systems of equations*, in ICS '92: Proceedings of the 6th international conference on Supercomputing, New York, NY, USA, 1992, ACM Press, pp. 456–465.

[221] S. J. Thomas, *A block algorithm for orthogonalization in elliptic norms*, in Proceedings of the Second Joint International Conference on Vector and Parallel Processing, Lyon, France, September 1–4, 1992, vol. 634 of Lecture Notes in Computer Science, Berlin, 1992, Springer, pp. 379–385.

[222] S. A. Toledo, *Quantitative performance modeling of scientific computations and creating locality in numerical algorithms*, PhD thesis, Massachusetts Institute of Technology, 1995.

[223] R. Underwood, *An iterative block Lanczos method for the solution of large sparse symmetric eigenproblems*, Tech. Rep. Computer Science Department Report 496, Stanford University, 1975.

[224] L. G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, 33 (1990), pp. 103–111.

[225] A. van der Sluis, *Condition numbers and equilibration of matrices*, Numerische Mathematik, 14 (1969).

[226] H. A. van der Vorst, *Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., 13 (1992), pp. 631–644.

[227] H. A. van der Vorst and C. Vuik, *The superlinear convergence behavior of GMRES*, Journal of Computational and Applied Mathematics, 48 (1993).

[228] J. van Rosendale, *Minimizing inner product data dependence in conjugate gradient iteration*, in Proc. IEEE Internat. Confer. Parallel Processing, 1983.

[229] ——. Personal communication, 2006.

[230] D. Vanderstraeten, *A stable and efficient parallel block Gram-Schmidt algorithm*, in Lecture Notes in Computer Science, vol. 1685, Springer, 1999, pp. 1128–1135.

[231] B. Vastenhouw and R. H. Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Review, 47 (2005), pp. 67–95.

[232] P. K. W. Vinsome, *Orthomin, an iterative method for solving sparse sets of simultaneous linear equations*, in Proceedings of the Fourth Symposium on Reservoir Simulation, Society of Petroleum Engineers of AIME, February 1976.

[233] B. Vital, *Étude de quelques méthodes de résolution de problèmes linéaires de grande taille sur multiprocesseur*, Ph.D. dissertation, Université de Rennes I, Rennes, Nov. 1990.

[234] V. Voevodin, *The problem of non-self-adjoint generalization of the conjugate gradient method is closed*, USSR Comput. Maths. and Math. Phys, 23 (1983), pp. 143–144.

[235] R. Vuduc, *Automatic Performance Tuning of Sparse Matrix Kernels*, PhD thesis, University of California Berkeley, Dec. 2003.

[236] ——, *OSKI: Optimized sparse kernel interface*. Online at `http://bebop.cs.berkeley.edu/oski/`, 2009.

[237] R. Vuduc, J. W. Demmel, and K. A. Yelick, *OSKI: A library of automatically tuned sparse matrix kernels*, in Proc. SciDAC 2005, Journal of Physics: Conference Series, San Francisco, CA, 2005.

[238] H. F. Walker, *Implementation of the GMRES and Arnoldi methods using Householder transformations*, Tech. Rep. UCRL-93589, Lawrence Livermore National Laboratory, Oct. 1985.

[239] ——, *Implementation of the GMRES method using Householder transformations*, SIAM Journal on Scientific and Statistical Computing, 9 (1988), pp. 152–163.

[240] J. H. Wilkinson, *The algebraic eigenvalue problem*, Oxford University Press, 1965.

[241] S. W. Williams, L. Oliker, R. Vuduc, J. Shalf, K. A. Yelick, and J. W. Demmel, *Optimization of sparse matrix-vector multiplication on emerging multicore platforms*, in Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, Nov 2007.

[242] M. E. Wolf, *Improving locality and parallelism in nested loops*, PhD thesis, Stanford University, 1992.

[243] M. M. Wolf, E. G. Boman, and B. Hendrickson, *Optimizing parallel sparse matrix-vector multiplication by corner partitioning*, in PARA08, Trondheim, Norway, May 2008.

[244] D. Wonnacott, *Using time skewing to eliminate idle time due to memory bandwidth and network limitations*, in Proceedings of the Fourteenth International Parallel and Distributed Processing Symposium (IPDPS), 2000, pp. 171–180.

[245] P. R. Woodward and S. E. Anderson, *Scaling the Teragrid by latency tolerant application design*, in Proc. of NSF / Department of Energy Scaling Workshop, Pittsburg, CA, May 2002.

[246] K. Wu and H. D. Simon, *TRLAN webpage*. `http://crd.lbl.gov/~kewu/ps/trlan_.html`.

[247] W. A. Wulf and S. A. McKee, *Hitting the memory wall: implications of the obvious*, ACM SIGARCH Computer Architecture News, 23 (1995).

[248] K. A. Yelick, D. Bonachea, W.-Y. Chen, and R. Nishtala, *Berkeley UPC: Unified Parallel C*. `https://buffy.eecs.berkeley.edu/PHP/resabs/resabs.php?f_year=2006&f_submit=chapgrp&f_chapter=19`, 2006.

[249] B. Zhongzhi, W. Deren, and D. J. Evans, *Models of asynchronous parallel matrix multisplitting relaxed iterations*, Parallel Computing, 21 (1995), pp. 565–582.