Avoiding Communication in Dense Linear Algebra

Grey Ballard

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-151
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-151.html

August 16, 2013

Avoiding Communication in Dense Linear Algebra

by

Grey Malone Ballard

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

and the Designated Emphasis
in

Computational Science and Engineering

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor James Demmel, Chair
Professor Ming Gu
Professor Katherine Yelick

Fall 2013

Avoiding Communication in Dense Linear Algebra

Abstract

Avoiding Communication in Dense Linear Algebra

by

Grey Malone Ballard

Doctor of Philosophy in Computer Science

with a Designated Emphasis in Computational Science and Engineering

University of California, Berkeley

Professor James Demmel, Chair

Dense linear algebra computations are essential to nearly every problem in scientific computing and to countless other fields. Most matrix computations enjoy a high computational intensity (*i.e.*, ratio of computation to data), and therefore the algorithms for the computations have a potential for high efficiency. However, performance for many linear algebra algorithms is limited by the cost of moving data between processors on a parallel computer or throughout the memory hierarchy of a single processor, which we will refer to generally as *communication*. Technological trends indicate that algorithmic performance will become even more limited by communication in the future. In this thesis, we consider the fundamental computations within dense linear algebra and address the following question: can we significantly improve the current algorithms for these computations, in terms of the communication they require and their performance in practice?

To answer the question, we analyze algorithms on sequential and parallel architectural models that are simple enough to determine coarse communication costs but accurate enough to predict performance of implementations on real hardware. For most of the computations, we prove lower bounds on the communication that any algorithm must perform. If an algorithm exists with communication costs that match the lower bounds (at least in an asymptotic sense), we call the algorithm *communication optimal*. In many cases, the most commonly used algorithms are not communication optimal, and we can develop new algorithms that require less data movement and attain the communication lower bounds.

In this thesis, we develop both new communication lower bounds and new algorithms, tightening (and in many cases closing) the gap between best known lower bound and best known algorithm (or upper bound). We consider both sequential and parallel algorithms, and we asses both classical and fast algorithms (*e.g.*, Strassen's matrix multiplication algorithm). In particular, the central contributions of this thesis are

- proving new communication lower bounds for nearly all classical direct linear algebra computations (dense or sparse), including factorizations for solving linear systems,

least squares problems, and eigenvalue and singular value problems,

- proving new communication lower bounds for Strassen's and other fast matrix multiplication algorithms,
- proving new parallel communication lower bounds for classical and fast computations that set limits on an algorithm's ability to perfectly strong scale,
- summarizing the state-of-the-art in communication efficiency for both sequential and parallel algorithms for the computations to which the lower bounds apply,
- developing a new communication-optimal algorithm for computing a symmetric-indefinite factorization (observing speedups of up to 2.8$\times$ compared to alternative shared-memory parallel algorithms),
- developing new, more communication-efficient algorithms for reducing a symmetric band matrix to tridiagonal form via orthogonal similar transformations (observing speedups of 2–6$\times$ compared to alternative sequential and parallel algorithms), and
- developing a new communication-optimal parallelization of Strassen's matrix multiplication algorithm (observing speedups of up to 2.84$\times$ compared to alternative distributed-memory parallel algorithms).

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank foremost my advisor, Jim Demmel, for his guidance and support. From our first meeting during my visit to Berkeley to the detailed discussions over the material in this thesis, he has inspired and excited me about this research. I will be forever grateful for the knowledge learned and experiences gained under his supervision, and I look forward to many more years of collaboration.

I also want to thank my committee members Kathy Yelick and Ming Gu. Their feedback and suggestions during and after my qualifying exam, in addition to the many discussions during BeBOP meetings and Matrix Computation seminars, have been invaluable to this work.

Thank you to all the members of the BeBOP research group, past and present, who have provided continuous feedback on my work and from whom I have learned so much. In particular, I would like to thank those members with whom I have worked most closely, including Mike Anderson, Aydın Buluç, Erin Carson, Andrew Gearhart, Mark Hoemmen, Nick Knight, Ben Lipshitz, and Edgar Solomonik. Thank you to Oded Schwartz for being a friend and mentor over the last five years (and many more to come). All of my graduate work has been the product of joint efforts, and I cannot overstate the value of the many exciting and productive sessions we had in the Par Lab breakout rooms.

I am also grateful to have had fantastic mentorship from outside UC Berkeley. Collaborating with Sivan Toledo at Tel-Aviv University and Laura Grigori at INRIA have been tremendous experiences—I've gained research acumen as well as received beneficial career advice from working with both of them. Working over multiple summers with Tammy Kolda at Sandia National Laboratories has also been incredibly important to my career. She has exposed me to new fields and possibilities, and she has been a great teacher of the tricks of the trade.

Many thanks also to the Par Lab staff, in particular Roxana Infante and Tamille Johnson, who have always been most helpful (and cheerful) in my times of ignorance.

I would also like to thank my wife, Parissa, for her infinite emotional support and love. Experiencing these years together (even at varying distances) has been terrific—you've enriched my emotional highs and tempered the lows—and I look forward to our next adventures together. Thanks also to my parents and family for making me who I am and for their love and support during this time in my life.

I also acknowledge funding from Microsoft (Award #024263) and Intel (Award #024894), and matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from ParLab affiliates National Instruments, Nokia, NVIDIA, Oracle and Samsung, as well as MathWorks. Research is also supported by DOE grants DE-SC0004938, DE-SC0005136, DE-SC0003959, DE-SC0008700, and AC02-05CH11231, and DARPA grant HR0011-12-2-0016, and grant 2010231 from the US-Israel Bi-National Science Foundation.

Chapter 1

Introduction

1.1 The Role of Scientific Computing

Numerical simulation has emerged as the "third pillar" of science along with theory and experimentation [33]. Because of the availability of increasingly powerful computers and the rich development in applied mathematical modeling and algorithmic efficiency, scientists are able to simulate physical phenomena that would otherwise be too expensive, too dangerous, too time-consuming, or simply impossible to observe. As computing facilities grow, the size and complexity of the problems scientists are able to tackle continue to increase. The field of scientific computing aims to enable domain-specific computational scientists to develop and test their ideas (through simulation, data analysis, or other means) by providing software that produces results both quickly and accurately.

While the diversity of scientific domains that depend on efficient computation is vast, from the computer science point of view, the number of different types of fundamental computational patterns required to support all these domains is quite small. Many applications that are very different from each other can be mathematically modeled in the same way. For example, simulating the behavior of a small group of electrons to high accuracy (using "coupled cluster" theory from computational chemistry) and statistically grouping variables within large social science data sets (using "principal component analysis" from statistics) are two applications whose main computational requirements are strikingly similar. In fact, one classification of fundamental computational patterns identifies seven different patterns (coined "the seven dwarves") that cover nearly all domains of computational science [52]. One of the dwarves is dense linear algebra, the topic of this thesis.

1.2 The Importance of Dense Linear Algebra

Dense linear algebra refers to matrix computations where the matrices involved do not have many zero entries. Common matrix computations include solving linear systems of equations, solving "least squares" problems, and computing eigenvalue and singular value decompositions. Dense linear algebra is an especially important dwarf for two reasons: (1) a large fraction of applications require some dense linear algebra computation, even if most of the time is spent on other dwarves, and (2) dense matrix computations have consistently performed well, even on emerging architectures. For example, based on the 2010 computing resource allocation requests of the National Energy Research Scientific Computing Center (a division of Lawrence Berkeley National Laboratory), out of a total of 427 projects, 204 use LAPACK [8] (ranked first of all requested libraries) and 113 use ScaLAPACK [44] (ranked third), both of which are dense linear algebra libraries. These libraries are able to obtain relatively high performance because they benefit from the computations being regular (and therefore have been heavily optimized over many years) and have high arithmetic intensity (a ratio of computation to data size).

Over the past few decades, there have been many advances in dense linear algebra software. The LAPACK library, which includes the Basic Linear Algebra Subroutines (BLAS) [45], was developed in the 1980s and targeted single processor machines (and vector machines of the time). In the 1990s, the Scalable LAPACK (ScaLAPACK) library extended the functionality to distributed-memory parallel computers, where processors communicate over a network. Today, these software packages provide reference implementations, but many other, more-optimized versions of the libraries exist. Some of these libraries are developed and supported by hardware vendors, like Intel's Math Kernel Library (MKL) [93], IBM's Engineering and Scientific Subroutine Library (ESSL) [91], Cray's LibSci [53], and NVIDIA's CUDA BLAS (CUBLAS) [117], and some are open source, like the Automatically Tuned Linear Algebra Software (ATLAS) project [149]. There are also open-source libraries targeting emerging parallel architectures like the Parallel Linear Algebra for Scalable Multicore Architectures (PLASMA) [6] and Matrix Algebra on GPU and Multicore Architectures (MAGMA) [143] libraries, as well as the BLIS, libFLAME, and Elemental libraries developed within the FLAME project [81]. This range of available software demonstrates both the demand for dense linear algebra computation as well as the various techniques for attaining high performance.

1.3 The Rise of Parallelism and the Relative Costs of Communication

In addition to the importance of dense linear algebra, a major reason for the long list of linear algebra packages is the diversity of today's machines. In particular, because of the power, memory, and instruction-level-parallelism walls that began to hit the hardware industry in 2004 [9], in order to improve processor performance, vendors were forced to introduce parallelism into mainstream computer architectures. Parallelism was introduced in the form of chips with multiple cores ("multicore") and with wider vector or Same Instruction Multiple Data (SIMD) lanes, as in general-purpose Graphics Processing Units (GPUs). Processor manufacturers continue to adapt to this new paradigm and use various techniques to handle the complications that arise. As a result, parallelism is both ubiquitous and heterogeneous.

Even before 2004, another hardware trend began to affect the performance of dense linear computations. Despite the fact that many dense matrix computations enjoy a high computational intensity—*i.e.*, every entry of the matrix, also referred to as a *word* of data, is involved in many arithmetic or floating point operations (*flops*)—the cost associated with accessing data from memory to perform computations has become non-negligible. This cost of data movement, or *communication*, is higher than that of performing a flop, and more importantly, this gap has been growing exponentially over time. Both costs have been improving, but the time to perform a flop has been improving by about 59% per year while the rate at which words can be accessed from memory (*i.e.*, memory bandwidth) has improved by about 23% per year [76]. Network improvements have followed a similar pattern, with the relative costs of communication growing exponentially also for distributed-memory machines. Thus, not only is communication becoming more important due to the rise in parallelism, the relative costs of communication are also increasing.

1.4 Thesis Goals and Contributions

The focus of this thesis is to develop a systematic approach for designing and analyzing dense linear algebra algorithms, paying particular attention to avoiding communication costs as much as possible, in order to optimize performance on a wide range of platforms. To maintain general applicability of algorithmic ideas, we will consider one sequential and one parallel architectural model that are simple enough to facilitate coarse asymptotic communication analysis of algorithms but accurate enough to predict performance of implementations on real hardware.

In many cases, we can prove lower bounds on the communication required of a computation on a particular machine model, thereby setting a target for algorithmic performance. Establishing lower bounds and developing and improving algorithms is often a simultaneous process, with the objective of identifying algorithms that are provably communication optimal. After an efficient algorithm has been developed for the modeled machine, we will rely on automatic performance tuning, or *autotuning*, to tweak the parameters of the algorithm for optimal performance on a particular platform. The ultimate goal of this work is to deliver the improved performance of the new and improved algorithms to scientists across many disciplines and other users by integrating the ideas into the state-of-the-art libraries.

To this end, the goals of this thesis are to

(1) prove lower bounds on the communication required by matrix computations, thereby setting targets for algorithmic performance;

(2) survey the current standard algorithms and their communication costs and identify possibilities for improvement; and

(3) present new algorithms for avoiding communication and demonstrate their impact on asymptotic costs and actual performance.

In particular, the central contributions of this work are that we

- prove new communication lower bounds for LU and Cholesky decompositions using reductions from matrix multiplication;
- extend an existing communication lower bound for matrix multiplication [95] to "three-nested-loops" computations for dense or sparse matrices on sequential or parallel machines, which include BLAS computations, one-sided factorizations like LU, Cholesky, LDL^T, and QR, two-sided factorizations for eigenvalue and singular value decompositions, and some computations outside of numerical linear algebra, like computing all-pairs shortest paths of a graph;
- prove new communication lower bounds for Strassen's [139] and "Strassen-like" fast matrix multiplication algorithms by analyzing the the edge expansion of their computation graphs;
- prove new communication lower bounds for parallel algorithms (both classical and Strassen-like) that are independent of the local memory size and impose limits of the possibility of perfect strong scaling for the algorithms;
- summarize the asymptotic communication costs of the current state-of-the-art sequential algorithms (including our recent contributions, some of which do not appear in this thesis) for numerical linear algebra, in terms of words, messages, and cache-obliviousness, and discuss their performance in practice;
- summarize the asymptotic communication costs of the current state-of-the-art parallel algorithms (including our recent contributions, some of which do not appear in this thesis) for numerical linear algebra, in terms of words, messages, and memory requirements, and discuss their performance in practice;
- introduce the first communication-optimal sequential factorization of symmetric indefinite matrices based on Aasen's algorithm [1], prove its backward stability (subject to a growth factor), and present numerical experiments measuring numerical stability (observing speedups of up to 2.8× compared to alternative shared-memory parallel algorithms [15]);
- present a set of improvements and new sequential and parallel algorithms based on successive band reduction that asymptotically reduce communication in computing the eigendecomposition of symmetric band matrices (observing speedups of 2–6× compared to alternative sequential and parallel algorithms [31]); and

- describe a communication-optimal parallelization of Strassen's matrix multiplication algorithm that is superior to all other matrix multiplication algorithms, both in terms of its asymptotic communication costs and its performance in practice (observing speedups of up to 2.84× compared to alternative distributed-memory parallel algorithms [20]).

1.5 Thesis Organization

The content of this thesis is divided between communication lower bounds and algorithms. Chapter 2 presents preliminary ideas that will be useful throughout the thesis, Part I (Chapters 3–6) presents communication lower bound results, and Part II (Chapters 7–11) discusses algorithms and their analyses. We conclude and discuss future directions in Chapter 12.

Of the chapters in Part I, Chapter 3 discusses known lower bounds for classical matrix multiplication and how to extend those results to other computations using reduction arguments. Chapter 4 establishes general theorems which can be applied to most "classical" linear algebra algorithms, and Chapter 5 proves communication lower bounds for Strassen's matrix multiplication algorithm. In Chapter 6 we discuss extensions and further applications of the lower bound techniques and results.

The chapters in Part II are organized as follows. Chapters 7 and 8 summarize the most communication-efficient algorithms on sequential and parallel machines, respectively. Chapters 9–11 discuss new communication-avoiding algorithms for three different computations: symmetric indefinite matrix factorization (Chapter 9), orthogonal tridiagonalization of a symmetric band matrix (Chapter 10), and parallelizing Strassen's matrix multiplication algorithm (Chapter 11).

Chapter 2

Preliminaries

2.1 Notation and Definitions

In this section we define general notation and terminology used throughout the thesis. The most common parameters considered include n, the matrix dimension of a square matrix; M, the size of local memory available to a processor; and P, the number of processors on a parallel computer. For rectangular matrices, we generally use m for the number of rows and n for the number of columns. We use the notation $\lg = \log_2$ and specify the base of other logarithms, except when using asymptotic notation.

2.1.1 Asymptotic Notation

We use standard asymptotic notation throughout the thesis. In particular, we use $O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$. Formally, $f(n) = O(g(n))$ implies that for sufficiently large n there exists a positive constant c such that $f(n) \leq cg(n)$; $f(n) = \Omega(g(n))$ implies that for sufficiently large n there exists a positive constant c such that $f(n) \geq cg(n)$; and $f(n) = \Theta(g(n))$ implies that both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. Informally, we use the notation to hide constants which are small relative to the values of n, M, and P (and possibly other parameters) for reasonably sized problems. We also use the notation $f(n) = \tilde{O}(g(n))$ to mean $f(n) = O(g(n)\log^k n)$ for some positive constant k, and $f(n) \ll g(n)$ and $g(n) \gg f(n)$ to mean that for every positive constant c, $f(n) \leq cg(n)$ for sufficiently large n. Informally, $\tilde{O}(\cdot)$ means "ignoring logarithmic factors" and $f(n) \ll g(n)$ means that f is insignificant relative to g.

We also emphasize that our asymptotic notation does not imply that the results require n to approach infinity to be correct. While the asymptotic analyses provide better estimates for larger parameter values, they have proved meaningful and useful in algorithmic design for modestly sized problems. For example, problems with matrix dimensions in the 100s and processor counts in the 10s are already affected by communication costs and can benefit from the ideas based on the asymptotic costs.

2.1.2 Algorithmic Terminology

In this thesis, we will focus primarily on *direct* algorithms in linear algebra, where the computation is specified by the structure of the input (*e.g.*, the matrix dimension) and not on the numerical values of the input. We specify direct algorithms in contrast to *iterative* algorithms (*e.g.*, Krylov subspace methods), where the computation repeatedly improves its computed solution and the amount of arithmetic depends on the numerical properties of the input.

We use the term *computation* to refer to a specified set of inputs and outputs, as well as the mathematical dependencies among them. We can thus represent a computation with a directed acyclic graph, or *CDAG*, consisting of nodes for input and output elements, as well as temporary intermediate values, and edges representing mathematical dependencies.

We use the term *algorithm* to refer to a particular scheduling of a computation on a given memory model. On a sequential machine, an algorithm specifies the order of evaluation of the scalar operations; on a parallel machine, an algorithm specifies on which processor an operation is performed and the order of operations for each processor. A correct algorithm specifies an order that respects the dependencies of the computation.

For the purposes of this thesis, even though we consider only unary and binary scalar operations, we do not require that all nodes in the CDAG have in-degree less than or equal to two. In particular, the output of a summation of n values has in-degree n. We allow the algorithm (rather than the computation) to exploit associativity and specify the shape of the binary tree used to compute the output.[1]

We will consider two main classes of computations/algorithms (because these classes do not depend on communication, we do not differentiate between computations and algorithms). The following definition is based on [56]:

Definition 2.1. *A* classical *algorithm in linear algebra is one that uses only data on which an output matrix entry mathematically depends*[2] *to compute its value.*

For example, in the case of matrix multiplication $AB = C$, a classical algorithm uses only nonzero entries in row i of A and column j of B to compute C_{ij}. In general, a classical algorithm applied to dense matrices performs $\Theta(n^3)$ flops. Note that a classical algorithm designed for sparse matrices will avoid flops with zeros and may do far fewer operations than $\Theta(n^3)$. We also use the term *conventional* as a synonym of classical. We define classical algorithms in order to differentiate them from *fast* algorithms.

Definition 2.2. *Given a dense linear algebra problem for which the most computationally efficient classical algorithm performs $\Theta(n^3)$ flops, a* fast *algorithm for that problem is one that performs $O(n^{\omega_0})$ flops, where $\omega_0 < 3$.*

[1]Note that because floating point addition is not associative, this implies that two correct algorithms for the same computation may compute different output values.

[2]We assume generic input, ignoring the possibility of exact cancellation.

In particular, fast algorithms do not provide merely a different scheduling for a classical computation; they are based on a different computation graph that computes equivalent outputs in exact arithmetic. They introduce computational dependencies between input and output entries which have no mathematical dependence, but they do so using distributivity and cancellation in a way that decreases the total amount of arithmetic required. The most well-known fast algorithm is Strassen's algorithm for dense matrix multiplication [139]. See Section 2.4 for two variants of his algorithm.

2.1.3 Communication Terminology

In order to estimate the running time of an algorithm, we consider both the computation and the communication it performs. Since the algorithms considered are most commonly used for matrices with real- or complex-valued entries, we measure computation in terms of floating point operations (flops). We measure communication in terms of *words* and *messages*, where a word is one floating point number and a message is a group of words communicated simultaneously. See Section 2.2 for a more precise description of a message in each of the memory models.

We use F, W, and S to denote the number of flops, words, and messages, respectively. While these quantities are generally functions of parameters like n, M, and/or P, we omit the arguments when the context is clear. We also use the terms computational cost, bandwidth cost, and latency cost (defined below) to refer to these quantities.

We assume a fixed cost γ to perform a single flop, and we do not differentiate among scalar additions, subtractions, multiplication, divisions, and square roots. For communication, we assume the cost to communicate a message of m words is $\alpha + \beta m$, where α is the fixed cost for a message and β is the fixed cost for each word. With these assumptions, we can model the running time T of an algorithm as a sum of three terms:

$$T = \gamma \cdot F + \beta \cdot W + \alpha \cdot S.$$

Note that this running time model ignores any overlap of computation and communication. If we model overlap of computation with communication, the running time may decrease by at most a factor of two, which will be inconsequential in our asymptotic analysis. However, to obtain satisfactory performance on an actual implementation, overlapping computation and communication is an important and necessary optimization.

Because we are interested in running time, for parallel algorithms we will consider these costs along the *critical path* of the algorithm, the longest path (measured by time) in the dependency graph of the algorithm's computation and communication steps. That is, if two processors perform a flop simultaneously, or if two pairs of processors communicate a message simultaneously, the cost is that of one flop or one message.[3] For example, see [155] for a discussion of critical path analysis, where the dependency graph is referred to as the "program activity graph," or PAG. In the PAG, each vertex corresponds to the beginning

[3]Note that this implies we ignore network contention; see Section 2.2.2.

or end of a computation step (performing some number of flops) or a communication step (communicating a message of some number of words) of a particular processor, and the weight of an edge between beginning and end vertices is the cost (in time) of the step. For example, if a computation step consists of f flops, the weight of the edge is $\gamma \cdot f$. Extra edges of weight zero represent precedence relationships (*e.g.*, a processor must wait to receive data before it can perform computation with it). The PAG is distinct from the CDAG defined in Section 2.1.2 because it depends on the algorithm used (not only the computation), but it inherits some of its dependencies from the computational dependencies.

Note that, in general, determining the critical path depends on the relative costs of computation and communication. For example, there may be one path that is dominated by computation steps and another dominated by communication steps, so either path may be the critical one. In order to keep the algorithmic analysis separate from the machine-specific costs of computation and communication, we expand the notion of one critical path to three (possibly) distinct paths. We define the *critical path with respect to flops* as the path through the algorithm's dependency graph with the greatest number of flops performed, and we define the *critical path with respect to words* and the *critical path with respect to messages* as the paths through the dependency graph with the greatest number of words and messages communicated, respectively. For most algorithms considered in this thesis, these three paths are the same, independent of the relative values of α, β, and γ.

We define these critical paths to distinguish our communication costs from another metric of communication: *communication volume*, or the sum over all processors of the communication performed. Because this metric is not as closely related to the running time of parallel algorithms on most networks, we do not consider it further.

We now define the principal metrics by which we will evaluate algorithms in terms of computation and communication. For generality, we define these metrics in terms of critical paths; sequential algorithms have only one path.

Definition 2.3. *The* computational cost *of an algorithm is the number of flops performed along the critical path with respect to flops.*

Definition 2.4. *The* bandwidth cost *of an algorithm is the number of words communicated along the critical path with respect to words on a given memory model.*

Definition 2.5. *The* latency cost *of an algorithm is the number of messages communicated along the critical path with respect to messages on a given memory model.*

Note that lower bounds on the computational, bandwidth, and latency costs can be established by proving the existence of a single path with a given number of flops, words, or messages. We will often use a single processor's path, the edges corresponding to that processor's computation and communication steps throughout the algorithm execution, to establish these lower bounds, thus ignoring all inter-processor dependencies.

We also define two commonly used terms to describe algorithms with low communication costs. The first definition is informal, and we use the term more loosely. A *communication-avoiding* algorithm is one that asymptotically reduces the bandwidth cost and/or latency cost compared to the previous most communication-efficient algorithms for the same computation. That is, we use the adjective "communication-avoiding" to denote asymptotic improvement, as opposed to improvements by constant factors that are independent of n, M, or P. We also use the descriptive to distinguish from communication "hiding" by overlapping computation and communication. Hiding communication is an important practical optimization, but it improves run time only by a constant factor and so is ignored in our theoretical model.

The next definition is more precise:

Definition 2.6. *A* communication-optimal *algorithm is one whose bandwidth and latency costs attain known lower bounds in an asymptotic sense*[4] *for the corresponding computation.*

Thus, the existence of communication-optimal algorithms indicates matching lower bounds (given by a proof) and upper bounds (given by an algorithm).

2.2 Memory Models

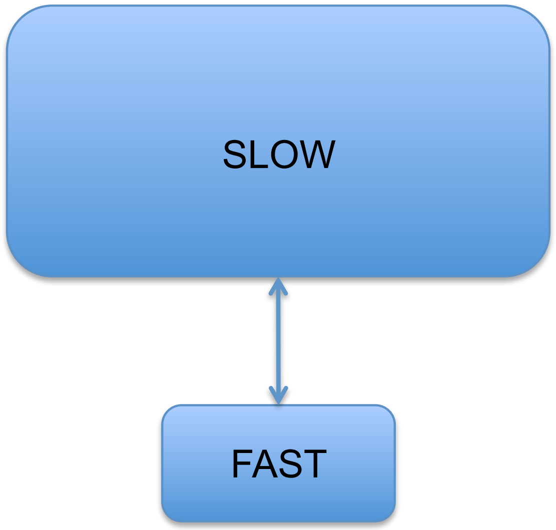
We consider two main types of architectures in this thesis: sequential and parallel computers (see Figure 2.1). Our goals in specifying the two machine models are to (1) provide abstractions that are simple enough to reason about theoretically but realistic enough to be useful in practice and (2) provide two models that are distinct enough to capture the differences between sequential and parallel computation but similar enough to share terminology and fundamental reasoning. To this end, both models include the parameters M, α, β, and γ and consider communication in terms of bandwidth and latency costs. However, the terminology must be interpreted slightly differently in the two models. For example, as we will see below, M is the size of the fast memory in the sequential model, and it is the size of the local memory in the parallel model.

2.2.1 Two-Level Sequential Memory Model

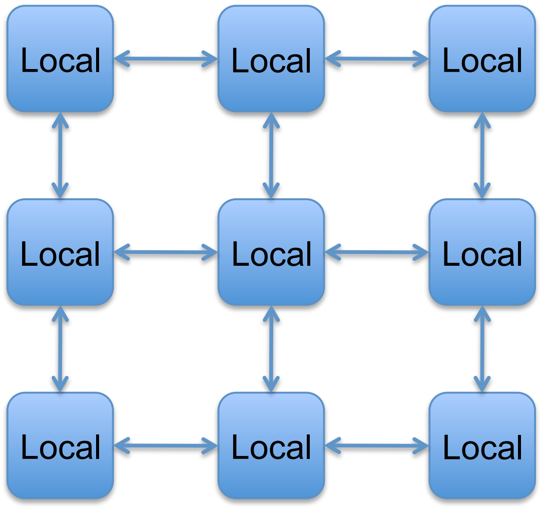
We model a sequential machine with two levels of memory hierarchy (fast and slow) and measure the communication between these two levels during the execution of the algorithm. See Figure 2.1a for a depiction of the model. We use M to denote the size of the fast memory in words. If words are stored contiguously in slow memory, then they can be read or written together as a message. Note that we allow messages to range in size from one word to the size of the fast memory. As described in Section 2.1.3, we measure the number of words and messages separately.

A simple generalization of this model is to consider a smaller maximum message size $L < M$, which is appropriate for modeling cache lines in cache memories, for example. We use this notation in the statement of Theorem 2.11, but in general we assume $L = M$ because this will minimize the latency lower bound for any possible architecture.

[4] By this we mean up to a constant factor and, in some cases, up to a polylogarithmic factor in n, M, or P.



(a) Two-level sequential memory model



(b) Distributed-memory parallel model

Figure 2.1: Main types of memory models used for communication cost analysis.

2.2.1.1 Related Models

This model is similar to the two-level I/O, disk access model, or ideal-cache model (*e.g.*, see [5, 70, 88]), and the bandwidth and latency costs are closely related to the I/O-complexity of an algorithm. In some variations of these models, the I/O complexity refers to the number of words transferred [88], and the contiguity of words in slow memory is ignored. In other variations, I/O complexity refers to the number of messages, or block transfers, where the transfer block size is limited to $B < M$ [5]. Note that in this model, every message is of size exactly B, as opposed to having only a maximum message size L. In this case, the costs of moving one word and B contiguous words are the same. Note also that in this variation, it is possible to communicate multiple blocks simultaneously to model the parallelism in the memory system even for a sequential computational unit. The ideal-cache model [70] is similar, with B representing the cache line size, but only one cache line can be communicated at a time.

2.2.1.2 Hierarchical Memory Model

We also consider a generalization of the two-level memory model to memory hierarchies (see [129], for example). In the hierarchical memory model, there are multiple levels of memory with monotonically increasing size from the smallest memory where computation is performed to the largest memory where all data can reside simultaneously. Data may move only between successive levels of memory, and the costs of data movement vary among pairs of successive levels. We consider this model particularly in the context of *cache-oblivious* algorithms (see [70] and Chapter 7) that attain optimal communication costs between all pairs of levels if they are communication-optimal in the two-level model. Note that cache-aware algorithms can also minimize communication in this model, though they must be tuned for each level of memory.

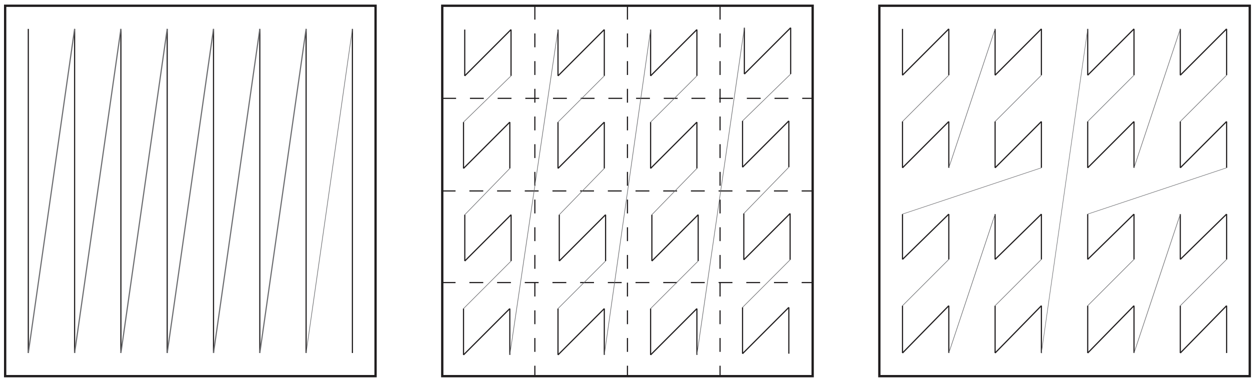
2.2.2 Distributed-Memory Parallel Model

In the distributed-memory parallel case, we model the machine as a collection of P processors, each with a limited local memory of size M, connected over a network. The local memory size limit may be a result of the physical hardware or a restriction on the algorithm (*e.g.*, the algorithm may be limited to using only a constant factor more memory than what is required to store the input and output). Processors communicate via point-to-point messages. See Figure 2.1b for a depiction of the model. Again, we are interested in both the number of words (bandwidth cost) and messages (latency cost), and we count these costs along the critical path(s) of the algorithm. That is, if two processors each send a message to separate processors simultaneously, the cost along the critical path is that of one message. We assume that the per-word and per-message costs include the time it takes a processor to pack words into a contiguous message before sending it over the network.

We assume that (1) the architecture is homogeneous (that is, γ is the same on all processors and α and β are the same between each pair of processors), (2) processors can send/receive only one message to/from one processor at a time, and (3) there is no communication resource contention among processors. That is, we assume that there is a link in the network between each pair of processors. Thus lower bounds derived in this model are valid for any network, but attainability of the lower bounds depends on the details of the network.

2.2.2.1 Related Models

We note that there are many related parallel models. The parallel random access machine (PRAM) model [67] was designed to separate concerns of communication from parallel efficiency given large numbers of processors and uses a shared-memory model. One of many later variants, the LPRAM model [4] captures communication complexity but still incorporates a global shared memory. The Bulk Synchronous Parallel (BSP) model [146] addresses distributed-memory parallel machines and separates algorithmic time into synchronous computational and communication steps. The “communication cost” in that model is equivalent to the bandwidth cost in our model, and the “synchronization cost” is closely related to our latency cost, though in the BSP model a processor can communicate with multiple processors simultaneously. The LogP model [54] is also very similar to our model. Using the notation in Section 2.1.3, α is roughly equivalent to the sum of the “latency” and “overhead” parameters, and β and the “gap” parameters are also roughly equivalent.



(a) Column-major (b) Block-contiguous (c) Recursive

Figure 2.2: Main types of data layouts used for storing matrices in slow memory.

2.2.2.2 Shared-Memory Parallel Models

We note that because of the current ubiquity of multicore processors, shared-memory parallel models are also important to consider in estimating algorithmic performance. While we do not explicitly consider shared-memory models in this thesis, there are several models to which it is possible to extend both our lower bound analyses and algorithms [46, 130, 145]. Another trend in hardware design is greater heterogeneity among processing units. See [18] for a shared-memory heterogeneous parallel model with an extension of the lower bounds and new algorithms for matrix multiplication.

2.3 Data Layouts

2.3.1 Matrix Layouts in Slow Memory

We consider three main types of matrix data layouts in slow memory: column major, block contiguous, and recursive. There are simple variations of these layouts, like row-major instead of column-major ordering, that we do not discuss. Adapting the analysis to these variations is straightforward.

The *column-major* data layout stores the matrix entries in column-wise order, as shown in Figure 2.2a. That is, each column is stored contiguously, ordering column entries from top to bottom, and columns are ordered from left to right. This is the most commonly used layout (*e.g.*, in Fortran) because the indexing into a linear array is simple, and column major is the data layout of choice in LAPACK [8].

The *block-contiguous* data layout involve a block size parameter b, storing the matrix in a way that $b \times b$ blocks are stored contiguously, as shown in Figure 2.2b. The entries within a block may be stored in any layout, and the $(n/b)^2$ blocks may be ordered using any (possibly different) layout. We generally assume column-major orderings both inside

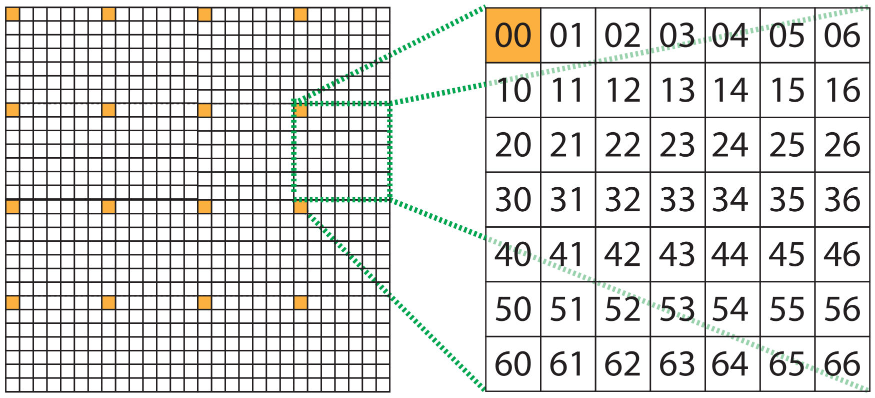


Figure 2.3: Block-cyclic matrix distribution. Each of the 16 submatrices shown on the left has exactly the same distribution. The colored blocks are the ones owned by processor 00. On the right is a zoomed-in view of one submatrix, showing which processor, numbered with row and column indices, owns each block.

and among blocks. Alternatively, the elements of each block may be stored as contiguous sub-blocks, where each sub-block is of size $b' < b$. The data structure may include several such layers of sub-blocks. The contiguous blocks need not be square, but the rectangular generalization will not be useful in this thesis.

The *recursive* layout [70, 154] is also known as the bit-interleaved layout, space-filling curve storage, or Morton ordering format. This layout stores each of the four $(n/2) \times (n/2)$ submatrices of a square matrix contiguously, and then the elements of each submatrix are ordered so that the smaller submatrices are each stored contiguously, and so on recursively, as shown in Figure 2.2c. The recursive layout can be extended to rectangular matrices in various ways, see [32] for an example.

We note that for symmetric matrices, only half the matrix needs to accessed (or stored). See [24, Section 3.1.1] for a summary of adaptations of each of these data structures to symmetric matrices.

2.3.2 Matrix Distributions on Parallel Machines

In the parallel case, we consider one general scheme for distributing matrices across the local memories of the processors: the block-cyclic distribution, shown in Figure 2.3. The *block-cyclic* distribution involves two block size parameters, b_r and b_c, and divides the matrix into $b_r \times b_c$ blocks. For a rectangular $m \times n$ matrix, these $(m/b_r) \cdot (n/b_c)$ blocks are then distributed to processors in a round-robin fashion. We generally assume a two-dimensional logical processor grid, so blocks are distributed so that all blocks in a given block row are distributed among a single processor row (and similarly for block columns).

The block-cyclic distribution is general enough to encompass many important distributions, and it is the method of choice of ScaLAPACK [44]. For example, choosing $b_r = b_c = 1$

gives the cyclic layout (used in Elemental [121]), and choosing $b_r = m/P_r$ and $b_c = n/P_c$ with a $P_r \times P_c$ processor grid gives the block layout. Column-cyclic, row-cyclic, block-column, and block-row distributions can all be achieved with appropriate choices of b_r and b_c.

As mentioned in Section 2.2.2, we assume the communication parameters α and β include the costs incurred by a processor for locally packing words into messages before sending them to another processor. Thus, we do not specify the layout each processor uses to store data in its local memory.

2.4 Fast Matrix Multiplication Algorithms

In this section we give two fast algorithms for matrix multiplication. They each have computational cost of $O(n^{\lg 7})$, where $\lg 7 \approx 2.81$, so they require fewer flops than a classical algorithm. We note that we present them as recursive *algorithms*, but they specify *computations* more generally, as the recursion tree can be traversed in many different orderings. See Section 2.1.2 for a differentiation between algorithms and computations.

2.4.1 Strassen's Algorithm

Here we give Strassen's algorithm for matrix multiplication in its original notation [139]. First, divide the input matrices A, B and output matrix C into 4 submatrices:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \qquad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Then, for one step of the algorithm, compute the following quantities:

$$\begin{aligned}
I &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
II &= (A_{21} + A_{22}) \cdot B_{11} \\
III &= A_{11} \cdot (B_{12} - B_{22}) \\
IV &= A_{22} \cdot (B_{21} - B_{11}) \\
V &= (A_{11} + A_{12}) \cdot B_{22} \\
VI &= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\
VII &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\
\\
C_{11} &= I + IV - V + VII \\
C_{12} &= II + IV \\
C_{21} &= III + V \\
C_{22} &= I + III - II + VI
\end{aligned}$$

The algorithm is recursive since it can be used for each of the 7 smaller matrix multiplications. The recursion for the computational cost of the algorithm is $F(n) = 7F(n/2) + 18n^2$, yielding a solution of

$$F(n) = 6n^{\lg 7} - 5n^2$$

for n a power of two and using a base case of $n = 1$.

2.4.2 Strassen-Winograd Algorithm

The Strassen-Winograd algorithm is usually preferred to Strassen's algorithm in practice since it requires fewer additions (15 instead of 18). As before, we divide the matrices into quadrants. Then we form 7 linear combinations of the submatrices of each of A and B, call these T_i and S_i, respectively; multiply them pairwise; then form the submatrices of C as linear combinations of these products:

$$\begin{array}{llll}
T_0 = A_{11} & S_0 = B_{11} & Q_0 = T_0 \cdot S_0 & U_1 \ = Q_0 + Q_3 \\
T_1 = A_{12} & S_1 = B_{21} & Q_1 = T_1 \cdot S_1 & U_2 \ = U_1 + Q_4 \\
T_2 = A_{21} + A_{22} & S_2 = B_{12} - B_{11} & Q_2 = T_2 \cdot S_2 & U_3 \ = U_1 + Q_2 \\
T_3 = T_2 - A_{11} & S_3 = B_{22} - S_2 & Q_3 = T_3 \cdot S_3 & C_{11} = Q_0 + Q_1 \\
T_4 = A_{11} - A_{21} & S_4 = B_{22} - B_{12} & Q_4 = T_4 \cdot S_4 & C_{12} = U_3 + Q_5 \\
T_5 = A_{12} - T_3 & S_5 = B_{22} & Q_5 = T_5 \cdot S_5 & C_{21} = U_2 - Q_6 \\
T_6 = A_{22} & S_6 = S_3 - B_{21} & Q_6 = T_6 \cdot S_6 & C_{22} = U_2 + Q_2
\end{array}$$

This is one step of Strassen-Winograd. The variation is often erroneously attributed to [152] and actually appears in [153]. In practice, one often uses only a few steps of Strassen-Winograd, although to attain $O(n^{\lg 7})$ computational cost, it is necessary to recursively apply it all the way down to matrices of size $O(1) \times O(1)$. The precise computational cost of Strassen-Winograd (for n a power of 2) is

$$\mathrm{F}(n) = c_s n^{\lg 7} - 5n^2.$$

Here c_s is a constant depending on the cutoff point at which one switches to the classical algorithm. For a cutoff size of n_0, the constant is $c_s = (2n_0 + 4)/n_0^{\lg 7 - 2}$ which is minimized at $n_0 = 8$ yielding a computational cost of approximately $3.73n^{\lg 7} - 5n^2$.

2.5 Lower Bound Lemmas

Chapters 3–6 establish several communication lower bounds for a variety of computations. In this section, we state fundamental definitions and lemmas that will be useful throughout the subsequent chapters.

2.5.1 Loomis-Whitney Inequality

Loomis and Whitney [107] proved a geometrical result that provides a surface-to-volume relationship in general high-dimensional space. We need only the simplest version of their result, which will prove instrumental in proving lower bounds for classical computations in Chapters 4 and 6.

Lemma 2.7 ([107])**.** *Let V be a finite set of lattice points in $\boldsymbol{R}^3$,* i.e., *points (x, y, z) with integer coordinates. Let V_x be the projection of V in the x-direction,* i.e., *all points (y, z)*

such that there exists an x so that $(x, y, z) \in V$. Define V_y and V_z similarly. Let $|\cdot|$ denote the cardinality of a set. Then $|V| \leq \sqrt{|V_x| \times |V_y| \times |V_z|}$.

An intuition for the correctness of this lemma is as follows: think of a box of dimensions $a \times b \times c$. Then its (rectangular) projections on the three planes have areas $a \cdot b$, $b \cdot c$ and $a \cdot c$, and we have that its volume $a \cdot b \cdot c$ is equal to the square root of the product of the three areas. In this instance equality is achieved; only the inequality applies in general.

2.5.2 Expansion Preliminaries

In this section we define the edge expansion of a graph and generalize it slightly for our purposes. These preliminaries will be used in Chapters 5 and 6.

We use common graph notation here, with $V = V(G)$ denoting the set of vertices and $E = E(G)$ the set of edges of a graph $G = (V, E)$. We also use the symbol $\setminus$ to denote set subtraction. Recall that a d-regular graph is one whose vertices all have degree d.

Definition 2.8. *The* edge expansion *$h(G)$ of a d-regular undirected graph $G = (V, E)$ is:*

$$h(G) \equiv \min_{U \subseteq V, |U| \leq |V|/2} \frac{|E(U, V \setminus U)|}{d \cdot |U|} \tag{2.1}$$

where $E(A, B)$ is the set of edges connecting the vertex sets A and B.

If a graph $G = (V, E)$ is not regular but has a bounded maximal degree d, then we can add ($< d$) loops to vertices of degree $< d$, obtaining a regular graph G'. We use the convention that a loop adds 1 to the degree of a vertex. Note that for any $S \subseteq V$, we have $|E_G(S, V \setminus S)| = |E_{G'}(S, V \setminus S)|$, as none of the added loops contributes to the edge expansion of G'.

For many graphs, small sets expand better than larger sets. Let $h_s(G)$ denote the edge expansion for sets of size at most s in G:

$$h_s(G) \equiv \min_{U \subseteq V, |U| \leq s} \frac{|E(U, V \setminus U)|}{d \cdot |U|}. \tag{2.2}$$

For many interesting graph families, $h_s(G)$ does not depend on $|V(G)|$ when s is fixed, although it may decrease when s increases. One way of bounding $h_s(G)$ is by decomposing G into small subgraphs of large edge expansion. Let us first define graph decomposition:

Definition 2.9. *We say that the set of graphs $\{G'_i = (V_i, E_i)\}_{1 \leq i \leq l}$ is an* edge-disjoint decomposition *of $G = (V, E)$ if $V = \bigcup_i V_i$ and $E = \biguplus_i E_i$.*

Now, we state and prove a lemma relating the small-set edge expansion of a graph based on the edge expansion of its component graphs.

Lemma 2.10. *Let $G = (V, E)$ be a d-regular graph that can be decomposed into edge-disjoint (but not necessarily vertex-disjoint) copies of a graph $G' = (V', E')$ with maximum degree d'. Then the edge expansion of G for sets of size at most $|V'|/2$ is $h(G') \cdot \frac{d'}{d}$, namely*

$$h_{\frac{|V'|}{2}}(G) \equiv \min_{U \subseteq V, |U| \leq |V'|/2} \frac{|E_G(U, V \setminus U)|}{d \cdot |U|} \geq h(G') \cdot \frac{d'}{d} \ .$$

Proof. Let $U \subseteq V$ be of size $U \leq |V'|/2$. Let $\{G'_i = (V_i, E_i)\}_{1 \leq i \leq l}$ be an edge-disjoint decomposition of G, where every G'_i is isomorphic to G'. Let $U_i = V_i \cap U$. Then

$$\begin{aligned} |E_G(U, V \setminus U)| &= \sum_{i=1}^{l} |E_{G'_i}(U_i, V_i \setminus U_i)| \geq \sum_{i=1}^{l} h(G'_i) \cdot d' \cdot |U_i| \\ &= h(G') \cdot d' \cdot \sum_{i=1}^{l} |U_i| \geq h(G') \cdot d' \cdot |U| \ . \end{aligned}$$

Therefore $\frac{|E_G(U, V \setminus U)|}{d \cdot |U|} \geq h(G') \cdot \frac{d'}{d}$. □

2.5.3 Latency Lower Bounds

In the subsequent chapters, we state all results in terms of the number of words moved. Corresponding bounds on the number of messages moved exist with a very simple proof: divide the bandwidth cost lower bound by the largest possible message size. Thus, we have the following theorem:

Theorem 2.11. *Suppose a computation has a bandwidth cost lower bound of $W \geq W'$. If L is the largest message size, then its latency cost lower bound is $S \geq W'/L$.*

Because it applies so generally, we do not restate the latency cost lower bound for every bandwidth cost lower bound result.

2.6 Numerical Stability Lemmas

In this section we define our model of floating point computation and state several well-known lemmas regarding the component-wise backward stability of fundamental linear algebra computations. Nearly all of the results in this section can be found in [85].

Our model of floating point arithmetic is:

$$fl\,(x \operatorname{op} y) = (x \operatorname{op} y)\,(1 + \delta)\,, \qquad |\delta| \leq u, \qquad \operatorname{op} = +,\ -,\ \times,\ \div, \tag{2.3}$$

where u is unit roundoff [85, Section 2.2]. In other words, we ignore underflow and overflow. We also assume that 0.5 is a floating point number and that

$$fl\,(0.5x) = 0.5x. \tag{2.4}$$

We begin by citing a few lemmas of floating point error analysis, all of which are either well known or can be easily derived from well-known results. We use the notation $\gamma_n = nu/(1-nu)$ for any positive n. The following lemma provides a rule for manipulating expressions involving γ_n or quantities bounded by it.

Lemma 2.12 ([85, Lemma 3.3]). *The bound*

$$\gamma_m + \gamma_n + \gamma_m\gamma_n \le \gamma_{m+n}$$

holds. Furthermore, if θ_m and θ_n are such that $|\theta_m| \le \gamma_m$ and $|\theta_n| \le \gamma_n$ then

$$(1+\theta_m)(1+\theta_n) = 1 + \theta_{m+n}, \qquad |\theta_{m+n}| \le \gamma_{m+n}.$$

The following lemma provides a bound on the accuracy of matrix-matrix products. In our analysis we assume that matrices are multiplied using the conventional method, as opposed to Strassen's algorithm or any related scheme.

Lemma 2.13 ([85, Section 3.5]). *Let A and B be $m \times p$ and $p \times n$ matrices respectively. If the product $X = AB$ is formed in floating point arithmetic, then*

$$X = AB + \Delta, \qquad |\Delta| \le \gamma_p |A|\,|B|.$$

The following two lemmas also deal with matrix-matrix multiplication. Their proofs are similar to the proof of Lemma 8.4 in [85], with the only difference stemming from the possible scaling by 0.5 in Lemma 2.14. The assumption (2.4) in our model guarantees that this scaling has no effect on the ultimate bound.

Lemma 2.14. *Let A, B and C be matrices of dimensions $m\times p$, $p\times n$ and $m\times n$ respectively, and let α be one of the scalars 0.5 and 1. If the matrix $X = C - \alpha AB$ is formed in floating point arithmetic then*

$$C = \alpha AB + X + \Delta, \qquad |\Delta| \le \gamma_p (\alpha |A|\,|B| + |X|).$$

Lemma 2.15. *Let A, B and C be matrices of dimensions $m\times p$, $p\times m$ and $m\times m$ respectively. If the matrix $X = C - AB - (AB)^T$ is formed in floating point arithmetic then*

$$C = AB + (AB)^T + X + \Delta, \qquad |\Delta| \le \gamma_{2p}\Big(|A|\,|B| + (|A|\,|B|)^T + |X|\Big).$$

Finally, the following two lemmas provide bounds on the accuracy of triangular solves and of the LU factorization.

Lemma 2.16 ([85, Section 8.1]). *Let T and B be matrices of dimensions $m \times m$ and $m \times n$ respectively, and assume that T is triangular. If the $m \times n$ matrix X is computed by solving the system $TX = B$ using substitution in floating point arithmetic then*

$$TX = B + \Delta, \qquad |\Delta| \le \gamma_m |T|\,|X|.$$

Furthermore, if the system being solved is $XT = B$ and the dimensions of X and B are $n \times m$ then

$$XT = B + \Delta, \qquad |\Delta| \le \gamma_m |X|\,|T|.$$

Lemma 2.17 ([85, Section 9.3]). *Let A be an $m \times n$ matrix and let $r = \min\{m, n\}$. If L and U are the LU factors of A, computed in floating point arithmetic, then*

$$A = LU + \Delta, \qquad |\Delta| \leq \gamma_r \, |L| \, |U| \, .$$

Part I

Communication Lower Bounds

Chapter 3

Communication Lower Bounds via Reductions

We establish communication lower bounds for three fundamental classical computations in this chapter: matrix multiplication, LU decomposition, and Cholesky decomposition. In Chapters 4–6, we extend these lower bound results in various ways. The main contributions of this chapter are

- a summary of known communication lower bounds for classical matrix multiplication,
- a reduction proof to extend the bounds to classical LU decomposition, and
- a reduction proof to extend the bounds to classical Cholesky decomposition.

That is, we show how to perform matrix multiplication using a black-box call to an LU or Cholesky factorization routine. Thus, we prove that a lower bound that applies to matrix multiplication also applies to these decompositions (under the same assumptions).

The content of this chapter appears in both [23] (conference version) and [24] (journal version), written with coauthors James Demmel, Olga Holtz, and Oded Schwartz. The journal version of the paper includes a more detailed discussion of the algorithms, but the lower bound argument for Cholesky, presented in Section 3.2.2, appears in both papers. The simpler argument for LU, presented in Section 3.2.1, also appears in [80].

3.1 Classical Matrix Multiplication

In 1981 Hong and Kung [88] proved a lower bound on the bandwidth cost required to perform dense matrix multiplication in the sequential two-level memory model using the classical algorithm, where the input matrices are too large to fit in the fast memory. They obtained the following result using what they called a "red-blue pebble game" analysis of the computation graph of the algorithm. For algorithms attaining this bound, see Section 7.1.1.

Theorem 3.1 ([88, Corollary 6.2])**.** *For classical matrix multiplication of dense $m \times k$ and $k \times n$ matrices implemented on a machine with fast memory of size M, the number of words transferred between fast and slow memory is*

$$W = \Omega\left(\frac{mkn}{M^{1/2}}\right).$$

This result was proven using a different technique by Irony, Toledo, and Tiskin [95] and generalized to the distributed-memory parallel case. They state the following parallel bandwidth cost lower bound using an argument based on the Loomis-Whitney inequality [107], given as Lemma 2.7.

Theorem 3.2 ([95, Theorem 3.1])**.** *For classical matrix multiplication of dense $m \times k$ and $k \times n$ matrices implemented on a distributed-memory machine with P processors each with a local memory of size M, the number of words communicated by at least one processor is*

$$W = \Omega\left(\frac{mkn}{PM^{1/2}} - M\right).$$

In the case where $m = k = n$ and each processor stores the minimal $M = O(n^2/P)$ words of data, the lower bound on bandwidth cost becomes $\Omega(n^2/P^{1/2})$. The authors also consider the case where the local memory size is much larger, $M = \Theta(n^2/P^{2/3})$, in which case $O(P^{1/3})$ times as much memory is used (compared to the minimum possible) and less communication is necessary. In this case the bandwidth cost lower bound becomes $\Omega(n^2/P^{2/3})$. See Sections 8.1.1 and 8.2.1 for discussions of algorithms that attain this bound.

3.2 Reduction Arguments

3.2.1 LU Decomposition

Given a lower bound for one algorithm, we can make a reduction argument to extend that bound to another algorithm. In our case, given the matrix multiplication bounds from Section 3.1, if we can show how to perform matrix multiplication using another algorithm (assuming the transformation requires no extra communication in an asymptotic sense), then the same bound must apply to the other algorithm, under the same assumptions.

A reduction of matrix multiplication to LU decomposition is straightforward given the following identity:

$$\begin{pmatrix} I & 0 & -B \\ A & I & 0 \\ 0 & 0 & I \end{pmatrix} = \begin{pmatrix} I & & \\ A & I & \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} I & 0 & -B \\ & I & A \cdot B \\ & & I \end{pmatrix}. \tag{3.1}$$

That is, given two input matrices A and B, we can compute $A \cdot B$ by constructing the matrix on the left hand side of the identity above, performing an LU decomposition, and then extracting the $(2, 3)$ block of the upper triangular output matrix. Thus, given an algorithm for LU decomposition that communicates less than the lower bound for multiplication, we have an algorithm for matrix multiplication that communicates less than the lower bound, a contradiction. Note that although the dimension of the LU decomposition is three times that of the original multiplication, the same communication bound holds in an asymptotic sense. This reduction appears in [80]. Stated formally:

Theorem 3.3. *Given a fast/local memory of size M, the bandwidth cost lower bound for classical LU decomposition of a dense $n \times n$ matrix is*

$$W = \Omega\left(\frac{n^3}{PM^{1/2}}\right).$$

For a discussion of algorithms attaining this bound, see Sections 7.1.4 and 8.1.4.

3.2.2 Cholesky Decomposition

A similar identity to Equation 3.1 holds for Cholesky decomposition:

$$\begin{pmatrix} I & A^T & -B \\ A & I + A\cdot A^T & 0 \\ -B^T & 0 & D \end{pmatrix} = \begin{pmatrix} I & & \\ A & I & \\ -B^T & (A\cdot B)^T & X \end{pmatrix} \cdot \begin{pmatrix} I & A^T & -B \\ & I & A\cdot B \\ & & X^T \end{pmatrix}$$

where X is the Cholesky factor of $D' \equiv D - B^T B - B^T A^T A B$, and D can be any symmetric matrix such that D' is positive definite.

However, the reduction is not as straightforward as in the case of LU because the matrix-multiplication-by-Cholesky algorithm would include the computation of $A\cdot A^T$ which requires as much communication as general matrix multiplication.[1] We next show how to change the computation so that we can avoid constructing the $I + A\cdot A^T$ term and still perform Cholesky decomposition to obtain the product $A \cdot B$.

In addition to the real numbers $\mathbb{R}$, consider new "starred" numerical quantities, called 1^* and 0^*, with arithmetic properties detailed in the following tables. The quantities 1^* and 0^* mask any real value in addition/subtraction operation, but behave similarly to $1 \in \mathbb{R}$ and $0 \in \mathbb{R}$ in multiplication and division operations.

Table 3.1 defines arithmetic operations with these new quantities. Consider the set $\mathbb{R} \cup \{1^*, 0^*\}$ with the specified arithmetic operations.

- The set is commutative with respect to addition and to multiplication (by the symmetries of the corresponding tables).
- The set is associative with respect to addition: regardless of ordering of summation, the sum is 1^* if one of the summands is 1^*, otherwise it is 0^* if one of the summands is 0^*.

[1]To see why, take $A = \begin{pmatrix} X & 0 \\ Y^T & 0 \end{pmatrix}$, and then $A \cdot A^T = \begin{pmatrix} * & XY \\ * & * \end{pmatrix}$.

$\pm$	1^*	0^*	y
1^*	1^*	1^*	1^*
0^*	1^*	0^*	0^*
x	1^*	0^*	$x \pm y$

$\cdot$	1^*	0^*	y
1^*	1^*	0^*	y
0^*	0^*	0	0
x	x	0	$x \cdot y$

$/$	1^*	0^*	$y \neq 0$
1^*	1^*	$-$	$1/y$
0^*	0^*	$-$	0
x	x	$-$	x/y

$\sqrt{\cdot}$	
1^*	1^*
0^*	0^*
$x \geq 0$	$\sqrt{x}$

Table 3.1: Arithmetic operations with starred values. The variables x, y stand for any *real* values. For consistency, $-0^* \equiv 0^*$ and $-1^* \equiv 1^*$.

- The set is also associative with respect to multiplication: $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. This is trivial if all factors are in $\mathbb{R}$. As 1^* is a multiplicative identity, it is also immediate if some of the factors equal 1^*. Otherwise, at least one of the factors is 0^*, and the product is 0.

- Distributivity, however, does not hold: $1 \cdot (1^* + 1^*) = 1 \neq 2 = (1 \cdot 1^*) + (1 \cdot 1^*)$.

Let us return to the construction. We set T to be:

$$T \equiv \begin{pmatrix} I & A^T & -B \\ A & C & 0 \\ -B^T & 0 & C \end{pmatrix}$$

where C has 1^* on the main diagonal and 0^* everywhere else:

$$C \equiv \begin{pmatrix} 1^* & 0^* & & \cdots & 0^* \\ 0^* & 1^* & 0^* & & \vdots \\ & & & \ddots & \\ \vdots & & & & 0^* \\ 0^* & \cdots & & 0^* & 1^* \end{pmatrix}.$$

One can verify that the (unique) Cholesky decomposition of C is

$$C = \begin{pmatrix} 1^* & 0 & \dots & 0 \\ 0^* & 1^* & & \vdots \\ \vdots & & \ddots & 0 \\ 0^* & \cdots & 0^* & 1^* \end{pmatrix} \cdot \begin{pmatrix} 1^* & 0^* & \cdots & 0^* \\ & \ddots & \ddots & \vdots \\ \vdots & & 1^* & 0^* \\ 0 & \dots & & 1^* \end{pmatrix} \equiv C' \cdot C'^T. \tag{3.2}$$

Note that if a matrix X does not contain any "starred" values 0^* and 1^* then $X = C \cdot X = X \cdot C = C' \cdot X = X \cdot C' = C'^T \cdot X = X \cdot C'^T$ and $C + X = C$. Therefore, one can confirm that the Cholesky decomposition of T is:

$$T \equiv \begin{pmatrix} I & A^T & -B \\ A & C & 0 \\ -B^T & 0 & C \end{pmatrix} = \begin{pmatrix} I & & \\ A & C' & \\ -B^T & (A \cdot B)^T & C' \end{pmatrix} \cdot \begin{pmatrix} I & A^T & -B \\ & C'^T & A \cdot B \\ & & C'^T \end{pmatrix} \equiv L \cdot L^T. \tag{3.3}$$

One can think of C as masking the $A \cdot A^T$ previously appearing in the central block of T, therefore allowing the lower bound of computing $A \cdot B$ to be accounted for by the Cholesky decomposition, and not by the computation of $A \cdot A^T$. More formally, let Alg be any classical algorithm for Cholesky factorization. We convert it to a matrix multiplication algorithm using Algorithm 3.1.

Algorithm 3.1 Matrix Multiplication by Cholesky Decomposition

```
Require: Two $n \times n$ matrices, $A$ and $B$
1: Let $Alg'$ be $Alg$ updated to correctly handle the new $0^*$ and $1^*$ values
                                  ▷ note that $Alg'$ can be constructed off-line
2: Construct $T$ as in Equation (3.3)
3: $L = Alg'(T)$
4: return $(L_{32})^T$
```

The simplest conceptual way to implement line 1 of the algorithm is to attach an extra bit to every numerical value, indicating whether it is "starred" or not, and modify every arithmetic operation to check this bit before performing an operation. This increases the bandwidth cost by at most a constant factor. Alternatively, we can use Signalling NaNs as defined in the IEEE Floating Point Standard [92] to encode 1^* and 0^* with no extra bits.

If the instructions implementing Cholesky are scheduled deterministically, there is another alternative. One can run the algorithm "symbolically", propagating 0^* and 1^* arguments from the inputs forward, simplifying or eliminating arithmetic operations whose inputs contain 0^* or 1^*. One can also eliminate operations for which there is no path in the directed acyclic graph (describing how outputs of each operation propagate to inputs of other operations) to the desired output $A \cdot B$. The resulting Alg' performs a strict subset of the arithmetic and memory operations of the original Cholesky algorithm.

We note that updating Alg to form Alg' is done off-line, so that line 1 does not actually take any time to perform when Algorithm 3.1 is called.

3.2.2.1 Correctness of Algorithm 3.1

We next verify the correctness of this reduction: that the output of this procedure on input A and B is indeed the multiplication $A \cdot B$, as long as Alg is a classical algorithm, in a sense we now define carefully.

Let $T = L \cdot L^T$ be the Cholesky decomposition of T. Then we have the following formulas:

$$L(i,i) = \sqrt{T(i,i) - \sum_{k=1}^{i-1} (L(i,k))^2} \tag{3.4}$$

$$L(i,j) = \frac{1}{L(j,j)} \left(T(i,j) - \sum_{k=1}^{j-1} L(i,k) \cdot L(j,k) \right), \ i > j. \tag{3.5}$$

A "classical" Cholesky decomposition algorithm computes each of these $O(n^3)$ flops, which may be reordered using only commutativity and associativity of addition. By the no-pivoting and no-distributivity restrictions on Alg, when an entry of L is computed, all the entries on which it depends have already been computed and combined by the above formulas, with the sums occurring in any order. These dependencies form a dependency graph on the entries of L, and impose a partial ordering on the computation of the entries of L (see Figure 3.1). That is, when an entry $L(i,i)$ is computed, by Equation (3.4), all the entries $\{L(i,k)\}_{1\le k<i}$ have already been computed.[2] Denote this set of entries by S_{ii}, namely,

$$S_{ii} \equiv \{L(i,k)\}_{1\le k<i}. \tag{3.6}$$

Similarly, when an entry $L(i,j)$ (for $i > j$) is computed, by Equation (3.5), all the entries $\{L(i,k)\}_{1\le k<j}$ and all the entries $\{L(j,k)\}_{1\le k\le j}$ have already been computed. Denote this set by S_{ij} namely,

$$S_{ij} \equiv \{L(i,k)\}_{1\le k<j} \cup \{L(j,k)\}_{1\le k\le j}. \tag{3.7}$$

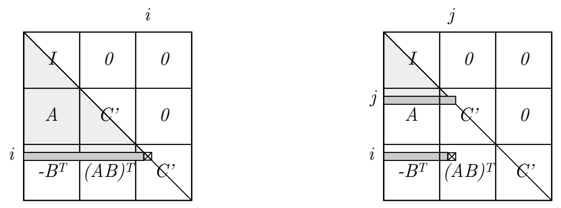


Figure 3.1: Dependencies of $L(i,j)$, for diagonal entries (left) and other entries (right). Dark grey represents the sets S_{ii} (left) and S_{ij} (right). Light grey represents indirect dependencies.

Lemma 3.4. *Any ordering of the computation of the elements of L that respects the partial ordering induced by the computation graph results in a correct computation of $A \cdot B$.*

Proof. We need to confirm that the starred entries 1^* and 0^* of T do not somehow "contaminate" the desired entries of L_{32}^T. The proof is by induction on the partial order on pairs (i,j) implied by (3.6) and (3.7). The base case —the correctness of computing $L(1,1)$— is immediate. Assume by induction that all elements of S_{ij} are correctly computed and consider the computation of $L(i,j)$ according to the block in which it resides:

- If $L(i,j)$ resides in block L_{11}, L_{21} or L_{31} then S_{ij} contains only real values, and no arithmetic operations with 0^* or 1^* occur (recall Figure 3.1 or Equations (3.3),(3.6)

[2]While this partial ordering constrains the scheduling of flops, it does not uniquely identify a computation DAG (directed acyclic graph), for the additions within one summation can be in arbitrary order (forming arbitrary subtrees in the computation DAG).

and (3.7)). Therefore, the correctness follows from the correctness of the original Cholesky algorithm.

- If $L(i,j)$ resides in L_{22} or L_{33} then S_{ij} may contain "starred" value (elements of C'). We treat separately the case where $L(i,j)$ is on the main diagonal and the case where it is not.

  If $i = j$ then by Equation (3.4) $L(i,i)$ is determined to be 1^* since $T(i,i) = 1^*$ and since adding to, subtracting from and taking the square root of 1^* all result in 1^* (recall Table 3.1 and Equation (3.4)).

  If $i > j$ then by the inductive assumption the divisor $L(j,j)$ of Equation (3.5) is correctly computed to be 1^* (recall Figure 3.1 and the definition of C' in Equation (3.2)). Therefore, no division by 0^* is performed. Moreover, $T(i,j)$ is 0^*. Then $L(i,j)$ is determined to be the correct value 0^*, unless 1^* is subtracted (recall Equation (3.5)). However, every subtracted product (recall Equation (3.5)) is composed of two factors of the same column but of different rows. Therefore, by the structure of C', none of them is 1^* so their product is not 1^* and the value is computed correctly.

- If $L(i,j)$ resides in L_{32} then S_{ij} may contain "starred" values (see Figure 3.1, right-hand side, row j). However, every subtraction performed (recall Equation (3.5)) is composed of a product of two factors, of which one is on the ith row (and on a column $k < j$). Hence, by induction (on i, j), the (i,k) element has been computed correctly to be a real value, and by the multiplication properties so is the product. Therefore no masking occurs.

This completes the proof of Lemma 3.4. □

3.2.2.2 Lower Bound Argument

We now know that Algorithm 3.1 correctly multiplies matrices "classically", and so has known communication lower bounds given by Theorems 3.1 and 3.2. It remains to confirm that Step 2 (setting up T) and Step 4 (returning L_{32}^T) do not require much communication, so that these lower bounds apply to Step 3, running Alg' (recall that Step 1 may be performed off-line and so doesn't count). Since Alg' is either a small modification of Cholesky to add "star" labels to all data items (at most doubling the bandwidth cost), or a subset of Cholesky with some operations omitted (those with starred arguments, or not leading to the desired output L_{32}), a lower bound on communication for Alg' is also a lower bound for Cholesky.

That is, any sequential or parallel classical algorithm for the Cholesky decomposition of n-by-n matrices can be transformed into a classical algorithm for $\frac{n}{3}$-by-$\frac{n}{3}$ matrix-multiplication, in such a way that the bandwidth cost of the matrix-multiplication algorithm is at most a constant times the bandwidth cost of the Cholesky algorithm. Therefore any bandwidth or latency cost lower bound for classical matrix multiplication applies to classical Cholesky, asymptotically speaking:

Theorem 3.5. *Given a fast/local memory of size M, the bandwidth cost lower bound for classical Cholesky decomposition of a dense symmetric $n \times n$ matrix is*

$$W = \Omega\left(\frac{n^3}{PM^{1/2}}\right).$$

Proof. Constructing T (in any data format) requires bandwidth of at most $18n^2$ (copying a $3n$-by-$3n$ matrix, with another factor of 2 if each entry has a flag indicating whether it is "starred" or not), and extracting L_{32}^T requires another n^2 of bandwidth. Furthermore, we can assume $n^2 < n^3/M^{1/2}$, *i.e.*, that $M < n^2$, *i.e.*, that the matrix is too large to fit entirely in fast memory (the only case of interest). Thus the bandwidth lower bound $\Omega(n^3/M^{1/2})$ of Algorithm 3.1 dominates the bandwidth costs of Steps 2 and 4, and so must apply to Step 3 (Cholesky). Finally, as each message delivers at most M words, the latency lower bound for Step 3 is by a factor of M smaller than its bandwidth cost lower bound, as desired.

The argument in the parallel case is analogous. The construction of input and retrieval of output at Steps 2 and 4 of Algorithm 3.1 contribute bandwidth of $O(n^2/P)$. Therefore the lower bound of the bandwidth $\Omega(n^3/(PM^{1/2}))$ is determined by Step 3, the Cholesky decomposition. The lower bound on the latency of Step 3 is therefore $\Omega(n^3/(PM^{3/2}))$, as each message delivers at most M words. □

For algorithms attaining this bound, see Sections 7.1.2 and 8.1.2.

3.3 Conclusions

In this chapter we establish the fact that LU and Cholesky decompositions are as hard as matrix multiplication, in terms of their communication requirements. In the Chapter 4, we reproduce these results with a different, direct proof and show that the new proof can be applied to many other computations. In fact, the later results generalize those here because they also apply to sparse and rectangular (in the case of LU) matrices, while Theorems 3.3 and 3.5 assume dense, square matrices. We include the results of this chapter because (1) they predate those of Chapter 4 and (2) the proof technique may prove valuable for other computations. For instance, the best known lower bound results for QR decomposition (see Section 4.3) require technical assumptions; a reduction from matrix multiplication to QR could make the results more robust. In addition, there exists an incomplete reduction in [17, Section 4.1] of computing a Schur decomposition to computing a QR decomposition. Note also that the results here assume "classical" decompositions (see Definition 2.1); algorithms that compute the decompositions with the help of Strassen's or other fast matrix multiplication algorithm can communicate less than algorithms for classical matrix multiplication.

Chapter 4

Lower Bounds for Classical Linear Algebra

While Chapter 3 extends the lower bounds for matrix multiplication (given in Section 3.1) via reduction arguments, this chapter presents a much more general set of arguments to establish lower bounds for nearly all of classical linear algebra. In particular, this approach makes no assumption on the sparsity of the matrices involved in the computation, so it generalizes all of the results in the previous chapter. Intuitively, matrix multiplication is the most fundamental computation in numerical linear algebra—in fact, it is used as a subroutine in nearly every other algorithm in linear algebra—so it seems no surprise that any lower bound for matrix multiplication would also apply to many other computations. The goal of this section is to confirm that suspicion rigorously.

The key observation, and the basis for the arguments in this section, is that the proof technique of Irony, Toledo, and Tiskin [95] can be applied more generally than just to dense matrix multiplication. The geometric argument is based on the lattice of indices (i, j, k) which corresponds to the updates $C_{ij} := C_{ij} + A_{ik} \cdot B_{kj}$. However, the proof does not depend on, for example, the scalar operations being multiplication and addition, the matrices being dense, or the input and output matrices being distinct. The important property is the relationship between the indices (i, j, k) which allows for the embedding of the computation in three dimensions. These observations let us state and prove a more general set of theorems and corollaries that provide a lower bound on the number of words moved into or out of a fast or local memory of size M: for a large class of classical linear algebra algorithms,

$$W = \Omega(G/M^{1/2}),$$

where G is proportional to the total number of flops performed by the processor. In other words, a computation executed using a classical linear algebra algorithm requires at least $\Omega(1/\sqrt{M})$ memory operations for every arithmetic operation, or conversely, the maximum amount of re-use for any word read into fast or local memory during such a computation is $O(\sqrt{M})$ arithmetic operations.

The main contributions of this chapter are lower bound results for dense or sparse matrices, on sequential and parallel machines, for the following computations:

- Basic Linear Algebra Subroutines (BLAS), including matrix multiplication and solving triangular systems;
- LU, Cholesky, LDL^T, LTL^T factorizations, including incomplete versions;
- QR factorization, including approaches based on solving the normal equations, Gram-Schmidt orthogonalization, or applying orthogonal transformations;
- eigenvalue and singular value reductions via orthogonal transformations and computing eigenvectors from Schur form; and
- all-pairs shortest paths computation based on the Floyd-Warshall approach.

This chapter is organized as follows. Section 4.1 generalizes the argument of [95] slightly and demonstrates that it applies to several other fundamental computations, including LU and Cholesky decompositions. In Section 4.2, we address a set of computations which violate certain assumptions in the argument of Section 4.1 (*e.g.*, LDL^T decomposition) and show how to obtain asymptotically equivalent results. Section 4.3 considers computations involving orthogonal transformations, where the lower bound arguments require more complicated analysis and some further assumptions.

All of the results in this chapter (with the exceptions of Sections 4.1.2.4 and 4.2.2.4) appear in [28], written with coauthors James Demmel, Olga Holtz, and Oded Schwartz, though the presentation of the material has been reorganized into the three sections described above. The paper was awarded the SIAM Linear Algebra Prize in 2011.

4.1 Lower Bounds for Three-Nested-Loops Computation

Recall the simplest pseudocode for multiplying $n \times n$ matrices, as three nested loops:

```
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      C[i,j] = C[i,j] + A[i,k] * B[k,j]
```

While the pseudocode above specifies a particular order on the n^3 inner loop iterations, any complete traversal of the index space yields a correct computation (and all orderings generate equivalent results in exact arithmetic). As we will see, nearly all of classical linear algebra can be expressed in a similar way—with three nested loops.

Note that the matrix multiplication computation can be specified more generally in mathematical notation:

$$C_{ij} = \sum_k A_{ik} B_{kj},$$

where the order of summation and order of computation of output entries are left undefined. In this chapter, we specify computations in this general way, but we will use the term "three-nested-loops" to refer to computations that can be expressed with pseudocode similar to that of matrix multiplication above.

4.1.1 Lower Bound Argument

We first define our model of computation formally, and illustrate it on the case of matrix multiplication: $C = C + A \cdot B$. Let $S_a \subseteq \{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$, corresponding in matrix multiplication to the subset of entries of the indices of the input matrix A that are accessed by the algorithm (*e.g.*, the indices of the nonzero entries of a sparse matrix). Let $\mathcal{M}$ be the set of locations in slow/global memory (on a parallel machine $\mathcal{M}$ refers to a location in some processor's memory; the processor number is implicit). Let $\mathfrak{a} : S_a \mapsto \mathcal{M}$ be a mapping from the indices to memory, and similarly define S_b, S_c and $\mathfrak{b}(\cdot, \cdot), \mathfrak{c}(\cdot, \cdot)$, corresponding to the matrices B and C. The value of a memory location $l \in \mathcal{M}$ is denoted by $Mem(l)$. We assume that the values are independent—*i.e.*, determining any value requires accessing the memory location.

Definition 4.1 (3NL Computation). *A computation is considered to be three-nested-loops (3NL) if it includes computing, for all $(i, j) \in S_c$ with $S_{ij} \subseteq \{1, 2, \ldots, n\}$,*

$$Mem(\mathfrak{c}(i, j)) = f_{ij}\left(\{g_{ijk}(Mem(\mathfrak{a}(i, k)), Mem(\mathfrak{b}(k, j)))\}_{k \in S_{ij}}\right)$$

where

(a) mappings $\mathfrak{a}$, $\mathfrak{b}$, and $\mathfrak{c}$ are all one-to-one into slow/global memory, and

(b) functions f_{ij} and g_{ijk} depend non-trivially on their arguments.

Further, define a 3NL operation as an evaluation of a g_{ijk} function, and let G be the number of unique 3NL operations performed:

$$G = \sum_{(i,j) \in S_c} |S_{ij}|.$$

Note that while each mapping $\mathfrak{a}$, $\mathfrak{b}$ and $\mathfrak{c}$ must be one-to-one, the ranges are not required to be disjoint. For example, if we are computing the square of a matrix, then $A = B$ and $\mathfrak{a} = \mathfrak{b}$, and the computation is still 3NL.

By requiring that the functions f_{ij} and g_{ijk} depend "non-trivially" on their arguments, we mean the following: we need at least one word of space to compute f_{ij} (which may or may not be $Mem(\mathfrak{c}(i,j)))$ to act as "accumulator" of the value of f_{ij}, and we need the values $Mem(\mathfrak{a}(i,k))$ and $Mem(\mathfrak{b}(k,j))$ to be in fast or local memory before evaluating g_{ijk}. Note that f_{ij} and g_{ijk} may depend on other arguments, but we do not require that the functions depend non-trivially on them.

Note also that we may not know until after the computation what S_c, f_{ij}, S_{ij}, or g_{ijk} were, since they may be determined on the fly. For example, in the case of sparse matrix multiplication, the sparsity pattern of the output matrix C may not be known at the start of the algorithm. There may even be branches in the code based on random numbers, or in the case of LU decomposition, pivoting decisions are made through the course of the computation.

Now we illustrate the notation in Definition 4.1 for the case of sequential dense $n \times n$ matrix multiplication $C = C + A \cdot B$, where A, B and C are stored in column-major layout in slow memory. We take S_c as all pairs (i,j) with $1 \leq i,j \leq n$ with output entry C_{ij} stored in location $\mathfrak{c}(i,j) = \mathcal{C} + (i-1) + (j-1) \cdot n$, where $\mathcal{C}$ is some memory location. Input matrix entry A_{ik} is analogously stored at location $\mathfrak{a}(i,k) = \mathcal{A} + (i-1) + (k-1) \cdot n$ and B_{kj} is stored at location $\mathfrak{b}(k,j) = \mathcal{B} + (k-1) + (j-1) \cdot n$, where $\mathcal{A}$ and $\mathcal{B}$ are offsets chosen so that none of the matrices overlap. The set $S_{ij} = \{1, 2, ..., n\}$ for all (i,j). Operation g_{ijk} is scalar multiplication, and f_{ij} computes the sum of its n arguments. Thus, $G = n^3$. In the case of parallel matrix multiplication, a single processor will perform only a subset of the computation. In this case, for a given processor, the sizes of the sets S_c and S_{ij} may be smaller than n^2 and n, respectively, and G will become n^3/P if the computation is load-balanced.

We now state and prove the communication lower bound for 3NL computation.

Theorem 4.2. *The bandwidth cost lower bound of a 3NL computation (Definition 4.1) is*

$$W \geq \frac{G}{8\sqrt{M}} - M$$

where M is the size of the fast/local memory.

Proof. Following [95], we consider any implementation of the computation as a stream of instructions involving computations and memory operations: loads and stores from and to slow/global memory. The argument is as follows:

- Break the stream of instructions executed into segments, where each segment contains exactly M load and store instructions (*i.e.*, that cause communication), where M is the fast (or local) memory size.
- Bound from above the number of 3NL operations that can be performed during any given segment, calling this upper bound F.
- Bound from below the number of (complete) segments by the total number of 3NL operations divided by F, *i.e.*, $\lfloor G/F \rfloor$.

- Bound from below the total number of loads and stores, by M (load/stores per segment) times the minimum number of complete segments, $\lfloor G/F \rfloor$, so it is at least $M \cdot \lfloor G/F \rfloor$.

Because functions f_{ij} and g_{ijk} depend non-trivially on their arguments, an evaluation of a g_{ijk} function requires that the two input operands must be resident in fast memory and the output operand (which may be an accumulator) must either continue to reside in fast memory or be written to slow/global memory (it cannot be discarded).

For a given segment, we can bound the number of input and output operands that are available in fast/local memory in terms of the memory size M. Consider the values $\text{Mem}(\mathfrak{c}(i,j))$: for each (i,j), at least one accumulator must reside in fast memory during the segment; since there are at most M words in fast memory at the end of a segment and at most M store operations, there can be no more than $2M$ distinct accumulators. Now consider the values $\text{Mem}(\mathfrak{a}(i,k))$: at the start of the segment, there can be at most M distinct operands resident in fast memory since $\mathfrak{a}$ is one-to-one; during the segment, there can be at most M additional operands read into fast memory since a segment contains exactly M memory operations. If the range of $\mathfrak{a}$ overlaps with the range of $\mathfrak{c}$, then there may be values $\text{Mem}(\mathfrak{a}(i,k))$ which were computed as $\text{Mem}(\mathfrak{c}(i,j))$ values during the segment. Since there are at most $2M$ such operands, the total number of $\text{Mem}(\mathfrak{a}(i,k))$ values available during a segment is $4M$. The same argument holds for $\text{Mem}(\mathfrak{b}(k,j))$ independently. Thus, the number of each type of operand available during a given segment is at most $4M$. Note that this constant can be improved in particular cases, for example when the ranges of $\mathfrak{a}$, $\mathfrak{b}$, and $\mathfrak{c}$ do not overlap.

Now we compute the upper bound F using the geometric result of Loomis and Whitney [107], a simpler form of which is given as Lemma 2.7. Let the set of lattice points (i,j,k) represent each function evaluation $g_{ijk}(\text{Mem}(\mathfrak{a}(i,k)), \text{Mem}(\mathfrak{b}(k,j)))$. For a given segment, let V be the set of indices (i,j,k) of the g_{ijk} operations performed during the segment, V_z be the set of indices (i,j) of their destinations $\mathfrak{c}(i,j)$, V_y be the set of indices (i,k) of their arguments $\mathfrak{a}(i,k)$, and V_x be the set of indices (j,k) of their arguments $\mathfrak{b}(j,k)$. Then by Lemma 2.7,

$$|V| \le \sqrt{|V_x| \cdot |V_y| \cdot |V_z|} \le \sqrt{(4M)^3} \equiv F.$$

Therefore the total number of loads and stores over all segments is bounded by

$$M \left\lfloor \frac{G}{F} \right\rfloor = M \left\lfloor \frac{G}{\sqrt{(4M)^3}} \right\rfloor \ge \frac{G}{8\sqrt{M}} - M.$$

□

Note that the proof of Theorem 4.2 applies to any ordering of the g_{ijk} operations. In the case of matrix multiplication, there are no dependencies between g_{ijk} operations, so every ordering will compute the correct answer. However, for most other computations, there are many dependencies that must be respected for correct computation. This lower bound

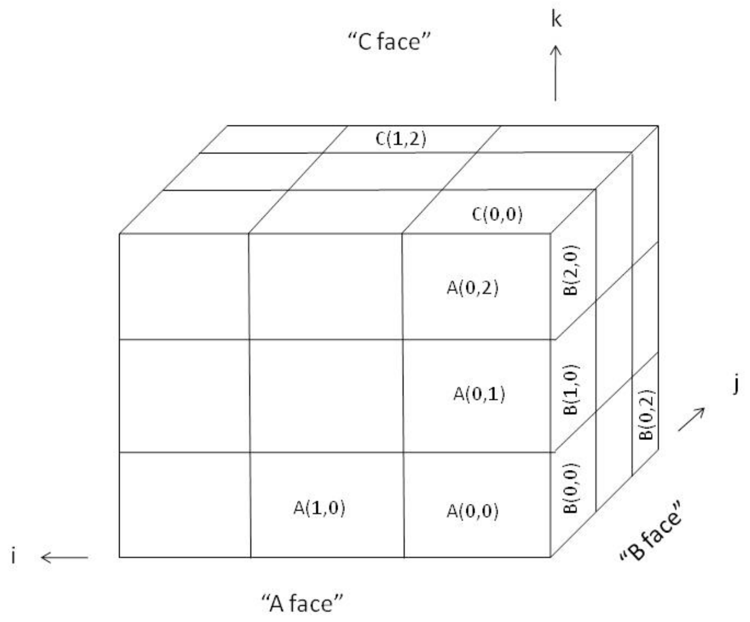


Figure 4.1: Geometric model of matrix multiplication.

argument thus applies not only to correct algorithms, but also to incorrect ones, as long as the computation satisfies the conditions of Definition 4.1.

To see the relationship of the geometric inequality given by Lemma 2.7 to 3NL computations (Definition 4.1), see Figure 4.1, shown for the special case of 3×3 matrix multiplication. We model the computation as an $3 \times 3 \times 3$ set of lattice points, drawn as a set of n^3 $1 \times 1 \times 1$ cubes for easier labeling: each $1 \times 1 \times 1$ cube represents the lattice point at its bottom front right corner. The cubes (or lattice points) are indexed from corner $(i, j, k) = (0, 0, 0)$ to $(n-1, n-1, n-1)$. Cube (i, j, k) represents the multiplication $A(i, k) \cdot B(k, j)$ and its accumulation into $C(i, j)$. The 1×1 squares on the top face of the cube, indexed by (i, j), represent $C(i, j)$, and the 1×1 squares on the other two faces represent $A(i, k)$ and $B(k, j)$, respectively. The set of all multiplications performed during a segment are some subset (V in Lemma 2.7) of all the cubes. All the $C(i, j)$ needed to store the results are the projections of these cubes onto the "C-face" of the cube (V_z in Lemma 2.7). Similarly, the $A(i, k)$ needed as arguments are the projections onto the "A-face" (V_y in Lemma 2.7), and the $B(k, j)$ are the projections onto the "B-face" (V_x in Lemma2.7).

4.1.2 Applications of the Lower Bound

We now show how Theorem 4.2 applies to a variety of classical computations for numerical linear algebra, by which we mean algorithms that would cost $O(n^3)$ arithmetic operations when applied to dense n-by-n matrices, as opposed to Strassen-like algorithms.

4.1.2.1 BLAS

We begin with matrix multiplication, on which we base Definition 4.1. The proof is implicit in the illustration of the definition with matrix multiplication in Section 4.1.1.

Corollary 4.3. *The bandwidth cost lower bound for classical (dense or sparse) matrix multiplication is $G/(8\sqrt{M}) - M$, where G is the number of scalar multiplications performed. In the special case of multiplying a dense $m \times k$ matrix times a dense $k \times n$ matrix, this lower bound is $mkn/(8\sqrt{M}) - M$.*

This reproduces Theorem 3.2 from [95] (with a different constant) for the case of two distinct, dense matrices, though we need no such assumptions. We note that this result could have been stated for sparse A and B in [88]: combine their Theorem 6.1 (their $\Omega(|V|)$ is the number of scalar multiplications) with their Lemma 6.1 (whose proof does not require A and B to be dense). For algorithms attaining this bound in the dense case, see Sections 7.1.1, 8.1.1, and 8.2.1. For further discussion of this bound in the sparse case, see [16, Section 3].

We next extend Theorem 4.2 beyond matrix multiplication. The simplest extension is to the so-called Level-3 BLAS (Basic Linear Algebra Subroutines [45]), which include related operations like multiplication by (conjugate) transposed matrices, by triangular matrices and by symmetric (or Hermitian) matrices. Corollary 4.3 applies to these operations without change (in the case of $A^T \cdot A$ we use the fact that Theorem 4.2 makes no assumptions about the matrices being multiplied not overlapping).

More interesting is the Level-3 BLAS operation for solving a triangular system with multiple right hand sides (TRSM), computing for example $C = A^{-1}B$ where A is triangular. The classical dense computation (when A is upper triangular) is specified by

$$C_{ij} = (B_{ij} - \sum_{k=i+1}^{n} A_{ik} \cdot C_{kj})/A_{ii} \tag{4.1}$$

which can be executed in any order with respect to j, but only in decreasing order with respect to i.

Corollary 4.4. *The bandwidth cost lower bound for classical (dense or sparse) TRSM is $G/(8\sqrt{M}) - M$, where G is the number of scalar multiplications performed. In the special case of solving a dense triangular $n \times n$ system with m right hand sides, this lower bound is $\Omega(mn^2/\sqrt{M})$.*

Proof. We need only verify that TRSM is a 3NL computation. We let f_{ij} be the function defined in Equation (4.1) (or similarly for lower triangular matrices or other variants). Then we make the correspondences that C_{ij} is stored at location $\mathfrak{c}(i,j) = \mathfrak{b}(i,j)$, A_{ik} is stored at location $\mathfrak{a}(i,k)$, and g_{ijk} multiplies $A_{ik} \cdot C_{kj}$. Since A is an input stored in slow/global memory and C is the output of the operation and must be written to slow/global memory, the mappings $\mathfrak{a}$, $\mathfrak{b}$, and $\mathfrak{c}$ are all one-to-one into slow/global memory. Note that $\mathfrak{c} = \mathfrak{b}$ does not prevent the computation from being 3NL. Further, functions f_{ij} (involving a summation of g_{ijk} outputs) and g_{ijk} (scalar multiplication) depend non-trivially on their arguments. Thus, the computation is 3NL.

In the case of dense $n \times n$ triangular A and dense $n \times m$ B, the number of scalar multiplications is $G = \Theta(mn^2)$. $\square$

See Sections 7.1.1, 8.1.1, and 8.2.2 for discussions of algorithms attaining this bound for dense matrices.

Given a lower bound for TRSM, we can obtain lower bounds for other computations for which TRSM is a subroutine. For example, given an $m \times n$ matrix A ($m \geq n$), the Cholesky-QR algorithm consists of forming $A^T A$ and computing the Cholesky decomposition of that $n \times n$ matrix. The R factor is the upper triangular Cholesky factor and, if desired, Q is obtained by solving the equation $Q = AR^{-1}$ using TRSM. Note that entries of R and Q are outputs of the computation, so both are mapped into slow/global memory. The communication lower bounds for TRSM thus apply to the Cholesky-QR algorithm (and reflect a constant fraction of the total number of multiplications of the overall dense algorithm).

We note that Theorem 4.2 also applies to the Level 2 BLAS (like matrix-vector multiplication) and Level 1 BLAS (like dot products), though the lower bound is not attainable. In those cases, the number of words required to access each of the input entries once already exceeds the lower bound of Theorem 4.2.

4.1.2.2 LU factorization

Independent of sparsity and pivot order, the classical LU factorization (with L unit lower triangular) is specified by

$$\begin{aligned} L_{ij} &= (A_{ij} - \sum_{k<j} L_{ik} \cdot U_{kj})/U_{jj} && \text{for} \quad i > j \\ U_{ij} &= A_{ij} - \sum_{k<i} L_{ik} \cdot U_{kj} && \text{for} \quad i \leq j. \end{aligned} \tag{4.2}$$

In the sparse case, the equations may be evaluated for some subset of indices (i, j) and the summations may be over some subset of the indices k. Equation (4.2) also assumes pivoting has already been incorporated in the interpretation of the indices i, j, and k. Note that since the set of input and output operands overlap, there are data dependencies which must be respected for correct computation.

Corollary 4.5. *The bandwidth cost lower bound for classical (dense or sparse) LU factorization is $G/(8\sqrt{M}) - M$, where G is the number of scalar multiplications performed. In the special case of factoring a dense $m \times n$ matrix with $m \geq n$, this lower bound is $\Omega(mn^2/\sqrt{M})$.*

Proof. We need only verify that LU factorization is a 3NL computation. We let f_{ij} be the (piecewise) function defined in Equation (4.2). Then we make the correspondences that L_{ij} and U_{ij} are stored at location $\mathfrak{a}(i, j) = \mathfrak{b}(i, j) = \mathfrak{c}(i, j)$. Note that while $\mathfrak{a} = \mathfrak{b}$, the sets S_a and S_b do not overlap, as they access lower and upper triangles, respectively, though S_c does overlap both S_a and S_b. Since L and U are outputs of the operation and must be written to slow/global memory, the mappings $\mathfrak{a}$, $\mathfrak{b}$, and $\mathfrak{c}$ are all one-to-one into slow/global memory. Further, functions f_{ij} (involving a summation of g_{ijk} outputs) and g_{ijk} (scalar multiplication) depend non-trivially on their arguments. Thus, the computation is 3NL.

In the case of dense $m \times n$ LU factorization where $m \geq n$, the number of scalar multiplications is $G = \Theta(mn^2)$. □

Note that Corollary 4.5 reproduces the result from the reduction argument in Section 3.2.1 for dense and square factorization. However, this corollary is a strict generalization, as it also applies to sparse and rectangular factorizations. For a discussion of algorithms attaining this bound for dense matrices, see Sections 7.1.4 and 8.1.4.

Consider incomplete LU (ILU) factorization [127], where some entries of L and U are omitted in order to speed up the computation. In the case of *level-based* incomplete factorizations (*i.e.*, ILU(p)), Corollary 4.5 applies with G corresponding to the scalar multiplications performed. However, consider *threshold-based* ILU, which computes a possible nonzero entry L_{ij} or U_{ij} and compares it to a threshold, storing it only if it is larger than the threshold and discarding it otherwise. Does Corollary 4.5 apply to this computation?

Because a computed entry L_{ij} may be discarded, the assumption that f_{ij} depends non-trivially on its arguments is violated. However, if we restrict the count of scalar multiplications to the subset of S_c for which output entries are *not* discarded, then all the assumptions of 3NL are met, and the lower bound applies (with G computed based on the subset). This count may underestimate the computation by more than a constant factor (if nearly all computed values fall beneath the threshold), but the lower bound will be valid nonetheless. We consider another technique to arrive at a lower bound for threshold-based incomplete factorizations in Section 4.2.2.2.

4.1.2.3 Cholesky Factorization

Independent of sparsity and (diagonal) pivot order, the classical Cholesky factorization is specified by

$$\begin{aligned} L_{jj} &= (A_{jj} - \sum_{k<j} L_{jk}^2)^{1/2} \\ L_{ij} &= (A_{ij} - \sum_{k<j} L_{ik} \cdot L_{jk})/L_{jj} \quad \text{for} \quad i > j. \end{aligned} \tag{4.3}$$

In the sparse case, the equations may be evaluated for some subset of indices (i, j) and the summations may be over some subset of the indices k. Equation (4.3) also assumes pivoting has already been incorporated in the interpretation of the indices i, j, and k. As in the case of LU factorization, there are data dependencies which must be respected for correct computation.

Corollary 4.6. *The bandwidth cost lower bound for classical (dense or sparse) Cholesky factorization is $G/(8\sqrt{M}) - M$, where G is the number of scalar multiplications performed. In the special case of factoring a dense $n \times n$ matrix, this lower bound is $\Omega(n^3/\sqrt{M})$.*

Proof. We need only verify that Cholesky factorization is a 3NL computation. We let f_{ij} be the (piecewise) function defined in Equation (4.3). Then we make the correspondences that L_{ij} is stored at location $\mathfrak{a}(i,j) = \mathfrak{b}(i,j) = \mathfrak{c}(i,j)$. Note that all three sets S_a, S_b, and S_c overlap. Since L is the output of the operation and must be written to slow/global memory, the mappings $\mathfrak{a}$, $\mathfrak{b}$, and $\mathfrak{c}$ are all one-to-one into slow/global memory. Further, functions f_{ij} (involving a summation of g_{ijk} outputs) and g_{ijk} (scalar multiplication) depend non-trivially on their arguments. Thus, the computation is 3NL.

In the case of dense $n \times n$ Cholesky factorization, the number of scalar multiplications is $G = \Theta(n^3)$. □

Note that Corollary 4.6 reproduces the result from the reduction argument in Section 3.2.2 for dense factorization. However, this corollary is a strict generalization, as it also applies to sparse factorizations. For algorithms attaining this bound in the dense case, see Sections 7.1.2 and 8.1.2. As in the case of LU (Section 4.1.2.2), Corollary 4.6 is general enough to accommodate incomplete Cholesky (IC) factorizations [127].

We now consider Cholesky factorization on a particular class of sparse matrices for which computational lower bounds are known. Since these computational bounds apply to G, Corollary 4.6 leads directly to a concrete communication lower bound. Hoffman, Martin, and Rose [87] and George [73] prove that a lower bound on the number of multiplications required to compute the sparse Cholesky factorization of an n^2-by-n^2 matrix representing a 5-point stencil on a 2D grid of n^2 nodes is $\Omega(n^3)$. This lower bound applies to any matrix containing the structure of the 5-point stencil. This yields:

Corollary 4.7. *In the case of the sparse Cholesky factorization of a matrix which includes the sparsity structure of the matrix representing a 5-point stencil on a two-dimensional grid of n^2 nodes, the bandwidth cost lower bound is $\Omega(n^3/\sqrt{M})$.*

George [73] shows that this arithmetic lower bound is attainable with a nested dissection algorithm in the case of the 5-point stencil. Gilbert and Tarjan [74] show that the upper bound also applies to a larger class of structured matrices, including matrices associated with planar graphs. Recently, David, Demmel, Grigori, and Peyronnet [79] obtained new algorithms for sparse cases of Cholesky decomposition that are proven to be communication optimal using this lower bound.

4.1.2.4 Computing Eigenvectors from Schur Form

The Schur decomposition of a matrix A is the decomposition $A = QTQ^T$, where Q is unitary and T is upper triangular. Note that in the real-valued case, Q is orthogonal and T is quasi-triangular. The eigenvalues of a triangular matrix are given by the diagonal entries.

Assuming all the eigenvalues are distinct, we can solve the equation $TX = XD$ for the upper triangular eigenvector matrix X, where D is a diagonal matrix whose entries are the diagonal of T. This implies that for $i < j$,

$$X_{ij} = \left(T_{ij} X_{jj} + \sum_{k=i+1}^{j-1} T_{ik} X_{kj} \right) / \left(T_{jj} - T_{ii} \right) \tag{4.4}$$

where X_{jj} can be arbitrarily chosen for each j. Note that in the sparse case, the equations may be evaluated for some subset of indices (i, j) and the summations may be over some subset of the indices k. After computing X, the eigenvectors of A are given by QX.

Corollary 4.8. *The bandwidth cost lower bound for classically computing the eigenvectors of a (dense or sparse) triangular matrix with distinct eigenvalues is $G/(8\sqrt{M}) - M$, where G is the number of scalar multiplications performed. In the special case of a dense triangular $n \times n$ matrix, this lower bound is $\Omega(n^3/\sqrt{M})$.*

Proof. We need only verify that the computation is 3NL. We let f_{ij} be the function defined in Equation (4.4). Then we make the correspondences that T_{ij} is stored at location $\mathfrak{a}(i, j)$ and X_{ij} is stored at location $\mathfrak{b}(i, j) = \mathfrak{c}(i, j)$. Since T is the input and must be stored in slow/global memory, and X is the output of the operation and must be written to slow/global memory, the mappings $\mathfrak{a}$, $\mathfrak{b}$, and $\mathfrak{c}$ are all one-to-one into slow/global memory. Further, functions f_{ij} (involving a summation of g_{ijk} outputs) and g_{ijk} (scalar multiplication) depend non-trivially on their arguments. Thus, the computation is 3NL.

In the case of computing the eigenvectors of a dense triangular $n \times n$ matrix, the number of scalar multiplications is $G = \Theta(n^3)$. □

4.1.2.5 Floyd-Warshall All-Pairs Shortest Paths

Theorem 4.2 applies to more general computations than strictly linear algebraic ones, where g_{ijk} are scalar multiplications and f_{ij} are based on summations. We consider the Floyd-Warshall method [68, 148] for computing the shortest paths between all pairs of vertices in a graph. If we define $d_{ij}^{(k)}$ to be the shortest distance between vertex i and vertex j using the first k vertices, then executing the following computation for all k, i, and j determines in $D_{ij}^{(n)}$ the shortest path between vertex i and vertex j using the entire graph:

$$D_{ij}^{(k)} = \min \left(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \right). \tag{4.5}$$

Taking care to respect the data dependencies, the computation can be done in place, with $D^{(0)}$ being the original adjacency graph and each $D^{(k)}$ overwriting $D^{(k-1)}$. The original formulation of the algorithm consists of three nested loops with k as the outermost loop index, but there are many other orderings which maintain correctness.

Corollary 4.9. *The bandwidth cost lower bound for computing all-pairs shortest paths using the Floyd-Warshall method is $G/(8\sqrt{M}) - M$, where G is the number of scalar additions performed. In the special case of computing all-pairs shortest paths on a dense graph with n vertices, this lower bound is $\Omega(n^3/\sqrt{M})$.*

Proof. We need only verify that the Floyd-Warshall method is a 3NL computation. We make the correspondences that $D_{ij}^{(k)}$ is stored at location $\mathfrak{a}(i,j) = \mathfrak{b}(i,j) = \mathfrak{c}(i,j)$. Then we let g_{ijk} be the addition operation (adding values stored at $\mathfrak{a}(i,k)$ and $\mathfrak{b}(k,j)$) and f_{ij} be the minimum of outputs of g_{ijk} over all k. Note that all three sets S_a, S_b, and S_c overlap. Since D is the input and output of the operation and must be written to slow/global memory, the mappings $\mathfrak{a}$, $\mathfrak{b}$, and $\mathfrak{c}$ are all one-to-one into slow/global memory. Further, functions f_{ij} and g_{ijk} depend non-trivially on their arguments. Thus, the computation is 3NL.

In the case of a dense graph with n vertices, the number of scalar additions is $G = \Theta(n^3)$. □

This result is also claimed (without proof) as [118, Lemma 1]. The authors also provide a sequential algorithm attaining the bound. For a parallel algorithm attaining this bound, see [136].

4.2 Lower Bounds for Three-Nested-Loop Computation with Temporary Operands

4.2.1 Lower Bound Argument

Many linear algebraic computations are nearly 3NL but fail to satisfy the assumption that $\mathfrak{a}$, $\mathfrak{b}$ and $\mathfrak{c}$ are one-to-one mappings into slow/global memory. In this section, we show that, under certain assumptions, we can still prove meaningful lower bounds. We will first consider two examples to provide intuition for the proof and then make the argument rigorous.

Consider computing the Frobenius norm of a product of matrices: $\|A{\cdot}B\|_F^2 = \sum_{ij}(A{\cdot}B)_{ij}^2$. If we define f_{ij} as the square of the dot product of row i of A and column j of B, the output of f_{ij} does not necessarily map to a location in slow memory because entries of the product $A{\cdot}B$ are only temporary values, not outputs of the computation (the norm is the only output). However, in order to compute the norm correctly, every entry of $A \cdot B$ must be computed, so the matrix might as well be an output of the computation (in which case Theorem 4.2 would apply). In the proof of Theorem 4.10 below, we show using a technique of *imposing writes* that the amount of communication required for computations with temporary values like these is asymptotically the same as those forced to output the temporary values.

While the example above illustrates temporary output operands of f_{ij} functions, a computation may also have temporary input operands to g_{ijk} functions. For example, if we want to compute the Frobenius norm of a product of matrices where the entries of the input matrices are given by formulas (*e.g.*, $A_{ij} = i^2 + j$), then the computation may require very little communication since the entries can be recomputed on the fly as needed. However, if we require that each temporary input operand be computed only once, and we map each operand to a location in slow/global memory, then if the operand has already been computed and does not reside in fast memory, it must be read from slow/global memory. The assumption that each temporary operand be computed only once seems overly restrictive in the case of matrix entries given by simple formulas of their indices, but for many other common computations (see Section 4.2.2), the temporary values are more expensive to compute and recomputation on the fly is more difficult.

We now make the intuition given in the examples above more rigorous. First, we define a *temporary value* as any value involved in a computation that is not an original input or final output. In particular, a temporary value needs not be mapped to a location in slow memory. Next, we distinguish a particular set of temporary values: we define the temporary inputs to g_{ijk} functions and temporary outputs of f_{ij} functions as *temporary operands*. While there may be other temporary values involved in the computation (*e.g.*, outputs of g_{ijk} functions), we do not consider them temporary operands.[1] A temporary input $\mathfrak{a}(i,k)$ may be an input to multiple g_{ijk} functions (g_{ijk} and $g_{ij'k}$ for $j \neq j'$), but we consider it a single temporary operand. There may also be multiple accumulators for one output of an f_{ij} function, but we consider only the final computed output as a temporary operand. In the case of computing the Frobenius norm of a product of matrices whose entries are given by formulas, the number of temporary operands is $3n^2$, corresponding to the entries of the input and output matrices.

We now state the result more formally:

Theorem 4.10. *Suppose a computation is 3NL except that some of its operands (*i.e.*, inputs to g_{ijk} operations or outputs of f_{ij} functions) are temporary and are not necessarily mapped to slow/global memory. Then if the number of temporary operands is t, and if each (input or output) temporary operand is computed exactly once, then the bandwidth cost lower bound is given by*

$$W \geq \frac{G}{8\sqrt{M}} - M - t$$

where M is the size of the fast/local memory.

Proof. Let $\mathcal{C}$ be such a computation, and let $\mathcal{C}'$ be the same computation with the exception that for each of the three types of operands (defined by mappings $\mathfrak{a}$, $\mathfrak{b}$, and $\mathfrak{c}$), every temporary operand is mapped to a distinct location in slow/global memory and must be written to that location by the end of the computation. This enforces that the mappings $\mathfrak{a}$, $\mathfrak{b}$, and $\mathfrak{c}$ are all one-to-one into slow/global memory, and so $\mathcal{C}'$ is a 3NL computation and Theorem 4.2 applies. Consider an algorithm that correctly performs computation $\mathcal{C}$. Modify the algorithm by imposing writes: every time a temporary operand is computed, we impose a write to the corresponding location in slow/global memory (a copy of the value may remain in fast memory). After this modification, the algorithm will correctly perform the computation $\mathcal{C}'$. Thus, since every temporary operand is computed exactly once, the

[1]We ignore these other temporary values because, as in the case of true 3NL computations, they typically do not require any memory traffic.

bandwidth cost of the algorithm differs by at most t words from an algorithm to which the lower bound $W \geq G/(8\sqrt{M}) - M$ applies, and the result follows. □

4.2.2 Applications of the Lower Bound

4.2.2.1 Solving the Normal Equations

In Section 4.1.2.1, we prove a communication lower bound for the Cholesky-QR computation by applying Theorem 4.2 to the TRSM required to compute the orthogonal matrix Q. In some cases, such as solving the normal equations $A^TAx = A^Tb$ (by forming A^TA and performing a Cholesky decomposition), the Cholesky-QR computation does not form Q explicitly. Here, we apply Theorem 4.10 to the computation A^TA, the output of which is a temporary matrix.

Corollary 4.11. *The bandwidth cost lower bound for solving the normal equations to find the solution to a least squares problem with matrix A is $G/(8\sqrt{M}) - M - t$, where G is the number of scalar multiplications involved in the computation of A^TA, t is the number of nonzeros in A^TA, and we assume the entries of A^TA are computed only once. In the special case of a dense $m \times n$ matrix A with $m \geq n$, this lower bound is $\Omega(mn^2/\sqrt{M})$.*

Proof. As argued in Section 4.1.2.1, computing A^TA is a 3NL computation (we ignore the Cholesky factorization and triangular solves in this proof). That is, $\mathfrak{a}(i,j) = \mathfrak{b}(j,i)$ and f_{ij} is the summation function defined for either the lower ($i \geq j$) or upper ($i \leq j$) triangle because the output is symmetric. Since A is an input to the normal equations, it must be stored in slow memory. However, the output of A^TA need not be stored in slow memory (its Cholesky factor will be used to solve for the final output of the computation). Thus, the number of temporary operands is the number of nonzeros in the output of A^TA, which are all outputs of f_{ij} functions. In the case of a dense $m \times n$ matrix A with $m \geq n$, the output A^TA is $n \times n$. When $m, n \geq \sqrt{M}$, the $mn^2/\sqrt{M}$ term asymptotically dominates the (negative) M and $n^2/2$ terms. □

4.2.2.2 Incomplete Factorizations

In Section 4.1.2.2, we consider lower bounds for threshold-based incomplete LU factorizations. Because Theorem 4.2 requires that the f_{ij} functions depend non-trivially on their arguments, we must ignore all scalar multiplications that lead to discarded outputs (due to their values falling below the threshold). However, because the output values must be fully computed before comparing them to the threshold value, we may be ignoring a significant amount of the computation. Using Theorem 4.10, we can state another (possibly tighter) lower bound which counts all the scalar multiplications performed by imposing reads and writes on the discarded values.

Corollary 4.12. *Consider a threshold-based incomplete LU or Cholesky factorization, and let t be the number of output values discarded due to thresholding. Assuming each dis-*

carded value is computed exactly once, the bandwidth cost lower bound for the computation is $G/(8\sqrt{M}) - M - t$ where G is the number of scalar multiplications.

Proof. As argued in Section 4.1.2.2, sparse LU and Cholesky factorizations are 3NL computations. In a threshold-based incomplete factorizations, some output values are discarded if they are smaller than a given threshold. These discarded values are the only temporary operands in the computation, so the result follows from Theorem 4.10. □

4.2.2.3 LDL^T Factorization

Independent of sparsity and pivot order, the classical Bunch-Kaufman LDL^T factorization [47], where L is unit lower triangular and D is block diagonal with 1×1 and 2×2 blocks, is specified in the case of positive or negative definite matrices (*i.e.*, all diagonal blocks of D are 1×1) by

$$\begin{aligned} D_{jj} &= A_{jj} - \sum_{k<j} L_{jk}^2 D_{kk} \\ L_{ij} &= (A_{ij} - \sum_{k<j} L_{ik} \cdot (D_{kk} L_{jk}))/D_{jj} \quad \text{for} \quad i > j. \end{aligned} \tag{4.6}$$

In the sparse case, the equations may be evaluated for some subset of indices (i, j) and the summations may be over some subset of the indices k. Equation (4.6) also assumes pivoting has already been incorporated in the interpretation of the indices i, j and k.

Note that in this case, the operand $(D_{kk}L_{jk})$ is a temporary operand. This complication is overlooked in [28], where it is claimed that the argument for Cholesky also applies to LDL^T factorization. In the terminology of this chapter, it is assumed that LDL^T is 3NL. Here, we obtain the lower bound as a corollary of Theorem 4.10.

To specify the more general computation where D includes both 1×1 and 2×2 blocks, we define the matrix $W = DL^T$ and let S be the set of rows/columns corresponding to a 1×1 block of D. Then for $j \in S$, the computation of column j of L can be written similarly to Equation (4.6):

$$L_{ij} = (A_{ij} - \sum_{k<j} L_{ik} \cdot W_{kj})/D_{jj}, \tag{4.7}$$

for $i > j \in S$. For pairs of columns $j, j+1 \notin S$ corresponding to 2×2 blocks of D, we will use colon notation to describe the computation for pairs of elements:

$$L_{i,j:j+1} = (A_{i,j:j+1} - \sum_{k<j} L_{i,k} \cdot W_{k,j:j+1}) D^{-1}_{j:j+1,j:j+1}, \tag{4.8}$$

for $i > j + 1$.

Corollary 4.13. *The bandwidth cost lower bound for classical (dense or sparse) LDL^T factorization is $G/(\sqrt{8M}) - M - t$, where G is the number of scalar multiplications performed in computing L, t is the number of nonzeros in the matrix DL^T, and we assume the entries* *of DL^T are computed only once. In the special case of factoring a dense $n \times n$ matrix, this lower bound is $\Omega(n^3/\sqrt{M})$.*

Proof. We first verify that LDL^T is a 3NL computation, though with temporary operands. For simplicity, we consider the case of all 1×1 blocks, but the analysis follows for the case of 2×2 blocks using Equations (4.7) and (4.8). We let f_{ij} be the function defined in Equation (4.6) and g_{ijk} be the scalar multiplication of L_{ik} and W_{kj}. Then we make the correspondences that L_{ij} is stored at location $\mathfrak{c}(i,j) = \mathfrak{a}(i,j)$ and W_{ij} is stored at location $\mathfrak{b}(i,j)$. Since L is an input stored in slow/global memory, the mappings $\mathfrak{a}$ and $\mathfrak{c}$ are one-to-one into slow/global memory. Further, functions f_{ij} (involving a summation of g_{ijk} outputs) and g_{ijk} (scalar multiplication) depend non-trivially on their arguments. Thus, the computation is 3NL, with the exception that the mapping $\mathfrak{b}$ is not necessarily into slow/global memory, and the number of temporary operands is the number of nonzeros in the matrix $W = DL^T$.

In the case of dense $n \times n$ LDL^T factorization, the number of scalar multiplications required to compute L is $G = \Theta(n^3)$. □

See Sections 7.1.3 and 8.1.3 for a discussion of algorithms for this computation. No known sequential algorithm attains this bound for all matrix dimensions.

4.2.2.4 LTL^T Factorization

Another symmetric indefinite factorization computes a lower triangular matrix L and a symmetric tridiagonal matrix T such that $A = LTL^T$. Symmetric pivoting is required for numerical stability. Parlett and Reid [119] developed an algorithm for computing this factorization requiring approximately $(2/3)n^3$ flops, the same cost as LU factorization and twice the computational cost of Cholesky. Aasen [1] improved the algorithm and reduced the computational cost to $(1/3)n^3$, making use of a temporary upper Hessenberg matrix $H = TL^T$. Aasen's algorithm works by alternately solving for unknown values in the equations $A = LH$ and $H = TL^T$. Because the matrix H is integral to the computation but is a temporary matrix, we use Theorem 4.10 to obtain a communication lower bound.

In fact, the computation can be generalized to compute a symmetric band matrix T with bandwidth b (*i.e.*, b is the number of nonzero diagonals both below and above the main diagonal of T), in which case the matrix H has b nonzero subdiagonals. For uniqueness, the L matrix is set to have unit diagonal and the first b columns of L are set to the first b columns of the identity matrix. Because there are multiple ways to compute T and H, we specify a classical LTL^T computation in terms of the lower triangular matrix L:

$$L_{ij} = (A_{i,j-b} - \sum_{k=b+1}^{j-b} L_{ik} H_{k,j-b}) / H_{j,j-b} \quad \text{for} \quad b < j < i \leq n. \tag{4.9}$$

In the sparse case, the equations may be evaluated for some subset of indices (i,j) and the summations may be over some subset of the indices k. Equation (4.9) also assumes pivoting has already been incorporated in the interpretation of the indices i, j, and k.

Corollary 4.14. *The bandwidth cost lower bound for classical (dense or sparse) LTL^T factorization is $G/(8\sqrt{M})-M-t$, where G is the number of scalar multiplications performed in computing L, t is the number of nonzeros in the matrix TL^T, and we assume the entries of TL^T are computed only once. In the special case of factoring a dense $n \times n$ matrix (with T having bandwidth $b \ll n$), this lower bound is $\Omega(n^3/\sqrt{M})$.*

Proof. We first verify that LTL^T is a 3NL computation, though with temporary operands. We let f_{ij} be the function defined in Equation (4.9) and g_{ijk} be the scalar multiplication of L_{ik} and $H_{k,j-b}$. Then we make the correspondences that L_{ij} is stored at location $\mathfrak{c}(i,j) = \mathfrak{a}(i,j)$ and $H_{i,j-b}$ is stored at location $\mathfrak{b}(i,j)$. Since L is an input stored in slow/global memory, the mappings $\mathfrak{a}$ and $\mathfrak{c}$ are one-to-one into slow/global memory. Further, functions f_{ij} (involving a summation of g_{ijk} outputs) and g_{ijk} (scalar multiplication) depend non-trivially on their arguments. Thus, the computation is 3NL, with the exception that the mapping b is not necessarily into slow/global memory, and the number of temporary operands is the number of nonzeros in the matrix $H = TL^T$.

In the case of a dense $n \times n$ matrix, the number of scalar multiplications is $G = n^3/6 + O(n^2b)$ and the number of nonzeros in H is $n^2/2+O(nb)$. When $n \geq \sqrt{M}$, the $n^3/\sqrt{M}$ term asymptotically dominates the (negative) M and $O(n^2)$ terms. □

This bound is attainable in the sequential case by the algorithm presented in Chapter 9 (see Section 9.3.2 for details of the communication costs). See Sections 7.1.3 and 8.1.3 for more discussions of symmetric-indefinite algorithms.

4.2.2.5 Gram-Schmidt Orthogonalization

Here we consider the Gram-Schmidt process (both classical and modified versions) for orthogonalization of a set of vectors (see [57, Algorithm 3.1], for example). Given a set of vectors stored as columns of an $m \times n$ matrix A, the Gram-Schmidt process computes a QR decomposition, though the R matrix is sometimes not considered part of the output. For generality, we assume the R matrix is not a final output and use Theorem 4.10. Letting the columns of Q be the computed orthonormal basis, we specify the Gram-Schmidt computation in terms of the equation for computing entries of R. In the case of Classical Gram-Schmidt, we have

$$R_{ij} = \sum_{k=1}^{m} Q_{ki} A_{kj}, \tag{4.10}$$

and in the case of Modified Gram-Schmidt, we have

$$R_{ij} = \sum_{k=1}^{m} Q_{ki} Q_{kj}, \tag{4.11}$$

where Q_{kj} is the partially computed value of the jth orthonormal vector. In the sparse case, the equations may be evaluated for some subset of indices (i,j) and the summations may be over some subset of the indices k.

Corollary 4.15. *The bandwidth cost lower bound for (dense or sparse) QR factorization using classical or modified Gram-Schmidt orthogonalization is* $G/(8\sqrt{M}) - M - t$, *where* G *is the number of scalar multiplications performed in computing* R, t *is the number of nonzeros in* R, *and we assume the entries of* R *are computed only once. In the special case of orthogonalizing a dense* $m \times n$ *matrix with* $m \geq n$, *this lower bound is* $\Omega(mn^2/\sqrt{M})$.

Proof. We first verify that Gram-Schmidt orthogonalization is a 3NL computation, possibly with temporary operands. For the classical version, we let f_{ij} be the function defined in Equation (4.10) and g_{ijk} be the scalar multiplication of Q_{ki} and A_{kj}. For the modified version, we let f_{ij} be the function defined in Equation (4.11) and g_{ijk} be the scalar multiplication of Q_{ki} and Q_{kj}. Then we make the correspondences that Q_{ij} is stored at location $\mathfrak{a}(i,j)$, and either A_{ij} is stored at location $\mathfrak{b}(i,j)$ (in the classical version) or $\mathfrak{a} = \mathfrak{b}$ (in the modified version). Further, we let R_{ij} be stored at location $\mathfrak{c}(i,j)$. Since A is an input and Q is an output, the mappings $\mathfrak{a}$ and $\mathfrak{b}$ are one-to-one into slow/global memory. If R is an output of the computation, then $\mathfrak{c}$ is also one-to-one into slow/global memory; otherwise, we impose reads and writes on the temporary entries of R. Further, functions f_{ij} (involving a summation of g_{ijk} outputs) and g_{ijk} (scalar multiplication) depend non-trivially on their arguments. Thus, the computation is 3NL, with the exception that the mapping $\mathfrak{c}$ is not necessarily into slow/global memory, and the number of temporary operands is the number of nonzeros in R.

In the case of a dense $m \times n$ matrix, the number of scalar multiplications is $G = \Theta(mn^2)$ and the number of nonzeros in R is about $n^2/2$. When $m, n \geq \sqrt{M}$, the $mn^2/\sqrt{M}$ term asymptotically dominates the (negative) M and n^2 terms. □

4.3 Applying Orthogonal Transformations

The most stable and highest performing algorithms for QR decomposition are based on applying orthogonal transformations. Two-sided orthogonal transformations are used in the reduction step of the most commonly used approaches for solving eigenvalue and singular value problems: transforming a matrix to Hessenberg form for the nonsymmetric eigenproblem, tridiagonal form for the symmetric eigenproblem, and bidiagonal from for the SVD. In this section, we make two lower bound arguments in Sections 4.3.1 and 4.3.2, first in the context of one-sided orthogonal transformations (as in QR decomposition), and then in Section 4.3.3 we describe how to generalize the arguments for two-sided transformations. Each of the lower bound arguments requires certain assumptions on the algorithms. We discuss in Section 4.3.4 to which algorithms each of the lower bounds apply.

The case of applying orthogonal transformations is more subtle to analyze for several reasons: (1) there is more than one way to represent the orthogonal factor (*e.g.*, Householder reflections and Givens rotations), (2) the standard ways to reorganize or "block" transformations to reduce communication involve using the distributive law, not just summing terms in a different order [42, 122, 131], and (3) there may be many temporary operands that are not mapped to slow/global memory.

To be concrete, we consider Householder transformations, in which an elementary real orthogonal matrix Q_1 is represented as $Q_1 = I - \tau_1 u_1 u_1^T$, where u_1 is a column vector called a Householder vector and $\tau_1 = 2/\|u_1\|_2^2$. When applied from the left, a single Householder reflection Q_1 is chosen so that multiplying $Q_1 \cdot A$ annihilates selected rows in a particular column of A, and modifies one other row in the same column (accumulating the weight of the annihilated entries). We consider the Householder vector u_1 itself to be the output of the computation, rather than the explicit Q_1 matrix. Note that the Householder vector is nonzero only in the rows corresponding to annihilated entries and the accumulator entry.

We furthermore model the standard way of blocking Householder vectors, writing

$$Q_\ell \cdots Q_1 = I - U_\ell T_\ell U_\ell^T,$$

where $U_\ell = [u_1, u_2, \ldots, u_\ell]$ is n-by-ℓ and T_ℓ is ℓ-by-ℓ. We specify the application (from the left) of blocked Householder transformations to a matrix A by inserting parentheses as follows:

$$(I - U_\ell \cdot T_\ell \cdot U_\ell^T) \cdot A = A - U_\ell \cdot (T_\ell \cdot U_\ell^T \cdot A) = A - U_\ell \cdot Z_\ell,$$

defining $Z_\ell = T_\ell \cdot U_\ell^T \cdot A$. We also overwrite A with the output: $A := A - U_\ell \cdot Z_\ell$.

The application of one blocked transformation is a matrix multiplication (which is a 3NL computation though with some temporary operands), but in order to show an entire computation (like QR decomposition) is 3NL, we need a global indexing scheme to define the f_{ij} and g_{ijk} functions and $\mathfrak{a}$, $\mathfrak{b}$, and $\mathfrak{c}$ mappings. To that end, we let k be the index of the Householder vector, so that u_k is the kth Householder vector of the entire computation, and we let $U = [u_1, \ldots, u_h]$, where h is the total number of Householder vectors. We thus specify the application of orthogonal transformations (from the left) to a matrix A as follows:

$$A_{ij} = A_{ij} - \sum_{k=1}^{h} U_{ik} Z_{kj}, \tag{4.12}$$

where z_k (the kth row of Z) is a temporary quantity computed from A, u_k, and possibly other columns of U, depending on how Householder vectors are blocked. If A is $m \times n$, then U is $m \times h$ and Z is $h \times n$. Note that in the case of QR decomposition, it may be that $h \gg n$ (if one Householder vector is used to annihilate each entry below the diagonal, for example). The equations may be evaluated for some subset of indices (i, j) and the summations are over some subset of the indices k (even in the dense case). Equation (4.12) also assumes pivoting has already been incorporated in the interpretation of the indices i and j.

4.3.1 First Lower Bound Argument: Applying Theorem 4.10

Given the specification of applying orthogonal updates as a 3NL computation (with temporary operands), we can prove the following corollary of Theorem 4.10.

Corollary 4.16. *The bandwidth cost lower bound for applying orthogonal updates as specified in Equation* (4.12) *is* $G/(8\sqrt{M}) - M - t$*, where* G *is the number of scalar multiplications* *involved in the computation of UZ and t is the number of nonzeros in Z. In the special case of computing a QR decomposition of an $m \times n$ matrix using only a constant number of Householder vectors per column, this lower bound is $\Omega(mn^2/\sqrt{M})$.*

Proof. We first verify that applying orthogonal transformations is a 3NL computation, though with temporary operands. We let f_{ij} be the function defined in Equation (4.12) and g_{ijk} be the scalar multiplication of U_{ik} and Z_{kj}. Then we make the correspondences that U_{ij} is stored at location $\mathfrak{a}(i,j)$ and A_{ij} is stored at location $\mathfrak{c}(i,j)$. Since A is an input and U is an output, the mappings $\mathfrak{a}$ and $\mathfrak{c}$ are one-to-one into slow/global memory. If we let Z_{ij} be stored at location $\mathfrak{b}(i,j)$, then b may not map into slow/global memory. Further, functions f_{ij} (involving a summation of g_{ijk} outputs) and g_{ijk} (scalar multiplication) depend non-trivially on their arguments. Thus, the computation is 3NL, with the exception that the mapping b is not necessarily into slow/global memory, and the number of temporary operands is the number of nonzeros in Z.

In the case of a dense $m \times n$ matrix, the number of scalar multiplications is $G = \Theta(mn^2)$. If only a constant number of Householder vectors are used per column, then $h = \Theta(n)$ and the number of nonzeros in Z is $O(mh) = O(mn)$. When $m, n \gg \sqrt{M}$, the $mn^2/\sqrt{M}$ term asymptotically dominates the (negative) M and mn terms. $\square$

Note that in the sparse case where only one Householder vector per column is used, a separate argument can bound the number of nonzeros in Z in terms of the number of nonzeros in the input A and output U (see [28, Lemma 4.1]).

Many (but not all) algorithms for QR decomposition use only one Householder transformation per column. See Sections 7.1.5 and 8.1.5 for a discussion of such algorithms. However, for algorithms that use many Householder transformations per column, the bound in Corollary 4.16 can degenerate to zero because the number of temporary operands can be larger than the bandwidth cost guarantees of the 3NL argument.

4.3.2 Second Lower Bound Argument: Bounding Z Values

In order to devise a lower bound for algorithms that create many Z values, we reconsider the segment-based argument of Theorem 4.2. The assumption we are breaking in that argument is that one of the operands to each g_{ijk} function (the Z_{kj} entry) is a temporary value. In the argument of Section 4.3.1, we ignore how Z is computed and simply count the total number of nonzeros of Z. However, Z is defined in terms of U and A (and T, which itself depends only on U), and so entries of Z must be computed from entries of U and A. We use this property to bound not the overall number of Z entries, but the number of Z entries that are available in any given segment of the computation. Because a Z entry is computed from entries of U and A and updates an entry of A, in order for a Z entry to be computed, used as many times as necessary, and then discarded (without generating any memory traffic), all the relevant entries of U and A must be resident in fast memory. We use this observation to bound the number of Z entries available in any given segment, but this argument will require two new assumptions and some extra notation.

As a short-hand, we will sometimes refer to a matrix entry as being *treated as nonzero* (TAN) if the algorithm assumes that its value could be nonzero in deciding whether to bother performing g_{ijk}. Thus an algorithm for dense matrices treats all entries as nonzero, even if the input matrix is sparse, whereas a sparse factorization algorithm would not. We also introduce some notation:

- Let $U(k)$ be the kth column of U (which is the kth Householder vector). We will use $U(k)$ and $U(:, k)$ interchangeably when the context is clear.
- Let col_src_$U(k)$ be the index of the column in which $U(k)$ introduces zeros.
- Let rows_$U(k)$ be the set of indices of rows TAN in $U(k)$. Let row_dest_$U(k)$ be the index of the row in column col_src_$U(k)$ in which nonzero values in that column are accumulated by $U(k)$, and let zero_rows_$U(k)$ be rows_$U(k)$ with row_dest_$U(k)$ omitted.

4.3.2.1 Assumptions

We will make two central assumptions in this case. First, we assume that the algorithm does not block Householder updates (*i.e.*, all T matrices are 1×1). Second, we assume the algorithm makes "forward progress" which we define below. As explained later, forward progress is a natural property of most efficient implementations, precluding certain kinds of redundant work. See [27, Appendix A] for a motivating counterexample that breaks the assumption or Section 10.1.4 for a discussion of a useful algorithm that breaks the assumption and beats the lower bound.

The first assumption means that we are computing $\prod_k (I - \tau_k \cdot U(:, k) \cdot (U(:, k))^T) \cdot A$, where τ_k is scalar. This seems like a significant restriction, since blocked Householder transformations are widely used in practice. We do not believe this assumption is necessary for the communication lower bound to be valid, but it is necessary for our proof technique. This assumption yields a partial order (PO) in which the Householder updates must be applied to get the right answer. It is only a partial order because if, say, $U(:, k)$ and $U(:, k+1)$ do not "overlap", *i.e.*, have no common rows that are TAN, then $(I - \tau_k \cdot U(:, k) \cdot (U(:, k))^T)$ and $(I - \tau_{k+1} \cdot U(:, k+1) \cdot (U(:, k+1))^T)$ commute, and either one may be applied first (indeed, they may be applied independently in parallel).

Definition 4.17 (Partial Order on Householder vectors (PO))**.** *Suppose $k_1 < k_2$ and rows_$U(k_1) \cap$ rows_$U(k_2) \neq \{\emptyset\}$, then $U(k_1) < U(k_2)$ in the partial order.*

We note that this relation is transitive. That is, two Householder vectors $U(k_1)$ and $U(k_2)$ are partially ordered if there exists $U(k^*)$ such that $U(k_1) < U(k^*) < U(k_2)$, even if rows_$U(k_1) \cap$ rows_$U(k_2) = \{\emptyset\}$.

Our second assumption is that the algorithm makes forward progress:

Definition 4.18 (Forward Progress (FP)). *We say an algorithm which applies orthogonal transformations to zero out entries makes forward progress if the following two conditions hold:*

1. *an element that was deliberately zeroed out by one transformation is never again zeroed out or filled in by another transformation,*
2. *if*
   a) $U(k_1), \ldots, U(k_b) < U(\hat{k})$ *in* PO,
   b) $col\_src\_U(k_1) = \cdots = col\_src\_U(k_b) = c \neq \hat{c} = col\_src\_U(\hat{k})$,
   c) *and no other* $U(k_i)$ *satisfies* $U(k_i) < U(\hat{k})$ *and* $col\_src\_U(k_i) = c$,

   then

$$rows\_U(\hat{k}) \subset \bigcup_{i=1}^{b} zero\_rows\_U(k_i) \cup \{rows\ of\ column\ c\ that\ are\ TAZ\}\,. \tag{4.13}$$

The first condition holds for most efficient algorithms applying orthogonal transformations (though not all–see Chapter 10). By "deliberately," we mean the algorithm converted a TAN entry into a TAZ entry with an orthogonal transformation. The introduction of a zero due to accidental cancellation (such zero entries are still TAN) is not deliberate. We note also that *FP* is not violated if an original TAZ entry of the matrix is filled in (so that it is no longer TAZ); this is a common situation when doing sparse QR. It is easy to see that it is necessary to prove any nontrivial communication lower bound, since without it an algorithm could "spin its wheels" by repeatedly filling in and zeroing out entries, doing an arbitrary amount of arithmetic with no memory traffic at all.

The second condition holds for every correct algorithm for QR decomposition that does not violate the first condition. This condition means any later Householder transformation ($U(\hat{k})$) that depends on earlier Householder transformations ($U(k_1), ..., U(k_b)$) creating zeroes in a common column c may operate only "within" the rows zeroed out by the earlier Householder transformations. We motivate this assumption in [27, Appendix B] by showing that if an algorithm violates the second condition, it can "get stuck." This means that it cannot achieve triangular form without filling in a deliberately created zero.

4.3.2.2 Roots and Destinations

In order to reason more directly about temporary operands, we categorize in more detail the data available in fast memory for 3NL computation. To this end, we consider each input or output operand of g_{ijk} functions that appears in fast memory during a segment of M slow memory operations. It may be that an operand appears in fast memory for a while, disappears, and reappears, possibly several times. For each period of continuous existence of an operand in fast memory, we label its *Root* (how it came to be in fast memory) and its *Destination* (what happens when it disappears):

- *Root R1*: The operand was already in fast memory at the beginning of the segment, and/or read from slow memory. There are at most $2M$ such operands altogether, because the fast memory has size M, and because a segment contains at most M reads from slow memory.
- *Root R2*: The operand is computed (created) during the segment. Without more information, there is no bound on the number of such operands.
- *Destination D1*: An operand is left in fast memory at the end of the segment (so that it is available at the beginning of the next one), and/or written to slow memory. There are at most $2M$ such operands altogether, again because the fast memory has size M, and because a segment contains at most M writes to slow memory.
- *Destination D2*: An operand is *neither* left in fast memory nor written to slow memory, but simply discarded. Again, without more information, there is no bound on the number of such operands.

We may correspondingly label each period of continuous existence of any operand in fast memory during one segment by one of four possible labels Ri/Dj, indicating the Root and Destination of the operand at the beginning and end of the period. Based on the above description, the total number of operands of all types except R2/D2 is bounded by $4M$ (the maximum number of R1 operands plus the number of D1 operands, an upper bound). The R2/D2 operands, those created during the segment and then discarded without causing any slow memory traffic, cannot be bounded without further information.

4.3.2.3 Bounding Z Values

With the assumptions of Section 4.3.2.1, we begin the argument to bound from below the number of memory operations required to apply the set of Householder transformations. As in the proof of Theorem 4.2, we will focus our attention on an arbitrary segment of computation in which there are $O(M)$ non-R2/D2 entries in fast memory. Our goal will be to bound the number of multiplications in a segment involving R2/D2 entries, since the number of remaining multiplications can be bounded as before. From here on, let us denote by $Z_2(k,j)$ the element $Z(k,j)$ if it is R2/D2, and by $Z_n(k,j)$ if it is non-R2/D2. We will further focus our attention within the segment on the update of an arbitrary column of the matrix, $A(:,j)$.

Each $Z(k,j)$ in memory is associated with one Householder vector $U(:,k)$ which will update $A(:,j)$. We will denote the associated Householder vector by $U_2(:,k)$ if $Z(k,j) = Z_2(k,j)$ is R2/D2 and $U_n(:,k)$ if $Z(k,j) = Z_n(k,j)$ is non-R2/D2. With this notation, we have the following two lemmas which make it easier to reason about what happens to $A(:,j)$ during a segment.

Lemma 4.19. *If $Z_2(k,j)$ is in memory during a segment, then $U_2(:,k)$ as well as the entries $A(rows\_U(k),j)$ are in memory during the segment.*

Proof. Since $Z_2(k, j)$ is discarded before the end of the segment and may not be re-computed later, the entire $A(:, j) = A(:, j) - U(:, k) \cdot Z_2(k, j)$ computation has to end within the segment. Thus, all entries involved must be resident in memory. □

However, even if a $Z_n(k, j)$ is in memory during a segment, the $U_n(:, k) \cdot Z_n(k, j)$ computation will possibly not be completed during the segment, and therefore the $U_n(:, k)$ vector and corresponding entries of $A(:, j)$ may not be completely represented in memory.

Lemma 4.20. *If $Z_2(k_1, j)$ and $Z_2(k_2, j)$ are in memory during a segment, and $U(k_1) < U(k) < U(k_2)$ in the* PO*, then $Z(k, j)$ must also be in memory during the segment.*

Proof. This follows from our first assumption that all T matrices are 1×1 and the partial order is imposed. Since $U(k_1) < U(k)$, $Z(k, j)$ cannot be fully computed before the segment. Since $U(k) < U(k_2)$, $U(:, k) \cdot Z(k, j)$ has to be performed in the segment too, at least enough to carry the dependency, so $Z(k, j)$ cannot be fully computed after the segment. That is, if $U(:, k)$ is $U_n(:, k)$, not all *rows_U*(k) rows of $A(:, j)$ must be updated, but enough for $Z_2(k_2, j)$ to be computed and $U_2(:, k_2) \cdot Z_2(k_2, j)$ to be applied correctly. Thus, $Z(k, j)$ is computed during the segment and therefore must exist in memory. Note that a partial sum of $(U(:, k))^T \cdot A(:, j)$ may have been computed before the beginning of the segment and used in the segment to compute $Z_n(k, j)$, but the final $Z_n(k, j)$ value cannot be computed until the segment. □

Roughly speaking, our goal now is to bound the number of $U_2(r, k) \cdot Z_2(k, j)$ multiplications by the number of multiplications in a different matrix multiplication $\widehat{U} \cdot \widehat{Z}$ where we can bound the number of $\widehat{U}$ entries by the number of U entries in memory, and bound the number of $\widehat{Z}$ entries by the number of A entries plus the number of Z_n entries in memory, which lets us use the geometric bound of Lemma 2.7.

Given a particular segment and column j, we construct $\widehat{U}$ by first partitioning the $U_2(:, k)$ by their col_src_$U(k)$ and then collapsing each partition into one column of $\widehat{U}$. Likewise, collapse $Z(:, j)$ by partitioning its rows corresponding to the partitioned columns of U and taking the union of TAN entries in each set of rows to be the TAN entries of the corresponding row of $\widehat{Z}(:, j)$. More formally,

Definition 4.21 ($\widehat{U}$ and $\widehat{Z}$)**.** *For a given segment of computation and column j of A, we set $\widehat{U}(r, c)$ to be TAN if there exists a $U_2(:, k)$ in fast memory such that $c =$ col_src_$U(k)$ and $r \in$ rows_$U(k)$. We set $\widehat{Z}(c, j)$ to be TAN if there exists a $Z_2(k, j)$ in fast memory such that $c =$ col_src_$U(k)$.*

We will "emulate" the computation $A(:, j) = A(:, j) - \sum U_2(:, k) \cdot Z_2(k, j)$ with the related computation $A(:, j) = A(:, j) - \sum \widehat{U}(:, c) \cdot \widehat{Z}(c, j)$ in the following sense: we will show that the number of multiplications done by $U_2(:, k) \cdot Z_2(k, j)$ is within a factor of 2 of the number of multiplications done by $\widehat{U}(:, c) \cdot \widehat{Z}(c, j)$.

The following example illustrates this construction on a small matrix, where K_2 contains three indices (*i.e.*, there are three Householder vectors that were computed to zero entries in the second column of A); just TAN patterns are shown.

$$U(:,K_2) = \begin{bmatrix} & & \\ & & \bullet \\ & & \\ & \bullet & \bullet \\ \bullet & \bullet & \\ & & \\ \bullet & & \\ \bullet & & \\ \bullet & & \end{bmatrix} \quad \Rightarrow \quad \widehat{U}(:,2) = \begin{bmatrix} \\ \bullet \\ \\ \bullet \\ \bullet \\ \\ \bullet \\ \bullet \\ \bullet \end{bmatrix}$$

Note that we do not care what the TAN values of $\widehat{U}$ and $\widehat{Z}$ are; this computation has no hope of getting a correct result because the rank of $\widehat{U} \cdot \widehat{Z}$ is generally less than the rank of the subset of $U \cdot Z$ it replaces. We emulate in this way only to count the memory traffic. We establish the following results with this construction.

Lemma 4.22. *$\widehat{U}(:,c)$ has at least half as many TAN entries, and at most as many TAN entries, as the columns of U from which it is formed.*

Proof. The sets zero_rows_$U(k)$ for k in a partition (*i.e.*, with the same col_src_$U(k)$) must be disjoint by the forward progress assumption, and there are at least as many of these rows as in all the corresponding row_dest_$U(k)$, which could potentially all coincide. By Lemma 4.19, we know that complete $U_2(:,k)$ are present (otherwise they could, for example, all be Givens transformations with the same destination row, and if zero rows were not present, they would all collapse into one row). And so since every entry of zero_rows_$U(k)$ contributes to a TAN entry of $\widehat{U}(:,c)$, and zero_rows_$U(k)$ constitutes at least half of the TAN entries of $U(k)$, $\widehat{U}(:,c)$ has at least half as many TAN entries as the corresponding columns of U.

If all the $U_2(:,k)$ being collapsed have TAN entries in disjoint sets of rows, then $\widehat{U}(:,c)$ will have as many entries TAN as all the $U(:,k)$. □

Because each TAN entry of $U(:,k)$ contributes one scalar multiplication to $A(:,j) = A(:,j) - \sum U_2(:,k) \cdot Z_2(k,j)$ and each TAN entry of $\widehat{U}(:,c)$ contributes one scalar multiplication to $A(:,j) = A(:,j) - \sum \widehat{U}(:,c) \cdot \widehat{Z}(c,j)$, we have the following corollary.

Corollary 4.23. *$\widehat{U}(:,c) \cdot \widehat{Z}(c,j)$ does at least half as many multiplications as all the corresponding $U_2(:,k) \cdot Z_2(k,j)$.*

In order to bound the number of $\widehat{U} \cdot \widehat{Z}$ multiplications in the segment, we must also bound the number of $\widehat{Z}$ entries available.

Lemma 4.24. *The number of TAN entries of $\widehat{Z}(:,j)$ is bounded by the number of $A(:,j)$ entries plus the number of $Z_n(:,j)$ entries resident in memory.*

Proof. Our goal is to construct an injective mapping $\mathcal{I}$ from the set of of $\widehat{Z}(:,j)$ entries to the union of the sets of $A(:,j)$ and $Z_n(:,j)$ entries. Consider the set of $Z(k,j)$ entries (both R2/D2 and non-R2/D2) in memory as vertices in a graph G. Each vertex has a unique label k (recall that j is fixed), and we also give each vertex two more non-unique labels: 2 or n to denote whether the vertex is $Z_2(k,j)$ or $Z_n(k,j)$ and col_src_$U(k)$ to denote the column source of the corresponding Householder vector. A directed edge (k_1,k_2) exists in the graph if $U(:,k_1) < U(:,k_2)$ in the *PO*. Note that all the vertices labeled both 2 and c are $Z_2(k,j)$ that lead to $\widehat{Z}(c,j)$ being TAN in Definition 4.21.

For all values of $c = \text{col\_src\_}U(k)$ appearing as labels in G, in order of which node labeled c is earliest in *PO* (not necessarily unique), find a (not necessarily unique) node k with label col_src_$U(k) = c$, that has no successors in G with the same label c. If this node is also labeled n, then we let $\mathcal{I}$ map $\widehat{Z}(c,j)$ to $Z_n(k,j)$. If node k is labeled 2, then we let $\mathcal{I}$ map $\widehat{Z}(c,j)$ to $A(\text{row\_dest\_}U(k),j)$. By Lemma 4.19, this entry of A must be in fast memory.

We now argue that this mapping $\mathcal{I}$ is injective. The mapping into the set of $Z_n(k,j)$ entries is injective because each $\widehat{Z}(c,j)$ can be mapped only to an entry with column source c. Suppose the mapping into the $A(:,j)$ entries is not injective, and let $\widehat{Z}(c,j)$ and $\widehat{Z}(\hat{c},j)$ be the entries which are both mapped to some $A(r,j)$. Then there are entries $Z_2(k,j)$ and $Z_2(\hat{k},j)$ such that $c = \text{col\_src\_}U(k)$, $\hat{c} = \text{col\_src\_}U(\hat{k})$, $r = \text{row\_dest\_}U(k) = \text{row\_dest\_}U(\hat{k})$, and neither k nor $\hat{k}$ have successors in G with the same column source label.

Since rows_$U(k)$ and rows_$U(\hat{k})$ intersect, they must be ordered with respect to the *PO*, so suppose $U(k) < U(\hat{k})$. Consider the second condition of *FP*. In this case, premises (2a) and (2b) hold, but the conclusion (4.13) does not. Thus, premise (2c) must not hold, so there exists another Householder vector $U(k^*)$ such that $c = \text{col\_src\_}U(k^*)$ and $r \in \text{zero\_rows\_}U(k^*)$.

Again, because their nonzero row sets intersect, each of these Householder vectors must be partially ordered. By the first condition of *FP*, since $\text{row\_dest\_}U(k) \in \text{zero\_rows\_}U(k^*)$, we have $U(k) < U(k^*)$. Also, since $U(k^*)$ satisfies (2a), we have $U(k^*) < U(\hat{k})$. Thus, $U(k) < U(k^*) < U(\hat{k})$, and by Lemma 4.20, $Z(k^*,j)$ must also be in fast memory and therefore in G. Since $Z(k^*,j)$ is a successor of $Z(k,j)$ in G, we have a contradiction. □

Theorem 4.25. *An algorithm which applies orthogonal transformations to annihilate matrix entries, does not compute T matrices of dimension 2 or greater for blocked updates, maintains forward progress as in Definition 4.18, and performs G flops of the form $U \cdot Z$ (as defined in Equation* (4.12)*), has a bandwidth cost of at least $\Omega(G/\sqrt{M}) - M$ words. In the special case of a dense m-by-n matrix with $m \geq n$, this lower bound is $\Omega(mn^2/\sqrt{M})$.*

Proof. We first argue that the number of A, U, and Z_n entries available during a segment are all $O(M)$.

Every $A(i,j)$ operand is destined either to be output (*i.e.*, D1) or converted into a Householder vector. Every $A(i,j)$ operand is either read from memory (*i.e.*, R1) or created on the fly due to sparse fill-in. So the only possible R2/D2 operands from A are entries which are filled in and then immediately become Householder vectors, and hence become R2 operands of U. We bound the number of these as follows.

All U operands are eventually output, as they compose Q. So there are no D2 operands of U (recall that we may compute each result $U(i,k)$ only once, so it cannot be discarded). So all R2 operands $U(i,k)$ are also D1, and so there are at most $2M$ of them (since at most M can remain in fast memory, and at most M can be written to slow memory, by the end of the segment). This also bounds the number of R2/D2 operands $A(i,j)$, and so bounds the total number of $A(i,j)$ operands by $6M$ (the sum of $2M$ = maximum number of D1 operands plus $2M$ = maximum number of R1 operands plus $2M$ = maximum number of R2/D2 operands).

The number of Z_n entries available in a segment is bounded by $2M$ because by definition, all entries are non-R2/D2.

From Lemma 4.22, the number of $\widehat{U}$ entries available is $O(M)$ because it is bounded by the number of U_2 entries which is in turn bounded by the number of U entries. From Lemma 4.24, the number of $\widehat{Z}$ entries available is $O(M)$ because it is bounded by the sum of the number of entries of A and of Z_n.

Thus, since the number of entries of each operand available in a segment are $O(M)$, by Lemma 2.7, the number of $\widehat{U} \cdot \widehat{Z}$ scalar multiplications is bounded by $O\left(M^{3/2}\right)$. By Corollary 4.23, the number of $U \cdot Z$ scalar multiplications within a segment is also bounded by $O\left(M^{3/2}\right)$.

Since there are $O(M)$ $Z_n(k,j)$ operands in a segment, the Loomis-Whitney argument bounds the number of multiplies involving such operands by $O(M^{3/2})$, so with the above argument that bounds the number of multiplies involving R2/D2 $Z(k,j)$ operands, the total number of multiplies involving both R2/D2 and non-R2/D2 Z entries is $O\left(M^{3/2}\right)$.

The rest of the proof is similar to before: a lower bound on the number of segments is then $\lfloor G/O\left(M^{3/2}\right)\rfloor \geq G/O\left(M^{3/2}\right) - 1$, so a lower bound on the number of slow memory accesses is $M \cdot \lfloor G/O\left(M^{3/2}\right)\rfloor \geq \Omega\left(G/M^{1/2}\right) - M$. For dense m-by-n matrices with $m \geq n$, the conventional algorithm does $G = \Theta(mn^2)$ multiplies. □

See Sections 7.1.5 and 8.1.5 for algorithms that satisfy the assumptions and attain this bound for dense matrices.

4.3.2.4 Counting Arithmetic Operations

It is natural to wonder whether the G operations in Theorem 4.25 (and Corollary 4.16) capture a constant fraction of the arithmetic operations performed by the algorithm, which would allow us to deduce that the lower bound is asymptotically as large as possible. The G operations are just the multiplications in all the different applications of block Householder transformations $A := A - U \cdot Z$, where $Z = T \cdot U^T \cdot A$. We argue that under a natural "genericity assumption" this constitutes a large fraction of all the multiplications in the algorithm (although this is not necessary for our lower bound to be valid). Suppose $(U^T \cdot$

$A)(k,j)$ is nonzero; the amount of work to compute this is at most proportional to the total number of entries stored (and so treated as nonzeros) in column k of U. Since T is triangular and nonsingular, this means $Z(k,j)$ will be generically nonzero as well and will be multiplied by column k of U and added to column j of A, which costs at least as much as computing $(U^T \cdot A)(k,j)$. The cost of the rest of the computation, forming and multiplying by T and computing the actual Householder vectors, are lower order terms in practice; the dimension of T is generally chosen small enough by the algorithm to try to assure this.

4.3.3 Generalizing to Eigenvalue and Singular Value Reductions

Standard algorithms for computing eigenvalues and eigenvectors, or singular values and singular vectors (the SVD), start by applying orthogonal transformations to both sides of A to reduce it to a "condensed form" (Hessenberg, tridiagonal or bidiagonal) with the same eigenvalues or singular values, and simply related eigenvectors or singular vectors [57]. We can extend our argument for one-sided orthogonal transformations to these computations. We can have some arbitrary interleaving of (block) Householder transformations applied on the left,

$$A = (I - U_L \cdot T_L \cdot U_L^T) \cdot A = A - U_L \cdot (T_L \cdot U_L^T \cdot A) = A - U_L \cdot Z_L,$$

where we define $Z_L = T_L \cdot U_L^T \cdot A$, and the right,

$$A = A \cdot (I - U_R \cdot T_R \cdot U_R^T) = A - (A \cdot U_R \cdot T_R) \cdot U_R^T = A - Z_R \cdot U_R^T,$$

where we define $Z_R = A \cdot U_R \cdot T_R$. Combining these, we can index the computation by Householder vector number similarly to Equation (4.12):

$$A(i,j) = A(i,j) - \sum_{k_L} U_L(i,k_L) \cdot Z_L(k_L,j) - \sum_{k_R} Z_R(i,k_R) \cdot U_R(j,k_R) \tag{4.14}$$

Of course there are lots of possible dependencies ignored here, much as we wrote down a similar formula for one-sided transformations. At this point we can apply either of the two lower bound arguments from before: we can either assume (1) the number of Householder vectors is small so that the number of temporary Z_L and Z_R values are bounded, applying Theorem 4.10, or (2) all T matrices are 1×1 and we make "forward progress," using the argument in the proof of Theorem 4.25. In case (1) we obtain a similar result to Corollary 4.16:

Corollary 4.26. *The bandwidth cost lower bound for applying two-sided orthogonal updates is $G/(8\sqrt{M}) - M - t$, where G is the number of scalar multiplications involved in the computation of $U_L Z_L$ and $Z_R U_R$ (as specified in Equation (4.14)) and t is the number of nonzeros in Z_L and Z_R. In the special case of reducing an $m \times n$ matrix to bidiagonal form using only a constant number of Householder vectors per row and column, this lower bound is $\Omega(mn^2/\sqrt{M})$. In the special case of reducing an $n \times n$ matrix to tridiagonal or Hessenberg form using only a constant number of Householder vectors per column, this lower bound is $\Omega(n^3/\sqrt{M})$.*

In case (2), the second lower bound argument requires a little more discussion to clarify the definitions of the partial order (Definition 4.17) and forward progress (Definition 4.18). There will be two partial orders, one for U_L and one for U_R. In parts 1 and 2 of Definition 4.18, we insist that no transformation (from left or right) fills in or re-zeros out an entry deliberately zeroed out by another transformation (left or right). This implies that there is an ordering between left and right transformations, but we do not need to use this order for our counting argument. We also insist that part 3 of Definition 4.18 holds independently for the left and for the right transformations.

With these minor changes, we see that the lower bound argument of Section 4.3.1 applies independently to $U_L \cdot Z_L$ and $Z_R \cdot U_R^T$. In particular, insisting that left (right) transformations cannot fill in or re-zeros out entries deliberately zeroed out by right (left) transformations means that number of arithmetic operations performed by the the left and right transformations can be bounded independently and added. This leads to the same lower bound on the number of words moved as before (in a Big-Oh sense):

Theorem 4.27. *An algorithm which applies two-sided orthogonal transformations to annihilate matrix entries, does not compute T matrices of dimension 2 or greater for blocked updates, maintains forward progress as in Definition 4.18, and performs G flops of the form $U \cdot Z$ (as defined in Equation* (4.14)*), has a bandwidth cost of at least $\Omega(G/\sqrt{M}) - M$ words. In the special case of reducing a dense m-by-n matrix to bidiagonal form with $m \geq n$, this lower bound is $\Omega(mn^2/\sqrt{M})$. In the special case of reducing an $n \times n$ matrix to tridiagonal or Hessenberg form, this lower bound is $\Omega(n^3/\sqrt{M})$.*

4.3.4 Applicability of the Lower Bounds

While we conjecture that all classical algorithms for applying one- or two-sided orthogonal transformations are subject to a lower bound in the form of Theorem 4.2, not all of those algorithms meet the assumptions of either of the two lower bound arguments presented in this section. However, many standard and efficient algorithms do meet the criteria; see Chapters 7 and 8 for a full discussion of these algorithms.

For example, algorithms for QR decomposition that satisfy this assumption of Corollary 4.16 include the blocked, right-looking algorithm (currently implemented in (Sca)LAPACK [8, 44]) and the recursive algorithm of Elmroth and Gustavson [66]. The simplest version of Communication-Avoiding QR (*i.e.*, one that does not block transformations, see last paragraph in Section 6.4 of [62]) satisfies the assumptions of Theorem 4.25. However, most practical implementations of CAQR do block transformations to increase efficiency in other levels of the memory hierarchy, and neither proof applies to these algorithms. The recursive QR decomposition algorithm of Frens and Wise [69] is also communication efficient, but again our proofs do not apply.

Further, Corollary 4.26 applies to the conventional blocked, right-looking algorithms in LAPACK [8] and ScaLAPACK [44] for reduction to Hessenberg, tridiagonal and bidiagonal forms. Our lower bound also applies to the first phase of the successive band reduction algorithm of Bischof, Lang, and Sun [38, 43], namely reduction to band form, because this satisfies our requirement of forward progress. However, the second phase of successive band reduction does not satisfy our requirement of forward progress, because it involves *bulge chasing*, which repeatedly creates nonzero entries outside the band and zeroes them out again. But since the first phase does asymptotically more arithmetic than the second phase, our lower bound based just on the first phase cannot be much improved (see Chapter 10 for more discussion of these and other algorithms).

There are many other eigenvalue computations to which these results may apply. For example, the lower bound applies to reduction of a matrix pair (A, B) to upper Hessenberg and upper triangular form. This is done by a QR decomposition of B, applying Q^T to A from the left, and then reducing A to upper Hessenberg form while keeping B in upper triangular form. Assuming one set of assumptions applies to the algorithm used for QR decomposition, the lower bound applies to the first two stages and reducing A to Hessenberg form. However, since maintaining triangular form of B in the last stage involves filling in entries of B and zeroing them out again, our argument does not directly apply. This computation is a fraction of the total work, and so this fact would not change the lower bound in an asymptotic sense.

4.4 Attainability

In this chapter, we obtain lower bounds for many different computations—most of classical linear algebra. The greatest value in establishing these lower bounds is that it sets a target for algorithmic development. For a given computation, we can determine if the best algorithms attain the lower bounds (and are asymptotically optimal), or we can identify a gap between algorithms (upper bounds) and lower bounds. In this case, tighter theoretical analysis may allow for higher lower bounds, or algorithmic innovation may lead to more efficient approaches to solving the problem. Once a communication-optimal algorithm is identified, we know that significant algorithmic improvements are no longer possible, and we can focus our attention on tuning the implementations to particular hardware platforms to maximize actual performance. We will discuss state-of-the-art algorithms and the attainability of the lower bounds presented here in Chapters 7 and 8.

Chapter 5

Lower Bounds for Strassen's Matrix Multiplication

While the approach for proving communication lower bounds discussed in Chapter 4 works for many algorithms in linear algebra, it no longer applies when distributivity is used, as in the case of Strassen's matrix multiplication algorithm. In this case, we consider the computation directed acyclic graph (CDAG) of an algorithm. While our approach of computation graph analysis is similar to the red-blue pebble game of Hong and Kung [88], we connect the communication costs of an algorithm to the *expansion* properties of its computation graph.

The expansion of a graph relates the number of vertices in a subset of the graph to its neighbors in the complement; see Definition 2.8 for a more rigorous definition. In the case of a computation graph, the vertices correspond to arithmetic operations, and the edges (particularly the ones between a given subset of vertices and its complement) correspond to communication. The analysis of the expansion properties of Strassen's CDAG relies on the recursive property of the algorithm and can be extended to other recursive algorithms. Indeed, other fast algorithms for matrix multiplication are recursive, and we obtain lower bound results for many of those algorithms (see Chapter 6).

This chapter is organized as follows. In Section 5.1 we discuss the relationship between the expansion of an algorithm's CDAG and its communication costs. We analyze the CDAG for Strassen's algorithm in particular in Section 5.2, and in Section 5.3 we combine these results in the form of a communication lower bound for Strassen's algorithm.

The main contributions of this chapter are

- introducing a new proof technique of relating communication lower bounds to a computation's dependency graph (CDAG) and
- using the technique to prove a communication lower bound for Strassen's matrix multiplication algorithm.

The results and proofs in this chapter appear in [25] (conference version), which was awarded the Best Paper prize at the ACM Symposium on Parallelism in Algorithms and

Architectures (SPAA) in 2011, and [26] (journal version), written with coauthors James Demmel, Olga Holtz, and Oded Schwartz.

5.1 Relating Edge Expansion to Communication

In this section we recall the notion of the computation graph of an algorithm, then show how a partition argument connects the expansion properties of the computation graph to the communication requirements of the algorithm. The partition argument is the same as the one used in Chapter 4.

5.1.1 Computation Graph

For a given algorithm, we let $G = (V, E)$ be the directed acyclic graph corresponding to the computation (CDAG), where there is a vertex for each input element and each arithmetic operation *(AO)* performed. The graph G contains a directed edge (u, v), if the output operand of the AO corresponding to u (or the input element corresponding to u), is an input operand to the AO corresponding to v. The in-degree of any vertex of G is, therefore, at most 2 (as the arithmetic operations are binary). The out-degree is, in general, unbounded (*i.e.*, it may be a function of $|V|$). We next show how an expansion analysis of this graph can be used to obtain a communication lower bound for the corresponding algorithm.

5.1.2 Partition Argument

Let M be the size of the fast memory. Let O be any total ordering of the vertices that respects the partial ordering of the CDAG G (this total ordering corresponds to the actual order in which the computations are performed). Let P be any partition of V into segments $S_1, S_2, \ldots$, so that a segment $S_i \in P$ is a subset of the vertices that are contiguous in the total ordering O.

For each segment S, let R_S and W_S be the set of read and write operands, respectively (see Figure 5.1). Namely, R_S is the set of vertices outside S that have an edge going into S, and W_S is the set of vertices in S that have an edge going outside S. Then the total number of reads of AOs to perform the computation in S is at least $|R_S| - M$, as at most M of the needed $|R_S|$ operands are already in fast memory when the execution of the segment starts. Similarly, S causes at least $|W_S| - M$ actual write operations, as at most M of the operands needed by other segments are left in the fast memory when the execution of the segment ends. The total bandwidth cost is therefore bounded below by

$$W \quad \geq \quad \max_P \sum_{S \in P} \left(|R_S| + |W_S| - 2M \right). \tag{5.1}$$

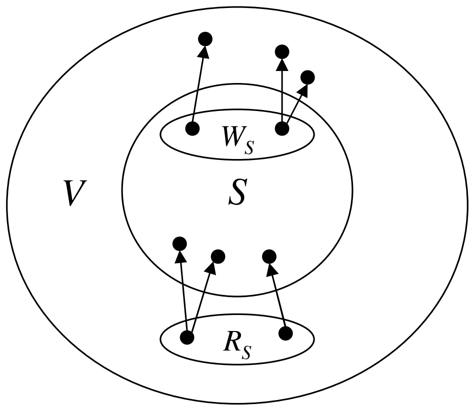


Figure 5.1: A segment of computation S and its corresponding read operands R_S and write operands W_S.

5.1.3 Edge Expansion and Communication

Recall the definition of edge expansion (Definition 2.8). We can use the edge expansion of a CDAG to relate a segment S to its read and write operands:

Claim 5.1. *Consider a segment S and its read and write operands R_S and W_S. If the graph G (with edge directions ignored) containing S has edge expansion $h(G)$, maximum degree d, and at least $2|S|$ vertices, then we have $|R_S| + |W_S| \geq h(G) \cdot |S|$.*

Proof. We have $|E(S, V \setminus S)| \geq h(G) \cdot d \cdot |S|$. Since $E(S, V \setminus S) = E(R_S, S) \uplus E(W_S, V \setminus S)$ we have $|E(S, V \setminus S)| = |E(R_S, S)| + |E(W_S, V \setminus S)| \leq d \cdot |R_S| + d \cdot |W_S|$ where the last inequality is by the degree bound. The claim follows. $\square$

Combining Claim 5.1 with Equation (5.1) and choosing to partition V into $|V|/s$ segments of equal size s, we obtain:

$$W \geq \max_{s \leq \frac{|V|}{2}} \frac{|V|}{s} \cdot (h(G) \cdot s - 2M) = \Omega\left(|V| \cdot h(G)\right),$$

for sufficiently large $|V|$. In many cases $h(G)$ is too small to attain the desired bandwidth cost lower bound. Typically, $h(G)$ is a decreasing function in $|V(G)|$ (*i.e.*, the edge expansion deteriorates as the input size and number of arithmetic operations increase). This is the case with Strassen's matrix multiplication algorithm. In such cases, it is better to consider the expansion of G on small sets (see Equation (2.2)):

$$W \geq \max_{s \leq \frac{|V|}{2}} \frac{|V|}{s} \cdot (h_s(G) \cdot s - 2M).$$

Choosing the minimal s so that

$$h_s(G) \cdot s \geq 3M \tag{5.2}$$

we obtain

$$W \geq \frac{|V|}{s} \cdot M. \tag{5.3}$$

The existence of a value $s \leq \frac{|V|}{2}$ that satisfies condition (5.2) is not always guaranteed. In the next section we confirm the existence of such s for Strassen's CDAG, for sufficiently large $|V|$. Indeed this is the interesting case, as otherwise all computations can be performed inside the fast memory, with no communication, except for reading the input once.

In some cases, the computation graph G does not fit these assumptions: it may not be regular, it may have vertices of unbounded degree, or its edge expansion may be hard to analyze. In such cases, we may consider some subgraph G' of G instead to obtain a lower bound on the bandwidth cost:

Claim 5.2. *Let $G = (V, E)$ be a computation graph of an algorithm Alg, and let $G' = (V', E')$ be a subgraph of G,* i.e., *$V' \subseteq V$ and $E' \subseteq E$. If G' is d-regular and $\alpha = \frac{|V'|}{|V|}$, then, for sufficiently large $|V'|$, the bandwidth cost of Alg is*

$$W \geq \frac{\alpha}{2} \cdot \frac{|V|}{s} \cdot M$$

where s is chosen so that $h_s(G') \cdot \alpha s \geq 3M$.

Proof. The correctness of this claim follows from Equations (5.2) and (5.3), and from the fact that at least an $\alpha/2$ fraction of the segments have at least $\alpha \cdot s$ of their vertices in G' (otherwise $V' < \frac{\alpha}{2} \cdot \frac{V}{s} \cdot s + (1 - \frac{\alpha}{2}) \cdot \frac{V}{s} \cdot \frac{\alpha}{2} s < \alpha V$). If we partition G into segments of size s, consider only those segments with at least $\alpha \cdot s$ of their vertices in G'. The expansion of G' guarantees that there are at least $h_s(G') \cdot \alpha s \geq 3M$ edges in E' that connect vertices in the segment to its complement. Since there at least $\alpha/2$ such segments, the result follows. $\square$

5.2 Expansion Properties of Strassen's Algorithm

Recall Strassen's algorithm for matrix multiplication (see Section 2.4.1) and consider its computation graph (see Figure 5.2). Let H_i be the computation graph of Strassen's algorithm for recursion depth i, so that $H_{\lg n}$ corresponds to the computation for input matrices of size $n \times n$. Then $H_{\lg n}$ has the following structure:

- Encode A: generate weighted sums of elements of A (this corresponds to the left factors of lines 5-11 of the algorithm).

- Similarly encode B (this corresponds to the right factors of lines 5-11 of the algorithm).

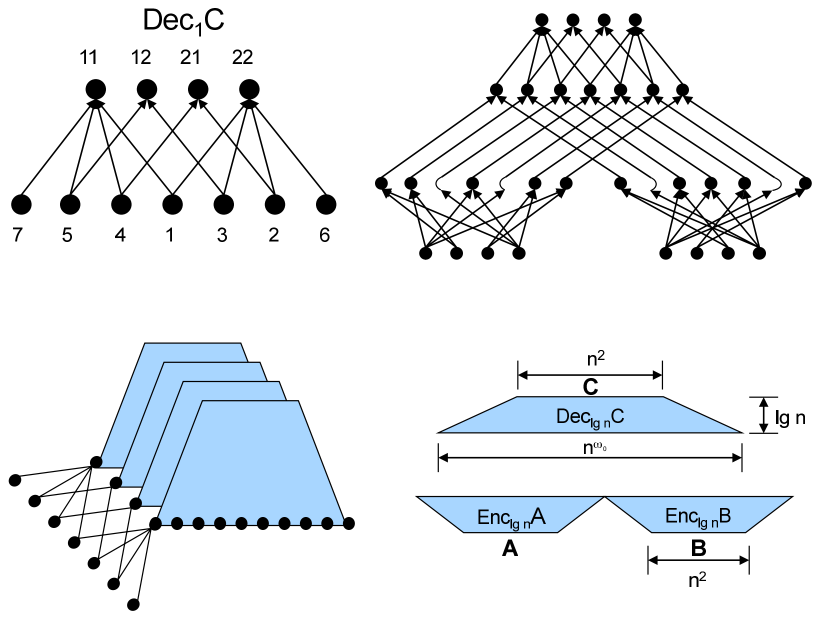


Figure 5.2: The computation graph of Strassen's algorithm (see Section 2.4.1). Top left: Dec_1C. Top right: H_1, with Dec_1C on the top and Enc_1A and Enc_1B at the bottom. Bottom left: $Dec_{\lg n}C$. Bottom right: $H_{\lg n}$.

- Then multiply the encodings of A and B element-wise (this corresponds to line 2 of the algorithm).
- Finally, decode C, by taking weighted sums of the products (this corresponds to lines 12-15 of the algorithm).

We let Enc_iA, Enc_iB, and Dec_iC be the subgraphs corresponding to the encoding and decoding of $2^i \times 2^i$ matrices A, B, and C, respectively. Note that Dec_1C is presented in Figure 5.2, for simplicity, with vertices of in-degree larger than two (but constant). A vertex of degree larger than two, in fact, represents a full binary (not necessarily balanced) tree. Note that replacing these high in-degree vertices with trees changes the edge expansion of the graph by a constant factor at most (as this graph is of constant size, and connected). Moreover, there is no change in the number of input and output vertices.

5.2.1 Computation Graph for n-by-n Matrices

Assume without loss of generality that n is an integer power of 2. Denote by $Enc_{\lg n}A$ the part of $H_{\lg n}$ that corresponds to the encoding of matrix A. Similarly, let $Enc_{\lg n}B$ and

$Dec_{\lg n}C$ correspond to the parts of $H_{\lg n}$ that compute the encoding of B and the decoding of C, respectively.

5.2.1.1 Top-Down Construction

We next construct the computation graph H_{i+1} by constructing $Dec_{i+1}C$ (from Dec_iC and Dec_1C) and similarly constructing $Enc_{i+1}A$ and $Enc_{i+1}B$, then composing the three parts together.

- Replicate Dec_1C 7^i times.
- Replicate Dec_iC four times.
- Identify the $4 \cdot 7^i$ output vertices of the copies of Dec_1C with the $4 \cdot 7^i$ input vertices of the copies of Dec_iC:
  - Recall that each Dec_1C has four output vertices.
  - The set of each first output vertex of the 7^i Dec_1C graphs is identified with the set of 7^i input vertices of the first copy of Dec_iC.
  - The set of each second output vertex of the 7^i Dec_1C graphs is identified with the set of 7^i input vertices of the second copy of Dec_iC, and so on.
  - We make sure that the jth input vertex of a copy of Dec_iC is identified with an output vertex of the jth copy of Dec_1C.
- We similarly obtain $Enc_{i+1}A$ from Enc_iA and Enc_1A and also $Enc_{i+1}B$ from Enc_iB and Enc_1B.
- For every i, H_i is obtained by connecting edges from the jth output vertices of Enc_iA and Enc_iB to the jth input vertex of Dec_iC.

This completes the construction. Let us note an important property of these graphs.

Claim 5.3. *All vertices of $Dec_{\lg n}C$ are of degree at most 6.*

Proof. The graph Dec_1C has no vertices which are both input and output. As all out-degrees are at most 4 and all in degree are at most 2 the claim follows. □

However, note that Enc_1A and Enc_1B do have vertices which are both input and output (*e.g.*, A_{11}), therefore $Enc_{\lg n}A$ and $Enc_{\lg n}B$ have vertices of out-degree $\Theta(\lg n)$. All in-degrees are at most 2, as an arithmetic operation has at most two inputs. As $H_{\lg n}$ contains vertices of large degrees, it is easier to consider $Dec_{\lg n}C$: it contains only vertices of constant bounded degree, yet at least one third of the vertices of $H_{\lg n}$ are in it.

5.2.1.2 Combinatorial Estimation of the Expansion

Let $G_k = (V, E)$ be $Dec_k C$, and let $S \subseteq V, |S| \le |V|/2$. Let l_i be the ith level of vertices of G_k, so

$$4^k = |l_1| < |l_2| < \cdots < |l_i| = 4^{k-i+1}7^{i-1} < \cdots < |l_{k+1}| = 7^k.$$

The following bounds on the fraction of vertices in the first level will be useful:

Claim 5.4. $\frac{3}{7} \cdot \left(\frac{4}{7}\right)^k \le \frac{|l_1|}{|V|} \le \frac{3}{7} \cdot \left(\frac{4}{7}\right)^k \cdot \frac{1}{1-\left(\frac{4}{7}\right)^{k+1}}$.

Proof. The claim follows from the following identities:

$$|V| = \sum_{i=1}^{k+1} |l_i| = \sum_{i=0}^{k} |l_{k+1}| \cdot \left(\frac{4}{7}\right)^i = |l_{k+1}| \cdot \left(1 - \left(\frac{4}{7}\right)^{k+1}\right) \cdot \frac{7}{3} = \left(\frac{7}{4}\right)^k \cdot |l_1| \cdot \left(1 - \left(\frac{4}{7}\right)^{k+1}\right) \cdot \frac{7}{3}$$

□

Let $S_i \equiv S \cap l_i$. Let $\sigma = \frac{|S|}{|V|}$ be the fractional size of S and $\sigma_i = \frac{|S_i|}{|l_i|}$ be the fractional size of S at level i. Due to averaging, there exist i and i' such that $\sigma_i \le \sigma \le \sigma_{i'}$.

Claim 5.5. *Let $\delta_i \equiv \sigma_{i+1} - \sigma_i$. Then $|E(S, V \setminus S) \cap E(l_i, l_{i+1})| \ge c_1 \cdot d \cdot |\delta_i| \cdot |l_i|$, where c_1 is a constant which depends on G_1.*

Proof. Let G' be a G_1 component connecting l_i with l_{i+1} (so it has four vertices in l_i and seven in l_{i+1}). G' has no edges in $E(S, V \setminus S)$ if all or none of its vertices are in S. Otherwise, as G' is connected, it contributes at least one edge to $E(S, V \setminus S)$. The number of such G_1 components with all their vertices in S is at most $\min\left\{\frac{\sigma_i \cdot |l_i|}{4}, \frac{\sigma_{i+1} \cdot |l_{i+1}|}{7}\right\} = \min\{\sigma_i, \sigma_{i+1}\} \cdot \frac{|l_i|}{4}$. Similarly, the number of such G_1 components with none of their vertices in S is at most $\min\{1 - \sigma_i, 1 - \sigma_{i+1}\} \cdot \frac{|l_i|}{4}$. Therefore, there are at least $|\sigma_i - \sigma_{i+1}| \cdot \frac{|l_i|}{4}$ G_1 components with at least one vertex in S and one vertex that is not. The claim follows with $c_1 = \frac{1}{4d}$. □

Claim 5.6 (Homogeneity between levels)**.** *If there exists i so that $\frac{|\sigma - \sigma_i|}{\sigma} \ge \frac{1}{10}$, then*

$$|E(S, V \setminus S)| \ge c_2 \cdot d \cdot |S| \cdot \left(\frac{4}{7}\right)^k$$

where c_2 is a constant which depends on G_1.

Proof. Assume that there exists j so that $\frac{|\sigma-\sigma_j|}{\sigma} \geq \frac{1}{10}$. By Claim 5.5, we have

$$\begin{aligned}
|E(S, V \setminus S)| &\geq \sum_{i\in[k]} |E(S, V \setminus S) \cap E(l_i, l_{i+1})| \\
&\geq \sum_{i\in[k]} c_1 \cdot d \cdot |\delta_i| \cdot |l_i| \\
&\geq c_1 \cdot d \cdot |l_1| \sum_{i\in[k]} |\delta_i| \\
&\geq c_1 \cdot d \cdot |l_1| \cdot \left(\max_{i\in[k+1]} \sigma_i - \min_{i\in[k+1]} \sigma_i \right).
\end{aligned}$$

By the initial assumption, there exists j so that $\frac{|\sigma-\sigma_j|}{\sigma} \geq \frac{1}{10}$, therefore $\max_i \sigma_i - \min_i \sigma_i \geq \frac{\sigma}{10}$, then

$$|E(S, V \setminus S)| \geq c_1 \cdot d \cdot |l_1| \cdot \frac{\sigma}{10}.$$

By Claim 5.4, $|l_1| \geq \frac{3}{7} \cdot \left(\frac{4}{7}\right)^k \cdot |V|$,

$$|E(S, V \setminus S)| \geq c_1 \cdot d \cdot \frac{3}{7} \cdot \left(\frac{4}{7}\right)^k \cdot |V| \cdot \frac{\sigma}{10},$$

and since $|S| = \sigma \cdot |V|$,

$$|E(S, V \setminus S)| \geq c_2 \cdot d \cdot |S| \cdot \left(\frac{4}{7}\right)^k$$

for any $c_2 \leq \frac{1}{10} \cdot \frac{3}{7} \cdot c_1$. □

Let T_k correspond to the recursive construction of G_k in the following way (see Figure 5.3): T_k is a tree of height $k+1$, where each internal node has four children. The root r of T_k corresponds to l_{k+1} (the largest level of G_k). The four children of r correspond to the largest levels of the four graphs that one can obtain by removing the level of vertices l_{k+1} from G_k. For every node u of T_k, denote by V_u the set of vertices in G_k corresponding to u, so if u is at level i of T_k then $V_u \subseteq l_i$. One can think of T_k as a quadtree partitioning of matrix C into blocks, where V_u is the largest level of the decoding subgraph of the C sub-block corresponding to u. Therefore $|V_r| = 7^k$ where r is the root of T_k, $|V_u| = 7^{k-1}$ for each node u that is a child of r; in general we have 4^i tree nodes u corresponding to a set of size $|V_u| = 7^{k-i+1}$. Each leaf corresponds to a set of size 1.

For a tree node u, let us define $\rho_u = |S \cap V_u|/|V_u|$ to be the fraction of S nodes in V_u, and $\delta_u = |\rho_u - \rho_{p(u)}|$, where $p(u)$ is the parent of u (for the root r we let $p(r) = r$). We let t_i be the ith level of T_k, counting from the bottom, so t_{k+1} is the root and t_1 are the leaves.

Claim 5.7. *As $V_r = l_{k+1}$ we have $\rho_r = \sigma_{k+1}$. For a tree leaf $u \in t_1$, we have $|V_u| = 1$. Therefore $\rho_u \in \{0, 1\}$. The number of vertices u in t_1 with $\rho_u = 1$ is $\sigma_1 \cdot |l_1|$.*

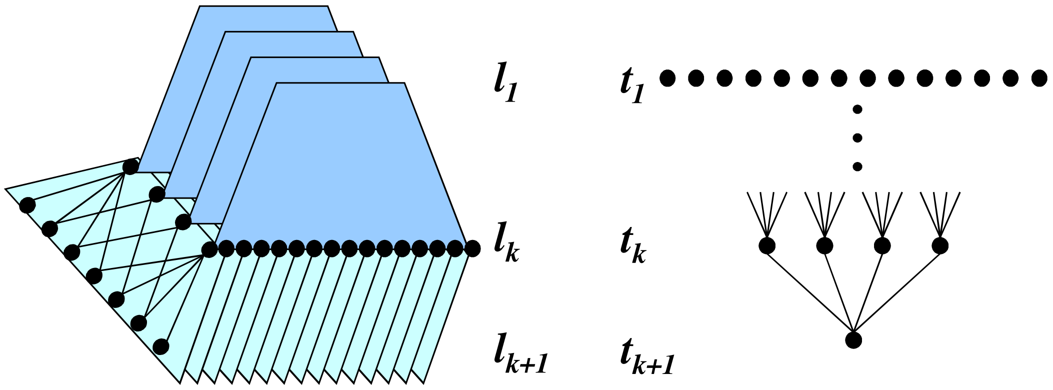


Figure 5.3: The graph G_k and its corresponding tree T_k.

Claim 5.8. *Let u_0 be an internal tree node, and let u_1, u_2, u_3, u_4 be its four children. Then*

$$\sum_i |E(S, V \setminus S) \cap E(V_{u_i}, V_{u_0})| \geq c_1 \cdot d \cdot \sum_i |\rho_{u_i} - \rho_{u_0}| \cdot |V_{u_i}|$$

where c_1 is a constant that depends on G_1.

Proof. The proof follows that of Claim 5.5. Let G' be a G_1 component connecting V_{u_0} with $\bigcup_{i\in[4]} V_{u_i}$ (so it has seven vertices in V_{u_0} and one in each of V_{u_1},V_{u_2},V_{u_3},V_{u_4}). G' has no edges in $E(S, V \setminus S)$ if all or none of its vertices are in S. Otherwise, as G' is connected, it contributes at least one edge to $E(S, V \setminus S)$. The number of G_1 components with all their vertices in S is at most $\min\{\rho_{u_0}, \rho_{u_1}, \rho_{u_2}, \rho_{u_3}, \rho_{u_4}\} \cdot |V_{u_1}|$. Therefore, there are at least $\max_{i\in[4]}\{|\rho_{u_0} - \rho_{u_i}|\} \cdot |V_{u_1}| \geq \frac{1}{4} \cdot \sum_{i\in[4]} |\rho_{u_i} - \rho_{u_0}| \cdot |V_{u_i}|$ G_1 components with at least one vertex in S and one vertex that is not. The claim follows with $c_1 = \frac{1}{4d}$. □

We now state and prove our main lemma on the edge expansion of the decoding graph of Strassen's CDAG:

Lemma 5.9. *The edge expansion of Dec_kC is*

$$h(Dec_kC) = \Omega\left(\left(\frac{4}{7}\right)^k\right).$$

Proof. Consider a subset S of the vertices of the decoding graph. Recall that $G_k = Dec_kC$ is a layered graph (with layers corresponding to recursion steps), so all edges (excluding loops) connect between consecutive levels of vertices. By Claim 5.6, each level of G_k contains about the same fraction of S vertices, or else we have many edges leaving S. By Claim 5.7, the lowest level is composed of distinct parts that cannot have homogeneity (of the fraction of their S vertices), otherwise many edges leave S.

Let T_k and V_u be defined as in the previous section. We will show that the homogeneity between levels, combined with the heterogeneity of the lowest level, guarantees that there are many edges leaving S.

We have

$$|E(S, V \setminus S)| = \sum_{u \in T_k} |E(S, V \setminus S) \cap E(V_u, V_{p(u)})|.$$

By Claim 5.8, this implies

$$\begin{aligned} |E(S, V \setminus S)| &\geq \sum_{u \in T_k} c_1 \cdot d \cdot |\rho_u - \rho_{p(u)}| \cdot |V_u| \\ &= c_1 \cdot d \cdot \sum_{i \in [k]} \sum_{u \in t_i} |\rho_u - \rho_{p(u)}| \cdot 7^{i-1} \\ &\geq c_1 \cdot d \cdot \sum_{i \in [k]} \sum_{u \in t_i} |\rho_u - \rho_{p(u)}| \cdot 4^{i-1}. \end{aligned}$$

As each internal node has four children, this means

$$|E(S, V \setminus S)| = c_1 \cdot d \cdot \sum_{v \in t_1} \sum_{u \in v \sim r} |\rho_u - \rho_{p(u)}|,$$

where $v \sim r$ is the path from v to the root r. By the triangle inequality for the function $|\cdot|$

$$|E(S, V \setminus S)| \geq c_1 \cdot d \cdot \sum_{v \in t_1} |\rho_u - \rho_r|.$$

By Claim 5.7,

$$|E(S, V \setminus S)| \geq c_1 \cdot d \cdot |l_1| \cdot ((1 - \sigma_1) \cdot \rho_r + \sigma_1 \cdot (1 - \rho_r)).$$

By Claim 5.6, w.l.o.g., $|\sigma_i - \sigma|/\sigma \leq \frac{1}{10}$ (otherwise $|E(S, V \setminus S)| \geq c_2 \cdot d \cdot |S| \cdot \left(\frac{4}{7}\right)^k$), so $\frac{9}{10}\sigma \leq \sigma_i \leq \frac{11}{20}$. As $\sigma \leq \frac{1}{2}$ and $\rho_r = \sigma_{k+1}$,

$$|E(S, V \setminus S)| \geq \frac{81}{100} \cdot c_1 \cdot d \cdot |l_1| \cdot \sigma,$$

and by Claim 5.4,

$$|E(S, V \setminus S)| \geq c_3 \cdot d \cdot |S| \cdot \left(\frac{4}{7}\right)^k,$$

for any $c_3 \leq \frac{3}{7} \cdot \frac{81}{100} \cdot c_1$.

Thus, since d is constant (Claim 5.3), we have $\frac{|E(S,V \setminus S)|}{|S|} = \Omega\left(\left(\frac{4}{7}\right)^k\right)$, where the hidden constant is $c_4 = d \cdot \min\{c_2, c_3\}$. □

5.3 Communication Lower Bounds

Theorem 5.10. *Consider Strassen's algorithm implemented on a sequential machine with fast memory of size M. Then for $M \leq n^2$, and assuming no recomputation, the bandwidth cost of Strassen's algorithm is*

$$W = \Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{\lg 7} \cdot M\right).$$

Proof. Let $k = \lg \sqrt{M} + c_5$ where c_5 is a constant to be determined, and assume k divides $\lg n$ evenly. Note that it is sufficient to prove the result for an infinite number of n's, but the smallest n for which the proof holds is $n = 2^{c_5}\sqrt{M}$ (so that $k = \lg n$). This assumption implies that $Dec_{\lg n}C$ is composed of edge-disjoint copies of $Dec_k C$, and we can apply Lemma 2.10 with $G = Dec_{\lg n}C$, $G' = Dec_k C$, and $s = |V(G')|/2$. Since d and d' are the same, we have

$$h_s(Dec_{\lg n}C) \geq h(Dec_k C)$$

and by Lemma 5.9 this implies

$$h_s(Dec_{\lg n}C) \geq c_4 \cdot \left(\frac{4}{7}\right)^k .$$

Note that $7^k/2 \leq s \leq 2 \cdot 7^k$.

We now apply Claim 5.2 with G as the entire CDAG of Strassen's algorithm of matrix dimension n and $G' = Dec_{\lg n}C$. Here $\alpha = 1/3$ and

$$h_s(Dec_{\lg n}C) \cdot \alpha s \geq \frac{c_2}{6} \cdot 4^{c_5} \cdot M \geq 3M$$

for $c_5 \geq \lg \sqrt{18/c_4}$, so

$$W \geq \alpha \cdot \frac{|V|}{s} \cdot M = \Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{\lg 7} \cdot M\right).$$

The above inequality holds for $M \leq c_6 \cdot n^2$, where $c_6 = 18/c_4 < 1$. For $c_6 \cdot n^2 \leq M \leq n^2$, note that

$$W \geq n^2 = \Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{\lg 7} \cdot M\right)$$

as one has to read $2n^2$ words of input data and at most n^2 of them can be in the fast memory at the start of the computation. $\square$

Theorem 5.10 holds for any implementation and any known variant of Strassen's algorithm[1] that is based on performing 2×2 matrix multiplication with 7 scalar multiplications.

[1]This lower bound for the sequential case seems to contradict the upper bound from [70] and later [46], due to a miscalculation in the former which is propagated in the latter ([104]).

This includes Winograd's $O(n^{\lg 7})$ variant (see Section 2.4.2) that uses 15 additions instead of 18, which is the most used fast matrix multiplication algorithm in practice [64, 65, 105]. See Section 7.2 for a scheduling of the computation that attains this lower bound. Note that Theorem 5.10 does not hold for values of M which are so large that the entire problem can fit into fast memory simultaneously. In the case that the input matrices start in fast memory and the output matrix finishes in fast memory, no communication is necessary.

For parallel algorithms, we have:

Corollary 5.11. *Consider Strassen's algorithm implemented on a parallel machine with P processors, each with a local memory of size M. There exists a constant c such that for $M \leq c \cdot \frac{n^2}{P^{2/\lg 7}}$, and assuming no recomputation, the bandwidth cost of Strassen's algorithm is*

$$W = \Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{\lg 7} \cdot \frac{M}{P}\right).$$

Proof. In the parallel case, we consider the busiest processor. Due to averaging, it must do at least $(1/p)$th of the work. We apply the same partitioning argument as in the proof of Theorem 5.10 to that processor's subset of computation. However, in order for the proof to work we must require $M \leq c\frac{n^2}{p^{2/\lg 7}}$ for some constant c (rather than $M \leq n^2$ in the sequential case). □

While Corollary 5.11 does not hold for all sizes of local memory (relative to the problem size and number of processors), there exists another lower bound that holds for all local memory sizes, though it requires separate assumptions (see Section 6.2). See Chapter 11 for a parallel algorithm that attains this bound where possible.

5.4 Conclusions

As we explain in Chapter 7, the lower bound of Theorem 5.10 (for sequential machines) is attainable with the natural recursive algorithm. This has an important (and somewhat surprising) implication. Re-writing the lower bound for classical square matrix multiplication given by Theorem 3.1 as $W_{\text{classical}} = \Omega((n/\sqrt{M})^3 \cdot M)$, we can more easily compare it to Theorem 5.10. Since $\lg 7 < 3$, we see that Strassen's algorithm requires not only less computation than the classical algorithm, it also requires less communication! While we often reduce communication at the expense of extra computation, in the case of Strassen's algorithm, we can reduce both simultaneously. In the parallel case, however, exploiting this opportunity is less straightforward. The absence of a parallel algorithm that attained the lower bound of Corollary 5.11 sparked our interest in developing a new and more communication-efficient algorithm. We present this new algorithm in Chapter 11 and show that it attains the lower bound (where possible), and we discuss other algorithms that use Strassen's or other fast matrix multiplication algorithms in Chapters 7 and 8.

Chapter 6

Extensions of the Lower Bounds

In this chapter we describe several extensions of the analysis and results presented in the previous chapters. The main contributions here are

- extending the lower bound for Strassen's algorithm to other fast matrix multiplication algorithms,
- proving a separate set of memory-independent lower bounds for both classical and fast parallel algorithms that (1) are tighter than the bounds proved in Chapters 4 and 5 in some cases and (2) provide limits on the possibility of perfect strong scaling (see Table 6.1 for a summary) , and
- summarizing further extensions of the lower bound results and proof techniques.

In Section 6.1 we define the "Strassen-like" class of algorithms and show how the analysis of Chapter 5 can be used to obtain similar results for this class of algorithms. These results extend beyond square matrix multiplication algorithms to other linear algebra computations as well as rectangular matrix multiplication. In the case of parallel algorithms, the lower bounds proved in the previous chapters are not always tight. We show in Section 6.2 that there are "memory-independent" lower bounds, proved using similar techniques as the bounds that do depend on the local memory size, which hold independently for most linear algebra computations. These new bounds, when applied in combination with the memory-dependent ones, tighten the parallel lower bounds for a wider range of problem sizes (relative to the number of processors and local memory sizes). In Section 6.3 we briefly discuss other extensions to the lower bound analysis, but we leave the details to the references given.

Most of the content of Section 6.1 appears in [26], written with coauthors James Demmel, Olga Holtz, and Oded Schwartz. The exception is Section 6.1.3 which appears in [29], and Section 6.1.4 is a brief summary of [21], written with the additional coauthor Benjamin Lipshitz. The results of Section 6.2 appear in [19] (without proof); the proofs are included in the technical report [22], both written with all four coauthors.

6.1 Strassen-like Algorithms

We can extend the bounds for Strassen's matrix multiplication to a wider class of algorithms, namely Strassen-like algorithms.

Definition 6.1. *A* Strassen-like algorithm *is a recursive algorithm for multiplying square*[1] *matrices which is constructed from a base case of multiplying $n_0 \times n_0$ matrices using $m_0 < n_0^3$ scalar multiplications, resulting in an algorithm for multiplying $n \times n$ matrices requiring $O(n^{\omega_0})$ flops where $\omega_0 = \log_{n_0} m_0$. In order to be* Strassen-like*, the base case decoding graph (referred to as Dec_1C in Section 5.2), which gives the dependencies between the m_0 scalar multiplication results and the n_0^2 entries of the output matrix, must be connected.*

Given two matrices of size $n \times n$, a Strassen-like algorithm splits them into n_0^2 blocks (each of size (n/n_0)-by-(n/n_0)), and works block-wise, according to the base algorithm. Additions (and subtractions) in the base algorithm are interpreted as additions (and subtractions) of blocks; multiplications in the base algorithm are interpreted as multiplications of blocks, which are performed by recursively calling the algorithm. The arithmetic count of the algorithm is then $T(n) = m_0 \cdot T(n/n_0) + O(n^2)$, so $T(n) = \Theta(n^{\omega_0})$ where $\omega_0 = \log_{n_0} m_0$.

This is the structure of nearly all the fast matrix multiplication algorithms that were obtained since Strassen's (see [151] for a summary and the most recent results). In fact, any $O(n^{\omega_0})$ matrix multiplication algorithm can be converted into a recursive matrix multiplication algorithm of running time $O(n^{\omega_0+\varepsilon})$ for any $\varepsilon > 0$ [124]. Furthermore, the algorithm can be made numerically stable while preserving this form [59].

However, to be considered Strassen-like, an algorithm's computation graph must also satisfy a technical assumption described in Section 6.1.1. This precludes some of the fast algorithms (and perhaps others whose computation graphs have not been explicitly specified) as well as the classical $O(n^3)$ algorithm.

6.1.1 Connected Decoding Graph Assumption

For our technique to work, and in order to be considered Strassen-like, we demand that the Dec_1C part of the computation graph is a connected graph (this is assumed in the proof of Claim 5.5 in Section 5.2.1.2). Thus the Strassen-like class includes Winograd's variant of Strassen's algorithm [152], which uses 15 additions rather than 18. The Strassen-like class does not contain the classical algorithm, where Dec_1C is composed of four disconnected graphs (corresponding to the four outputs). We believe this assumption is an artifact of our proof technique and unnecessary for the same lower bounds to apply.

6.1.2 Communication Costs of Strassen-like Algorithms

To prove Theorem 6.3, which generalizes the bandwidth cost lower bound of Strassen's algorithm (Theorem 5.10) to all Strassen-like algorithms, we note the following: the entire

[1]See Section 6.1.4 for extensions to fast rectangular matrix multiplication algorithms.

proof of Theorem 5.10, and in particular, the computations in the proof of Lemma 5.9, hold for any Strassen-like algorithm, where we plug in n_0^2 and m_0, instead of 4 and 7. For bounding the asymptotic bandwidth cost, we do not care about the number of internal vertices of Dec_1C; we need only to know that Dec_1C is connected (this critical technical assumption is used in the proof of Claim 5.5), and to know the sizes n_0 and m_0. The only nontrivial adjustment is to show the equivalent of Claim 5.3, that the decoding graph is of bounded degree. This is given in the following claim:

Claim 6.2. *The decoding graph of any Strassen-like algorithm is of degree bounded by a constant which is independent of the input size.*

Proof. If the set of input vertices of Dec_1C, and the set of its output vertices are disjoint, then the entire decoding graph is of constant bounded degree (its maximal degree is at most twice the largest degree of Dec_1C).

Assume (towards contradiction) that the base graph Dec_1C has an input vertex which is also an output vertex. An output vertex represents the inner product of two n_0-long vectors (*i.e.*, the corresponding row-vector of A and column vector of B). The corresponding bilinear polynomial is irreducible. This is a contradiction, since an input vertex represents the multiplication of a (weighted) sum of elements of A with a (weighted) sum of elements of B. □

Thus, we state (without formal proof) the extensions of Theorem 5.10 and Corollary 5.11 to Strassen-like algorithms:

Theorem 6.3. *Consider a recursive Strassen-like fast matrix multiplication algorithm with $O(n^{\omega_0})$ arithmetic operations implemented on a sequential machine with fast memory of size M. Then for $M \le n^2$, and assuming no recomputation, the bandwidth cost of the Strassen-like algorithm is*

$$W = \Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{\omega_0} \cdot M\right).$$

Corollary 6.4. *Consider a Strassen-like algorithm implemented on a parallel machine with P processors, each with a local memory of size M. There exists a constant c such that for $M \le c \cdot n^2/P^{2/\omega_0}$, and assuming no recomputation, the bandwidth cost of the Strassen-like algorithm is*

$$W = \Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{\omega_0} \cdot \frac{M}{P}\right).$$

6.1.3 Fast Linear Algebra

A straightforward extension of lower bounds for matrix multiplication extend to computations that involve a matrix multiplication subcomputation. As shown in [58], many linear algebra computations can be performed recursively, with the same computational complexity as the matrix multiplication subroutine they call. As we will see in Section 7.2, these algorithms also attain the same asymptotic communication costs as the matrix multiplication algorithm they employ. The following lower bound applies to all the algorithms in [58] assuming the fast matrix multiplication subroutine is Strassen-like.

Corollary 6.5. *Suppose an algorithm has a CDAG containing as a subgraph the CDAG of a Strassen-like matrix multiplication algorithm with input size $\Theta(n)$ which performs $\Theta(n^{\omega_0})$ flops. Then, assuming that no intermediate value is computed twice, the number of words moved during the computation on a machine with fast memory of size M is*

$$W = \Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{\omega_0} \cdot M\right).$$

Proof. The proof of Theorem 6.3 is based on an analysis of the CDAG of a Strassen-like matrix multiplication algorithm. If the CDAG of a computation includes as a subgraph a CDAG which corresponds to $\Theta(n) \times \Theta(n)$ Strassen-like matrix multiplication, then the analysis yields the same communication lower bound for that subset of the computation and therefore the entire computation. □

Note that there may be many different CDAGs which correspond to computing, for instance, an LU decomposition using a Strassen-like matrix multiplication as a subroutine. For example, the algorithms of [58] split the matrix into equal-sized left and right halves, but another algorithm may split the matrix into a tall-skinny panel and a larger trailing matrix. Corollary 6.5 applies to all such algorithms that contain a sufficiently large subgraph corresponding to a Strassen-like matrix multiplication.

This result implies that given the CDAG that a recursive algorithm of [58] produces, no re-ordering of the computation can improve the communication costs by more than a constant factor compared to the depth-first ordering given by the recursive algorithm. The result does not apply to algorithms which restructure the CDAG beyond the freedom allowed by commutativity and associativity of addition. See Section 7.2 for more details.

6.1.4 Fast Rectangular Matrix Multiplication Algorithms

Many fast algorithms have been devised for multiplication of rectangular matrices (see [21] for a detailed list). A fast algorithm for multiplying $m_0 \times n_0$ and $n_0 \times p_0$ matrices in $q < m_0n_0p_0$ scalar multiplications can be applied recursively to multiply $m_0^t \times n_0^t$ and $n_0^t \times p_0^t$ matrices in $O(q^t)$ flops. For such algorithms, the CDAG has very similar structure to Strassen and Strassen-like algorithms for square multiplication in that it is composed of two encoding graphs and one decoding graph. Assuming that the decoding graph is connected, the proofs of Theorem 5.10 and Lemma 5.9 apply where we plug in m_0p_0 and q for 4 and 7. In this case, we obtain a result analogous to Theorem 5.10 which states that the bandwidth cost of such an algorithm is given by

$$W = \Omega\left(\frac{q^t}{M^{\log_{m_0p_0} q-1}}\right).$$

If the output matrix is the largest of the three matrices (*i.e.*, $n_0 < m_0$ and $n_0 < p_0$), then this lower bound is tight (*e.g.*, it is attained by the natural recursive algorithm). The lower bound extends to the parallel case as well, analogous to Corollary 5.11.

However, in the case that the decoding graph is not connected, the proof does not apply, and in the case the output matrix is not the largest, the lower bound is not necessarily tight. In order to handle these technical challenges, we can employ modifications of the proof technique: we can deal with the decoding graph being disconnected by considering individual connected components, or we can consider one of the two encoding graphs, which may contain vertices of high degree. In either case, the proofs must be adapted, and we obtain slightly weaker results. We detail these approaches in [21] and discuss the application to rectangular matrix multiplication algorithms of [37] and [89].

6.2 Memory-Independent Lower Bounds

The lower bounds for classical linear algebra in Chapter 4 and Strassen's matrix multiplication in Chapter 5 each depend on the size of the fast or local memory, M. Since M appears in the denominator in both cases, these bounds suggest that as more local memory is used, less communication is necessary. In the parallel case, especially for large P, there may be much more local memory available than what is required to store the inputs and outputs of the computation. In fact, algorithms discussed in Section 8.2 show that extra local memory (*i.e.*, $M \gg n^2/P$) can be used effectively to reduce communication.

However, there is a limit to the tradeoff between extra memory and reduced communication. Using similar proof techniques to those used in Chapters 4 and 5, we can prove memory-independent lower bounds (with more restrictive assumptions on the initial data layout and computational load-balance) that begin to dominate the memory-dependent lower bounds for certain problem sizes.

In addition to tightening existing bounds, the lower bounds in this section yield another interesting conclusion regarding strong scaling. We say that an algorithm exhibits *perfect strong scaling* if it attains running time on P processors which is linear in $1/P$, including all communication costs, for some range of P. For example, Cannon's parallel matrix multiplication algorithm (see Section 8.1.1) has a parallel computational cost of $O(n^3/P)$ flops but a bandwidth cost of $O(n^2/\sqrt{P})$ words. Thus, Cannon's algorithm scales perfectly with respect to the computational cost but not with respect to the communication cost. While it is possible for classical and Strassen-based matrix multiplication algorithms to strongly scale perfectly, the communication costs restrict the strong scaling ranges much more than do the computation costs. These ranges depend on the problem size relative to the local memory size, and on the computational complexity of the algorithm.

Interestingly, in both cases the dominance of a memory-independent bound arises, and the strong scaling range ends, exactly when the memory-dependent latency lower bound becomes $\Omega(1)$. Of course, since the latency cost cannot possibly drop below a constant, it is an immediate result of the memory-dependent bounds that the latency cost cannot continue to strongly scale perfectly. However, for sufficiently large problems, the bandwidth cost typically dominates the cost, and the memory-independent bandwidth scaling bounds limit the strong scaling of matrix multiplication in practice. For simplicity we omit discussions of latency cost in this section, since the lower bound on the number of messages is always a factor of M below the bandwidth cost in the strong scaling range and is always constant outside the strong scaling range.

While the main arguments in this section focus on matrix multiplication, they can be generalized to other algorithms, including other three-nested-loops computations (see Definition 4.1) and other Strassen-like algorithms (see Definition 6.1). See Section 6.2.3 for more details.

6.2.1 Communication Lower Bounds

6.2.1.1 Classical Matrix Multiplication

In this section, we prove a memory-independent lower bound for classical matrix multiplication of $\Omega(n^2/P^{2/3})$ words. The same result appears elsewhere in the literature, under slightly different assumptions: in the LPRAM model [4], where no data exists in the (unbounded) local memories at the start of the algorithm; in the distributed-memory model [95], where the local memory size is assumed to be $M = \Theta(n^2/P^{2/3})$; and in the distributed-memory model [137], where the algorithm is assumed to perform a certain amount of input replication. Our bound is for the distributed memory model, holds for any M, and assumes no specific communication pattern.

Using Lemma 2.7 (in a similar way to Chapter 4 and [95]), we can describe the ratio between the number of scalar multiplications a processor performs and the amount of data it must access.

Lemma 6.6. *Suppose a processor has $\mathcal{I}$ words of initial data at the start of an algorithm, performs $\Theta(n^3/P)$ scalar multiplications within classical matrix multiplication, and then stores $\mathcal{O}$ words of output data at the end of the algorithm. Then the processor must send or receive at least $\Omega(n^2/P^{2/3}) - \mathcal{I} - \mathcal{O}$ words during the execution of the algorithm.*

Proof. We follow the proofs of Chapter 4 and [95]. Consider a discrete $n \times n \times n$ cube where the lattice points correspond to the scalar multiplications within the matrix multiplication $A \cdot B$ (*i.e.*, lattice point (i, j, k) corresponds to the scalar multiplication $a_{ik} \cdot b_{kj}$). Then the three pairs of faces of the cube correspond to the two input and one output matrices.

The projections on the three faces correspond to the input/output elements the processor has to access (and must communicate if they are not in its local memory). By Lemma 2.7, and the fact that $\sqrt{|V_x| \cdot |V_y| \cdot |V_z|} \leq \sqrt{\frac{1}{6}(|V_x| + |V_y| + |V_z|)^3}$, the number of words the processor must access is at least $\sqrt[3]{6}\ |V|^{2/3} = \Omega(n^2/P^{2/3})$. Since the processor starts with $\mathcal{I}$ words and ends with $\mathcal{O}$ words, the result follows. □

Theorem 6.7. *Suppose a parallel algorithm performing classical dense matrix multiplication begins with one copy of the input matrices and minimizes computational costs in an asymptotic sense. Then, for sufficiently large P, some processor has a bandwidth cost of*

$$W = \Omega\left(\frac{n^2}{P^{2/3}}\right).$$

Proof. At the end of the algorithm, every element of the output matrix must be fully computed and exist in some processor's local memory (though multiple copies of the element may exist in multiple memories). For each output element, we designate one memory location as the output and disregard all other copies. For each of the n^2 designated memory locations, we consider the n scalar multiplications whose results were used to compute its value and disregard all other redundantly computed scalar multiplications.

In order to minimize computational costs asymptotically, the running time for classical dense matrix multiplication must be $O(n^3/P)$. This is possible only if at least a constant fraction of the processors perform $\Theta(n^3/P)$ of the scalar multiplications corresponding to designated outputs.

Since there exists only one copy of the input matrices and designated output–$O(n^2)$ words of data–some processor which performs $\Theta(n^3/P)$ multiplications must start and end with no more than $I + O = O(n^2/P)$ words of data. Thus, by Lemma 6.6, some processor must read or write $\Omega(n^2/P^{2/3}) - O(n^2/P) = \Omega(n^2/P^{2/3})$ words of data. □

Note that the theorem applies to any $P \geq 2$ with a strict enough assumption on the load balance. For discussion of algorithms attaining this bound, see Section 8.2.1.

6.2.1.2 Strassen's Matrix Multiplication

In this section, we prove a memory-independent lower bound for Strassen's matrix multiplication of $\Omega(n^2/P^{2/\lg 7})$ words. We reuse notation and proof techniques from Chapter 5. By prohibiting redundant computations we mean that each arithmetic operation is computed by exactly one processor. This is necessary for interpreting edge expansion as communication cost.

Theorem 6.8. *Suppose a parallel algorithm performing Strassen's matrix multiplication minimizes computational costs in an asymptotic sense and performs no redundant computation. Then, for sufficiently large P, some processor must have a bandwidth cost of*

$$W = \Omega\left(\frac{n^2}{P^{2/\lg 7}}\right).$$

Proof. Recall that the computation DAG of Strassen's algorithm multiplying square matrices $A \cdot B = C$ can be partitioned into three subgraphs: an encoding of the elements of A, an encoding of the elements of B, and a decoding of the scalar multiplication results to compute the elements of C. These three subgraphs are connected by edges that correspond to scalar multiplications. Call the third subgraph $Dec_{\lg n}C$, where $\lg n$ is the number of levels of recursion for matrices of dimension n.

In order to minimize computational costs asymptotically, the running time for Strassen's matrix multiplication must be $O(n^{\lg 7}/P)$. Since a constant fraction of the flops correspond to vertices in $Dec_{\lg n}C$, this is possible only if some processor performs $\Theta(n^{\lg 7}/P)$ flops corresponding to vertices in $Dec_{\lg n}C$.

By Lemma 5.9, the edge expansion of Dec_kC is given by $h(Dec_kC) = \Omega((4/7)^k)$. Using Claim 5.2, we deduce that

$$h_s(Dec_{\lg n}C) = \Omega\left(\left(\frac{4}{7}\right)^{\log_7 s}\right), \tag{6.1}$$

where h_s is the edge expansion for sets of size at most s.

Let S be the set of vertices of $Dec_{\lg n}C$ that correspond to computations performed by the given processor. Set $s = |S| = \Theta(n^{\lg 7}/P)$. By Equation (6.1), the number of edges between S and $\overline{S}$ is

$$|E(S, \overline{S})| = \Omega\left(s \cdot h_s(Dec_{\lg n}C)\right) = \Omega\left(\frac{n^2}{P^{2/\lg 7}}\right),$$

and because $Dec_{\lg n}C$ is of bounded degree (Claim 5.3) and each vertex is computed by only one processor, the number of words moved is $\Theta(|E(S, \overline{S})|)$ and the result follows. □

Note that the theorem applies to any $P \geq 2$ with a strict enough assumption on the load balance among vertices in $Dec_{\lg n}C$ as defined in the proof. For details of the algorithm that attains this bound, see Chapter 11.

6.2.2 Limits of Strong Scaling

In this section we present limits of strong scaling of matrix multiplication algorithms. These are immediate implications of the memory independent communication lower bounds proved in Section 6.2.1. Roughly speaking, the memory-dependent communication cost lower bound is of the form $\Omega\left(f(n, M)/P\right)$ for both classical and Strassen matrix multiplication algorithms. However, the memory independent lower bounds are of the form $\Omega\left(f(n)/P^c\right)$ where $c < 1$ (see Table 6.1). This implies that strong scaling is not possible when the memory-independent bound dominates. We make this formal below.

Corollary 6.9. *Suppose a parallel algorithm performing Strassen's matrix multiplication minimizes bandwidth and computational costs in an asymptotic sense and performs no redundant computation. Then the algorithm can achieve perfect strong scaling only for*

$$P = O\left(\frac{n^{\lg 7}}{M^{(\lg 7)/2}}\right).$$

	Classical	Strassen
Memory-dependent lower bound	$\Omega\left(\frac{n^3}{P\sqrt{M}}\right)$	$\Omega\left(\frac{n^{\lg 7}}{PM^{(\lg 7)/2-1}}\right)$
Memory-independent lower bound	$\Omega\left(\frac{n^2}{P^{2/3}}\right)$	$\Omega\left(\frac{n^2}{P^{2/\lg 7}}\right)$
Perfect strong scaling range	$P = O\left(\frac{n^3}{M^{3/2}}\right)$	$P = O\left(\frac{n^{\lg 7}}{M^{(\lg 7)/2}}\right)$

Table 6.1: Bandwidth cost lower bounds for matrix multiplication and perfect strong scaling ranges. The classical memory dependent bound is due to [95], and the Strassen memory dependent bound is proved in Chapter 5. The memory-independent bounds are proved here, though variants of the classical bound appear in [4, 95, 137].

Proof. By Corollary 5.11, any parallel algorithm performing matrix multiplication based on Strassen moves at least $\Omega(n^{\lg 7}/PM^{(\lg 7)/2-1})$ words. By Theorem 6.8, a parallel algorithm that minimizes computational costs and performs no redundant computation moves at least $\Omega(n^2/P^{2/\lg 7})$ words. This latter bound dominates in the case $P = \Omega(n^{\lg 7}/M^{(\lg 7)/2})$. Thus, while a communication-optimal algorithm will strongly scale perfectly up to this threshold, after the threshold the communication cost will scale as $1/P^{2/\lg 7}$ rather than $1/P$. □

Corollary 6.10. *Suppose a parallel algorithm performing classical dense matrix multiplication starts and ends with one copy of the data and minimizes bandwidth and computational costs in an asymptotic sense. Then the algorithm can achieve perfect strong scaling only for*

$$P = O\left(\frac{n^3}{M^{3/2}}\right).$$

Proof. By [95], any parallel algorithm performing classical matrix multiplication moves at least $\Omega(n^3/(P\sqrt{M}))$ words. By Theorem 6.7, a parallel algorithm that starts and ends with one copy of the data and minimizes computational costs moves at least $\Omega(n^2/P^{2/3})$ words. This latter bound dominates in the case $P = \Omega(n^3/M^{3/2})$. Thus, while a communication-optimal algorithm will strongly scale perfectly up to this threshold, after the threshold the communication cost will scale as $1/P^{2/3}$ rather than $1/P$. □

In Figure 6.1 we present the asymptotic communication costs of classical and Strassen-based algorithms for a fixed problem size as the number of processors increases. Both types of perfectly strong scaling algorithms stop scaling perfectly above some number of processors, which depends on the matrix size and the available local memory size.

Let $P_{\min} = \Theta(n^2/M)$ be the minimum number of processors required to store the input and output matrices. By Corollaries 6.9 and 6.10 the perfect strong scaling range is $P_{\min} \leq P \leq P_{\max}$ where $P_{\max} = \Theta(P_{\min}^{3/2})$ in the classical case and $P_{\max} = \Theta(P_{\min}^{(\lg 7)/2})$ in the Strassen case. Note that the perfect strong scaling range is larger for the classical case, though the communication costs are higher. Also note that outside the perfect strong scaling range,

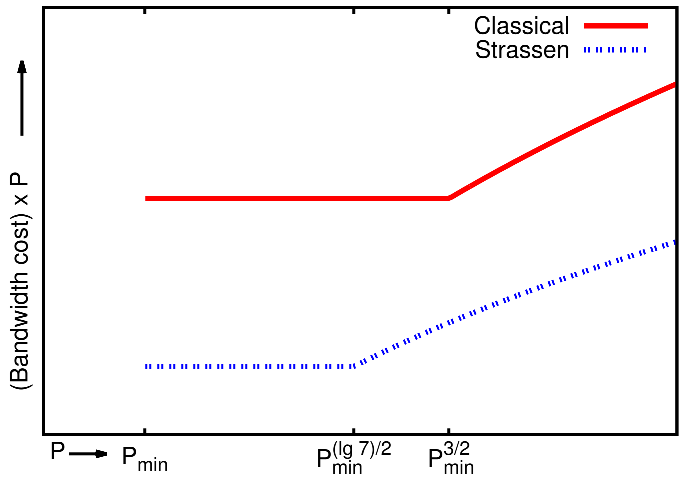


Figure 6.1: Bandwidth cost lower bounds and strong scaling of matrix multiplication: classical vs. Strassen. Horizontal lines correspond to perfect strong scaling. $\mathrm{P}_{\min}$ is the minimum number of processors required to store the input and output matrices.

the communication costs do not grow linearly as the scale of the figure seems to suggest. In the classical case, the (Bandwidth cost)$\times P$ is proportional to $P^{1/3}$; in the Strassen case, it is proportional to $P^{1-2/\lg 7} \approx P^{.29}$ outside the strong scaling range. Note that in both the classical and Strassen cases, there are algorithms attaining these lower bounds.

6.2.3 Extensions of Memory-Independent Bounds

The memory-dependent bound of classical matrix multiplication of [95] is generalized in Chapter 4 to algorithms that satisfy Definition 4.1. The memory-independent bound of classical matrix multiplication (Theorem 6.7) applies to these other algorithms as well. If the algorithm begins with one copy of the input data and minimizes computational costs in an asymptotic sense, then, for sufficiently large P, some processor must send or receive at least $\Omega\left(\left(\frac{G}{P}\right)^{2/3} - \frac{D}{P}\right)$ words, where G is the total number of g_{ijk} computations and D is the number of non-zeros in the input and output. The proof follows that of Lemma 6.6 and Theorem 6.7, setting $|V| = G$ (instead of n^3), replacing n^3/P with G/P , and setting $I + O = O(D/P)$ (instead of $O(n^2/P)$).

The memory-independent bound and perfect strong scaling bound of Strassen's matrix multiplication (Theorem 6.8 and Corollary 6.9) apply to other Strassen-like algorithms, as defined in Section 6.1, with ω_0 (the exponent of the total arithmetic count) replacing $\lg 7$, provided that the decoding graph is connected. The proof follows that of Theorem 6.8 and of Corollary 6.9, but uses Claim 6.2 instead of Claim 5.3 and replaces Lemma 5.9 with its extension.

6.3 Other Extensions

There are several more extensions of the lower bounds presented in this and the previous chapters and many open problems. In this section, we point out a few of these directions and the corresponding references.

6.3.1 k-Nested-Loops Computations

The heart of the argument made in Chapter 4 is based on relating a three-dimensional set of computation to two-dimensional data sets using the geometric inequality of Loomis and Whitney [107] (see Lemma 2.7). Since most linear algebra computations are three nested loops, this geometric relationship is sufficient for the algorithms considered here. Consider instead performing an "N-body" calculation, where we wish to compute all the pairwise interactions within a set of N particles. In this case, the data is one dimensional (a list of particles) and the computation is two dimensional (all N^2 pairwise interactions). Thus, Lemma 2.7, which is based on three-dimensional lattice points, no longer directly relates a subset of the computation to the data involved. However, the more general version of the result that appears in [107] can relate computation to communication for N-body calculation: it relates a d-dimensional volume to its projections onto $(d-1)$-dimensional subspaces.

If, on the other hand, a d-dimensional computation accesses some data of dimension $d-2$, some of $d-3$, and some of $d-1$, the Loomis-Whitney inequality is no longer helpful. A recent generalization of the Loomis-Whitney inequality [34] can be used to prove communication lower bounds for such computations (and many more). The statement of the more general inequality and its implications on communication costs for a wide class of algorithms are given in [51]. In the paper, the authors prove communication bounds for algorithms that have arbitrary numbers of loops and access arrays with arbitrary dimensions, as long as the index expressions are affine combinations of loop variables.

6.3.2 Sparse Matrix-Matrix Multiplication

The lower bounds given in Chapter 4 hold for both dense and sparse computations. However, particularly in the sparse case, the lower bounds may not be tight (*i.e.*, attainable). Although the focus of this dissertation is dense linear algebra, consider the multiplication of two sparse matrices. Corollary 4.3 effectively upper bounds the number of possible useful computations for every word of data read into fast/local memory at $O(\sqrt{M})$. However, for matrices which are very sparse, elements of an input matrix may be involved in far fewer than $O(\sqrt{M})$ computations, making that amount of data reuse unattainable. Even the memory-independent bounds described in Section 6.2.3 can be unattainable.

Using assumptions on the type of sparsity in the input matrices and properties particular to the computation, we can prove a tighter (and attainable) lower bound for sparse matrix-matrix multiplication [16]. The lower bound proof resembles the memory-independent bound proof of Section 6.2.1.1 but also depends on the randomness of the input matrices (and proves results that hold only in expectation).

Part II

Algorithms and Communication Cost Analysis

Chapter 7

Sequential Algorithms and their Communication Costs

The lower bound results of Chapters 3–6 provide targets for algorithmic development; in this chapter we focus on algorithms for sequential machines and discuss the current state-of-the-art in terms of communication costs. The main contribution of this chapter is to provide a comprehensive (though certainly not exhaustive) summary of communication-optimal sequential algorithms for dense linear algebra computations and provide references to papers that provide algorithmic details, communication cost analyses, and demonstrated performance improvements. We consider all of the fundamental computations—BLAS, Cholesky and symmetric-indefinite factorizations, LU and QR decompositions, eigenvalue and singular value decompositions—and compare the best sequential algorithms with the lower bounds established in Chapter 4. Chapter 8 summarizes parallel algorithms, and Chapters 9–11 focus on algorithms for particular computations.

Recall the sequential two-level memory model presented in Section 2.2.1. We consider communication between a fast memory of size M and a slow memory of unbounded size, and we track both the number of words and messages that an algorithm moves. Because a message requires its words to be stored contiguously in slow memory, we must specify the matrix data layout in determining latency costs (see Section 2.3.1 for details on data layouts). We also consider the multiple-level memory hierarchy model in this chapter, as it more accurately reflects today's machines.

One may imagine that sequential algorithms that minimize communication for any number of levels of a memory hierarchy (see Section 2.2.1.2) might be very complex, possibly depending not just on the number of levels, but also their sizes. In this context, it is worth distinguishing a class of algorithms, called *cache oblivious* [70], that can minimize communication between all levels (at least asymptotically) independent of the number of levels and their sizes. These algorithms are recursive, and provided a matching recursive layout is used, these algorithms may also minimize the number of messages independent of the number of levels of memory hierarchy. Not only do cache-oblivious algorithms perform well in theory, but they can also be adapted to perform well in practice (see [156], for example).

The rest of this chapter is divided into two sections, classical and fast linear algebra computations. In Section 7.1, we present Table 7.1 with references to the communication-optimal algorithms for the most fundamental dense linear algebra computations and then discuss each computation and the corresponding state-of-the-art algorithms in turn. In Section 7.2, we highlight extensions of fast matrix multiplication algorithms to other dense linear algebra computations.

The main contributions to the set of communication-optimal sequential algorithms, appearing either in this thesis or in manuscripts written with various sets of coauthors, are as follows:

- communication cost analysis for Cholesky decomposition algorithms [24],
- first sequential communication-optimal algorithm for symmetric-indefinite factorization [14] (see Chapter 9),
- first sequential cache-oblivious algorithm for LU decomposition to minimize latency costs (similar algorithm for QR decomposition is also communication optimal) [32],
- new sequential communication-avoiding algorithms for the symmetric eigendecomposition and SVD [30] (see Chapter 10),
- first sequential communication-optimal algorithm for the nonsymmetric eigendecomposition [17], and
- communication cost analysis for fast algorithms for linear algebra computations [29].

Table 7.1 is an updated version of Table 6.1 in [28], and some of the discussion here is based on that paper, written with coauthors James Demmel, Olga Holtz, and Oded Schwartz. The material on Cholesky decomposition in Section 7.1.2 is based on [24], written with the same set of coauthors, and the material on LU decomposition in Section 7.1.4 also appears in [32], written with coauthors James Demmel, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo.

7.1 Classical Linear Algebra

In this section, we discuss the current state-of-the-art for algorithms that perform the classical $O(n^3)$ computations on sequential machines. For each of the computations considered here, we can compare the communication costs of the algorithms to the lower bounds presented in Chapter 4. Table 7.1 summarizes the communication-optimal classical algorithms for the most fundamental dense linear algebra computations. We differentiate between algorithms that minimize communication only in the two-level model and those that are optimal also in a multiple-level memory hierarchy. We also differentiate between algorithms that minimize only bandwidth costs and those that minimize both bandwidth and latency costs.

Computation	Two Levels of Memory		Multiple Levels of Memory	
	Minimizes Words	Minimizes Messages	Minimizes Words	Minimizes Messages
BLAS-3	[32, 70]		[32, 70]	
Cholesky	[7, 8, 24, 83]	[7, 24, 83]	[7, 24, 83]	
Symmetric Indefinite	[14]	[14]	[14]	[14]
LU	[32, 80, 83, 142]	[32, 80]	[32, 83, 142]	[32]
QR	[32, 62, 66, 69]	[32, 62, 69]	[32, 66, 69]	[32, 69]
Sym Eig and SVD	[17, 30]		[17]	
Nonsym Eig	[17]		[17]	

Table 7.1: Sequential classical algorithms attaining communication lower bounds. We separately list algorithms that attain the lower bounds for two levels and multiple levels of memory hierarchy. In each of these cases, we separately list algorithms that minimize only the number of words moved and algorithms that also minimize the number of messages.

In order for an algorithm to be considered communication-optimal in the sequential model, we require that its communication costs be within a constant factor of the corresponding lower bound and that it performs no more than a constant factor more computation than alternative algorithms. Most of the algorithms have the same leading constant in computational cost as the standard algorithms, though we note where constant factor increases exist. In some cases, there exists a small range of matrix dimensions where the algorithm is suboptimal; the communication cost includes a term that exceeds the lower bound and is not always lower order in this range. For example, the rectangular recursive algorithms of [66, 83, 142] are suboptimal with respect to bandwidth cost when n satisfies $n/\log n \ll \sqrt{M} \ll n$ [32]. We omit these details here for sufficiently small ranges.

We emphasize that only a few of the communication-optimal algorithms referenced here are included in standard libraries like LAPACK. While this chapter focuses on asymptotic complexity rather than measured performance on current architectures, many of the papers referenced for algorithms here also include performance data and demonstrate significant speedups over asymptotically suboptimal alternatives. Our communal goal is to eventually make all of these algorithms available via widely used libraries. There is a very large body of work on many of these algorithms, and we do not pretend to have a complete list of citations. Instead we refer just to papers where these algorithms first appeared (to the best of our knowledge), with or without analysis of their communication costs, or to survey papers.

7.1.1 BLAS Computations

While the lower bounds given in Section 4.1.2.1 apply to all BLAS computations, only the BLAS-3 computations have algorithms that attain them. In the case of BLAS-2 and BLAS-1 computations, the arithmetic intensity (*i.e.*, ratio of computation to data) is $O(1)$, so it is impossible for the bandwidth cost to be a factor of $O(\sqrt{M})$ smaller than the arithmetic cost, assuming the data has to be read from slow memory. In other words, there exist tighter lower bounds for BLAS-2 and BLAS-1 computations based on the size of the inputs and outputs, so the lower bounds of Section 4.1.2.1 are valid but unattainable.

For BLAS-3 computations, blocked versions of the naive algorithms attain the lower bound in the two-level memory model when the block size size is chosen to be $\Theta(\sqrt{M})$ (see for example the Block-Mult algorithm in [70] for matrix multiplication). In order to attain the corresponding latency cost lower bound, a block-contiguous data structure is necessary so that every block computation involves contiguous chunks of memory. Furthermore, because of the self-similarity of matrices and these fundamental computations, the block computations can themselves be blocked. Using a nested level of blocking for each level of memory (and choosing the block sizes appropriately), these algorithms can minimize communication between every pair of successive levels in a memory hierarchy. Note that a matching hierarchical block-contiguous data structure is needed to minimize latency costs. We do not include a reference in Table 7.1, as these blocked algorithms are generally considered folklore.

In addition to the explicitly blocked algorithms, there are recursive algorithms for all of the BLAS-3 computations. As explained in [70] for rectangular matrix multiplication, these recursive algorithms also attain the lower bounds of Section 4.1.2.1. In order to minimize latency costs, we use a matching recursive data layout, like the rectangular recursive layout of [32] which matches the Rec-Mult algorithm of [70]. For computations involving square matrices, data layouts based on Morton orderings and its variants help minimize latency costs. For recursive algorithms for triangular solve, see for example [24, Algorithm 3], where the right hand sides form a square matrix, or [32, Algorithm 5] for the general rectangular case. Similar algorithms exist for symmetric and triangular matrix multiplications and symmetric rank-k updates. Because these algorithm are recursive and cache oblivious, they minimize communication costs between every pair of memory levels in a hierarchy.

7.1.2 Cholesky Decomposition

The Cholesky decomposition is used primarily for solving symmetric, positive-definite linear systems of equations. Because the computation inherits numerical stability properties from the matrices to which it is applied, it enjoys a freedom in algorithmic design (no pivoting is required), and communication-optimal algorithms are well known. For a more complete discussion of sequential algorithms for Cholesky decomposition and their communication properties, see [24].

The reference implementation in LAPACK [8] (`potrf`) is a blocked algorithm, and by choosing the block size to be $\Theta(\sqrt{M})$, the algorithm attains the lower bound of Corollary 4.6. As in the case of the BLAS computations, a block contiguous data structure is used to obtain the latency cost lower bound. An algorithm with nested levels of blocking and a matching data layout can minimize communication for multiple levels of memory.

A recursive algorithm for Cholesky decomposition was first proposed in [83] and later matched with a block-recursive data structure in [7]. We present the communication cost analysis in [24], where the algorithm is shown to be communication optimal and cache oblivious as long as cache-oblivious subroutines are used. Thus, the recursive algorithm is optimal for both two-level and multiple-level memory models.

Note also that for the Cholesky factorization of sparse matrices whose sparsity structure satisfy certain graph-theoretic conditions (having "good separators"), the lower bound of Corollary 4.6 can also be attained [79]. For general sparse matrices, the problem is open.

7.1.3 Symmetric-Indefinite Decompositions

If the linear system is symmetric but not positive definite, then pivoting is required to ensure numerical stability. The need for pivoting complicates the computation, and communication-efficient algorithms are not as straightforward. A brief history of symmetric-indefinite factorizations and their communication costs is given at the beginning of Chapter 9. The most commonly used factorization (and the one implemented in LAPACK) is LDL^T, where D is block diagonal with 2×2 and 1×1 blocks. An alternative factorization is due to Aasen [1] and computes a tridiagonal matrix T instead of a block diagonal matrix D. Both factorizations use symmetric pivoting. The same lower bound for dense matrices, given in Corollaries 4.13 and 4.14, applies to both computations.

However, no current communication-optimal algorithms exist that compute these factorizations directly. The implementations of LDL^T in LAPACK [8] (`sytrf`) and of LTL^T in [125] can attain the lower bound for large matrices (where $n \geq M$) but fail for reasonably sized matrices (where $\sqrt{M} \leq n \leq M$). They also never attain the latency cost lower bound, and work only for the two-level memory model. It is an open problem whether there exists a communication-optimal algorithm that computes the factorizations directly.

The communication-optimal algorithm presented in Chapter 9 and [14] first computes a factorization LTL^T where T is a symmetric band matrix (with bandwidth $\Theta(\sqrt{M})$) and then decomposes T in a second step. This algorithm is a block algorithm, and with a block contiguous data structure, it minimizes both words and messages in the two-level memory model.

Because the subroutines in the communication-avoiding symmetric-indefinite factorization can all be performed with blocked or recursive algorithms themselves, it is possible to extend the algorithm (with matching data structure) to minimize communication costs in the multiple-level memory model. Note that our Shape-Morphing LU algorithm [32] is necessary to perform the panel factorization subroutine with optimal latency cost for all subsequent levels of memory.

7.1.4 LU Decomposition

For general, nonsymmetric linear systems, an LU decomposition is the direct method of choice. As in the case of symmetric-indefinite systems, pivoting is required to maintain numerical stability. For performance reasons (and because it is generally sufficient in practice), we consider here performing only row interchanges. We leave consideration of communication-optimal complete-pivoting algorithms (those that perform both row and column interchanges) to future work (see [61, Section 5] for a possible approach). The content of this section also appears in [32, Section 6].

There is a long history of algorithmic innovation to reduce communication costs for LU factorizations. Table 1 in [32] highlights several of the innovations and compares the asymptotic communication costs of the algorithms discussed here.

The LU decomposition algorithm in LAPACK (`getrf`) is based on "blocking" in order to cast much of the work in terms of matrix-matrix multiplication rather than working column-by-column and performing most of the work as matrix-vector operations. The algorithm is a right-looking, blocked algorithm, and by choosing the right block size, the algorithm asymptotically reduces the communication costs compared to the column-by-column algorithm. In fact, for very large matrices ($m, n > M$) it can attain the communication lower bound (see Corollary 4.5). However, for reasonably sized matrices ($m, n < M$) the blocked algorithm is sub-optimal with respect to its communication costs.

In the late 1990s, both Toledo [142] and Gustavson [83] independently showed that using recursive algorithms can reduce communication costs. The analysis in [142] shows that the recursive LU (RLU) algorithm moves asymptotically fewer words than the algorithm in LAPACK when $m < M$ (though latency cost is not considered in that work). In fact, the RLU algorithm attains the bandwidth cost lower bounds. Furthermore, RLU is cache oblivious, so it minimizes bandwidth cost for any fast memory size and between any pair of successive levels of a memory hierarchy.

Motivated by the growing latency cost on both sequential and parallel machines, Grigori, Demmel, and Xiang [80] considered bandwidth and latency cost metrics and presented an algorithm called Communication-Avoiding LU (CALU) that minimizes both. In order to attain the lower bound for latency cost (proved in that paper via reduction from matrix multiplication–see Section 3.2.1), the authors used the block-contiguous layout and introduced tournament pivoting as a new and different scheme than partial pivoting. The tournament pivoting scheme makes different pivoting choices than partial pivoting and is theoretically less stable (though the two schemes are equivalent in a weak sense and have similar characteristics in practice [80]). The drawbacks to CALU are that it requires knowledge of the fast memory size for both algorithm and data layout (*i.e.*, it is not cache oblivious), and that, because of its youth, tournament pivoting does not enjoy the same confidence from the numerical community as partial pivoting (see [144], for example).

Making the RLU algorithm latency optimal had been an open problem for a few years. For example, arguments are made in [24] and [80] that RLU is not latency optimal for several different fixed data layouts. In [32], using a technique called "shape-morphing," we show that attaining communication optimality, being cache oblivious, and using partial pivoting are all simultaneously achievable.

7.1.5 QR Decomposition

The QR decomposition is commonly used for solving least squares problem, but because of its numerical stability, it has applications in many other computations such as eigenvalue and singular value decompositions and many iterative methods. While there are several approaches to computing a QR factorization, including Gram-Schmidt orthogonalization and Cholesky-QR, we focus in this section on those approaches that use a sequence of orthogonal transformations (*i.e.*, Householder transformations or Givens rotations) because they are the most numerically stable.

The history of reducing communication costs for QR decomposition is very similar to that of LU. The algorithm in LAPACK (`geqrf`) is based on the Householder QR algorithm (see [57, Algorithm 3.2] for example), computes one Householder vector per column, and also uses blocking to cast most of the computation in terms of matrix multiplication. However, this form of blocking is more complicated than in the case of LU and is based on the ideas of [42, 131]. While the blocking requires extra flops, the cost is only a lower order term. Though the algorithm in LAPACK is much more communication-efficient than the column-by-column approach, it still does not minimize bandwidth cost for reasonably sized matrices ($\sqrt{M} < n < M$), and the column-major data layout prevents latency optimality. Note that the representation of the Q factor is compactly stored as the set of Householder vectors used to triangularize the matrix (*i.e.*, one Householder vector per column of the matrix).

Shortly after the rectangular-recursive algorithms for LU were developed, a similar algorithm for QR was devised in [66]. As in the case of LU, this algorithm is cache oblivious and minimizes words moved (but not necessarily messages). It also computes one Householder vector per column. However, the algorithm performs a constant factor more flops than Householder QR, requiring about 17% more arithmetic for tall-skinny matrices and about 30% more for square matrices. To address this issue, the authors present a hybrid algorithm which combines the ideas of the algorithm in LAPACK and the rectangular-recursive one. The hybrid algorithm involves a parameter that must be chosen correctly (relative to the fast memory size) in order to minimize communication, so it is no longer cache oblivious.

Later, another recursive algorithm for QR was developed in [69]. The recursive structure of the algorithm involves splitting the matrix into quadrants instead of left and right halves, more similar to the recursive Cholesky algorithm than the previous rectangular-recursive LU and QR algorithms. Because recursive calls always involve matrix quadrants, the algorithm maps perfectly to the block-recursive data layout. Indeed, with this data layout, the algorithm minimizes both words and messages and is cache oblivious. Unfortunately, the algorithm involves forming explicit orthogonal matrices rather than working with their compact representations, which ultimately results in a constant factor increase in the number of flops of about $3\times$. It is an open question whether this algorithm can be modified to reduce this computational overhead.

At nearly the same time as the development of the CALU algorithm and tournament pivoting, a similar blocked approach for QR decomposition, called communication-avoiding QR (CAQR) was designed in [62]. CAQR maps to the block-contiguous data layout and minimizes both words and messages in the two-level model, but because it requires the algorithmic and data layout block size to be chosen correctly, it is not cache oblivious. Interestingly, it also requires a new representation of the Q factor. While just as compact as in the conventional Householder QR, the new representation varies with an internal characteristic of the algorithm (*i.e.*, the shape of the reduction trees). The ideas behind CAQR first appear in [75], and include [48, 82, 66]; see [62] for a more complete list of references. Note that the CAQR algorithm satisfies the assumptions of Theorem 4.25—it maintains forward progress and needs not compute T matrices of dimension 2 or greater—and attains both the bandwidth cost lower bound stated in the theorem as well as the latency lower bound corollary.

As explained in [32], our shape-morphing technique can be applied to the rectangular-recursive QR algorithm of [66] to obtain similar results as in the case of LU decomposition. Shape-Morphing QR is both communication optimal and cache oblivious, though it suffers from the same increase in computational cost as the original rectangular-recursive algorithm. Again, a hybrid version reduces the flops at the expense of losing cache obliviousness.

We also note that rank-revealing QR is an important variant of QR decomposition. While the conventional QR with column pivoting approach suffers from high communication costs, there do exist communication-optimal algorithms for this computation. See [61] for an application of the tournament pivoting idea of CALU to column pivoting within rank-revealing QR, and see [17, Algorithm 1] for a randomized rank-revealing QR algorithm that requires efficient (non-rank-revealing) QR decomposition and matrix multiplication subroutines.

7.1.6 Symmetric Eigendecomposition and SVD

The processes for determining the eigenvalues and eigenvectors of a symmetric matrix and the singular values and singular vectors of a nonsymmetric matrix are computationally similar. In both cases, the standard approach is to reduce the matrix via two-sided orthogonal transformations (stably preserving the eigenvalues or singular values) to a condensed form. In the symmetric case, this condensed form is a tridiagonal matrix; in the nonsymmetric case, the matrix is reduced to bidiagonal form. Computing the eigenvalues or singular values of these more structured matrices is much cheaper (both in terms of computation and communication) than reducing the full matrices to condensed form, so we do not consider this phase of computation here. The most commonly used tridiagonal and bidiagonal solvers include MRRR, Bisection/Inverse Iteration, Divide-and-Conquer, or QR iteration (see [63], for example). After both eigen- or singular values and vectors are computed for the condensed forms, the eigen- or singular vectors of the original matrix can be computed via a back-transformation, by applying the orthogonal matrices that transformed the dense matrix to tri- or bidiagonal. See Section 10.1.1 for more details in the symmetric case.

LAPACK's routines for computing the symmetric eigendecomposition (`syev`) and SVD (`gesvd`) use a similar approach to the LU and QR routines, blocking the computations to cast work in terms of calls to matrix multiplication. However, because the transformations are two-sided, a constant fraction of the work is cast as BLAS-2 operations, like matrix-vector multiply, which are communication inefficient. As a result, these algorithms do not minimize bandwidth or latency costs, for any matrix dimension; they require communicating $\Theta(n^3)$ words, meaning the data re-use achieved for a constant fraction of the work is as low as $O(1)$.

In 2000, Bischof, Lang, and Sun [38, 43] proposed a two-step approach to reducing a symmetric matrix to tridiagonal form known as Successive Band Reduction (SBR): first reduce the dense matrix to band form and then reduce the band matrix to tridiagonal. The advantage of this approach is that the first step can be performed so that nearly all of the computation is cast as matrix multiplication; that is, the data re-use can be $\Theta(\sqrt{M})$, which is communication optimal. The drawback is that reducing the band matrix to tridiagonal form is a difficult task, requiring $O(n^2 b)$ flops (as opposed to $O(nb^2)$ flops in the case of the symmetric-indefinite linear solver) and complicated data dependencies. Chapter 10 and [30] address performing this reduction in a communication-efficient manner.

If only eigenvalues are desired, then the two-step approach applied to a dense symmetric matrix performs asymptotically the same number of flops as the standard approach used in LAPACK. If eigenvectors are also desired, then the computational cost of the back-transformation phase is higher for the two-step approach by a constant factor. In terms of the communication costs, the two-step approach can be much more efficient, matching the lower bound of Corollary 4.26 and Theorem 4.27 (see [17, Section 6] for the bandwidth cost analysis). In order to minimize communication across the entire computation, the two-step approach also requires a communication-efficient tall-skinny QR decomposition (during the first step) and one of the algorithms proposed in Chapter 10 (for the second step). All of the SBR algorithms designed for the symmetric eigenproblem can be adapted for computing the SVD in the nonsymmetric case.

Another communication-optimal approach is to use the spectral divide-and-conquer algorithms described in Section 7.1.7, adapted for symmetric matrices or computing the SVD. In the symmetric case, a more efficient iterative scheme is presented in [116]. These approaches require efficient QR decomposition and matrix multiplication algorithms and perform a constant factor more computation than the reduction approaches.

7.1.7 Nonsymmetric Eigendecomposition

The standard approach for computing the eigendecomposition of a nonsymmetric matrix is similar to the symmetric case: orthogonal similarity transformations are used to reduce the dense matrix to a condensed form, from which the Schur form is computed. In the nonsymmetric case, the condensed form is an upper Hessenberg matrix (an upper triangular matrix with one nonzero subdiagonal), and QR iteration (with some variations) is typically used to annihilate the subdiagonal. In this case, the amount of data in the condensed form is asymptotically the same as the original matrix (about $n^2/2$ versus n^2), and $\Theta(n^3)$ computation is required to obtain Schur form from a Hessenberg matrix (as opposed to the symmetric case, where the data and computation involved in solving the tridiagonal eigenproblem are lower order terms). Thus, in determining the communication cost of the overall algorithm, we cannot ignore the second phase of computation as in Section 7.1.6.

LAPACK's routine for the nonsymmetric eigenproblem (`geev`) takes the reduction approach and moves $O(n^3)$ words in the reduction phase and $O(n^3)$ words in the QR iteration, so it is suboptimal both in terms of words and messages. While there have been approaches to reduce communication costs in the reduction phase in a manner similar to SBR, reducing first to band-Hessenberg and then to Hessenberg form (see [96]), it is an open question whether an asymptotic reduction is possible and the lower bound of Corollary 4.26 and Theorem 4.27 is attainable. Even if the reduction phase can be done optimally, it is also an open question whether QR iteration can be done in an equally efficient manner. Some work on reducing communication for multi-shift QR iteration in the sequential model appears in [113].

Because of the difficulties of the reduction approach, we consider a different approach for computing the nonsymmetric eigendecomposition, called spectral divide-and-conquer. In this approach, the goal is to compute an orthogonal similarity transformation which transforms the original matrix into a block upper triangular matrix, thereby generating two smaller subproblems whose Schur form can be combined to compute the Schur form of the original matrix. While there are a variety of spectral divide-and-conquer methods, we focus on the one proposed in [13], adapted in [58], and further developed in [17]. This approach relies on a randomized rank-revealing QR factorization and communication-optimal algorithms for QR decomposition and matrix multiplication. Under mild assumptions, the algorithm asymptotically minimizes communication (and is cache oblivious if the QR decomposition and matrix multiplication algorithms are) but involves a constant factor more computation than the reduction and QR iteration approach. The algorithm can be applied to the generalized eigenproblem as well as symmetric variants and the SVD. See [17] for full details of the algorithm and its communication costs.

Note also that to form the eigendecomposition from Schur form requires computing the eigenvectors of a (quasi-)triangular matrix. The LAPACK routine for this computation (`trevc`) computes one eigenvector at a time and does not minimize communication. A communication-optimal, blocked algorithm is presented in [17, Section 5].

7.2 Fast Linear Algebra

A depth-first traversal of the recursion tree given by Strassen's original algorithm minimizes bandwidth cost in the sequential model. See [26, Section 1.4.1] for the cost recurrence and solution. When the corresponding recursive data layout is used, the algorithm also minimizes latency cost.

Demmel, Dumitriu, and Holtz [58] showed that nearly all of the fundamental algorithms in dense linear algebra can be executed with asymptotically the same number of flops as matrix multiplication. Although the stability properties of fast matrix multiplication are slightly weaker than those of classical matrix multiplication, the authors show in [59] that all fast matrix multiplication is stable. Further, in [58] they show that fast linear algebra can be made stable at the expense of only a polylogarithmic (*i.e.*, polynomial in $\log n$) factor increase in cost. That is, to maintain stability, one can use polylogarithmically more bits to represent each floating point number and to compute each flop. While this increases the time to perform one flop or move one word, it does not change the number of flops computed or words moved by the algorithm.

The bandwidth cost analysis for the algorithms presented in [58] is given in [29]. While stability and computational complexity were the main concerns in [58], in [29] the bandwidth cost of the linear algebra algorithms is shown to match the lower bound of Corollary 6.5. To minimize latency costs, the analysis in [29] must be combined with the shape-morphing technique of [32].

7.3 Conclusions and Future Work

While there exist communication-optimal algorithms for all of the computations discussed in this chapter and presented in Table 7.1, much work remains to be done. In some cases, implementations of theoretically optimal algorithms have demonstrated performance improvements over previous algorithms; in others, implementations are still in progress. We highlight in this section possible future directions of algorithmic improvement—finding ways to reduce costs by constant factors or develop other theoretically optimal algorithms that might perform more efficiently in practice.

For example, it may be possible to minimize both words and messages for one-sided factorizations without relying on tournament pivoting, Householder reduction trees, or the block-Aasen algorithm by using more standard blocked algorithms (as in LAPACK) and adding a second level of blocking. This could produce a communication-optimal LDL^T algorithm. Other open problems with respect to QR decomposition include reconstructing Householder vectors from the tree representation of TSQR and modifying the algorithm of Frens and Wise to be more computationally efficient.

Because there are many variants of eigenvalue and singular value problems, several large-constant improvements are possible, particularly in the cases of computing eigenvectors of a symmetric matrix and singular vectors of a nonsymmetric matrix. The communication-optimal nonsymmetric eigensolver also suffers from high computational costs and requires more optimization to be competitive in practice.

Finally, the fast linear algebra algorithms are asymptotically as efficient as fast matrix multiplication, but they have never been demonstrated to perform better than classical algorithms in practice. One challenge to minimizing the constant factors is to optimize the rectangular matrix multiplication subroutines, using either square or rectangular fast matrix multiplication algorithms.

Chapter 8

Parallel Algorithms and their Communication Costs

While Chapter 7 summarizes algorithms for the sequential case, we focus in this chapter on parallel algorithms. The main contribution of the chapter is a comprehensive (though certainly not exhaustive) summary of communication-optimal parallel algorithms for dense linear algebra computations. Again, we provide references to the papers providing algorithmic details, communication cost analyses, and demonstrated performance improvements. We consider the same set fundamental computations—BLAS, Cholesky and symmetric-indefinite factorizations, LU and QR decompositions, eigenvalue and singular value decompositions—and compare the best parallel algorithms with the lower bounds established in Chapter 4. The following chapters, Chapters 9–11, will focus on algorithms for particular computations.

Recall the distributed-memory parallel memory model presented in Section 2.2.2. We consider communication among a set of P processors, all connected by a fully connected network, and we track both words and messages communicated by the parallel machine along the critical path(s) of the algorithm. All of the algorithms discussed in this chapter assume a block-cyclic data distribution (see Section 2.3.2), where a block size of 1 gives a cyclic distribution and a block size of $n/\sqrt{P}$ gives a blocked distribution.

The rest of this chapter is divided into three sections. The first two sections focus on classical algorithms: those that use no more than a constant factor more than the minimum amount of local memory required (Section 8.1) and those that use asymptotically more memory to reduce communication costs (Section 8.2). Section 8.3 considers fast linear algebra computations.

The main contributions to the set of communication-optimal parallel algorithms, appearing either in this thesis or in manuscripts written with various sets of coauthors, are as follows:

- communication cost analysis for minimal-memory parallel Cholesky decomposition algorithms [24],

- first minimal-memory communication-avoiding parallel algorithms for the symmetric eigendecomposition and SVD [30] (see Chapter 10),
- first minimal-memory communication-optimal parallel algorithm for the nonsymmetric eigendecomposition [17], and
- first communication-optimal parallel algorithm for parallelizing Strassen's and other fast matrix multiplication algorithms [20, 105] (see Chapter 11).

Table 8.1 is an updated version of Table 6.2 in [28], and some of the discussion here is based on that paper, written with coauthors James Demmel, Olga Holtz, and Oded Schwartz. The material on Cholesky decomposition in Section 7.1.2 is based on [24], written with the same set of coauthors. Section 8.3 includes a summary of Chapter 11 and [20], written written with James Demmel, Benjamin Lipshitz, Olga Holtz, and Oded Schwartz.

8.1 Classical Linear Algebra (with Minimal Memory)

In this section, we discuss the current state-of-the-art for algorithms that perform the classical $O(n^3)$ computations for dense matrices on parallel machines, assuming no more than a constant factor of extra local memory is used. For each of the computations considered here, we can compare the communication complexity of the algorithms to the lower bounds presented in Chapter 4, where we fix the local memory size to $M = \Theta(n^2/P)$. Table 8.1 summarizes the communication-optimal algorithms in this case for the most fundamental dense linear algebra computations. Another term for these minimal memory algorithms is "2D," which was first used to distinguish minimal memory matrix multiplication algorithms from so-called "3D" algorithms that do use more than a constant factor of extra memory. In these 2D algorithms, the processors are organized in a two-dimensional grid, with most communication occurring within processor rows or columns.

Recall the lower bounds that applies to these dense computations, where the number of g_{ijk} operations is $G = \Theta(n^3/P)$. For these values of G and M, the lower bound on the number of words communicated by any processor is $\Omega(n^2/\sqrt{P})$, and the lower bound on the number of messages is $\Omega(\sqrt{P})$. In order for an algorithm to be considered communication-optimal, we require that its communication complexity be within a polylogarithmic (in P) factor of the two lower bounds and that it performs no more than a constant factor more computation than alternative algorithms (*i.e.*, the parallel computational cost is $\Theta(n^3/P)$).

The asymptotic communication costs for ScaLAPACK algorithms are given in [44, Table 5.8]. Note that the bandwidth costs for those algorithms include a $\log P$ factor due to the assumption that collective communication operations (*e.g.*, reductions and broadcasts) are performed with tree-based algorithms. Better algorithms exist for these collectives that do not incur the extra factor in bandwidth cost. For example, a broadcast can be performed with a scatter followed by an all-gather (see [50] for more details). Thus, the extra logarithmic

Computation	Minimizes Words	Minimizes Messages
BLAS-3	[3, 44, 49, 71]	[3, 44, 49, 71]
Cholesky	[44]	[44]
Symmetric Indefinite	[14, 44]	[14]
LU	[44, 80]	[80]
QR	[44, 62]	[62]
Sym Eig and SVD	[17, 30, 44]	[17, 30]
Nonsym Eig	[17]	[17]

Table 8.1: Parallel classical algorithms attaining communication lower bounds assuming minimal memory is used. That is, these algorithms have computational cost $\Theta(n^3/P)$ and use local memory of size $O(n^2/P)$. We separately list algorithms that minimize only the number of words moved and algorithms that also minimize the number of messages.

factor is not inherent in the algorithm, only in the way collective operations were first implemented in SCALAPACK.

8.1.1 BLAS Computations

As in the sequential case, the lower bounds of Section 4.1.2.1 apply to all three levels of BLAS computations, but the only parallel algorithms that can attain the bounds are BLAS-3 routines. ScaLAPACK [44] uses the Parallel BLAS library, or PBLAS, which has algorithms for matrix multiplication (and its variants) and triangular solve that minimize both words and messages. The history of communication-optimal matrix multiplication goes back to Cannon [49]. While Cannon's algorithm is asymptotically optimal, a more robust and tunable algorithm known as SUMMA [3, 71] is more commonly used in practice. For a more complete summary of minimal memory, or 2D, matrix multiplication algorithms, see [95, Section 4].

8.1.2 Cholesky Decomposition

ScaLAPACK's parallel Cholesky routine (`pxposv`) minimizes both words and messages on distributed-memory machines. See [24] for a description of the algorithm and its communication analysis. Note that the bandwidth cost in that paper includes a $\log P$ factor that can be removed with a more efficient broadcast routine.

8.1.3 Symmetric-Indefinite Decompositions

ScaLAPACK's parallel LDL^T routine (`pxsysv`) minimizes words moved, but it is not latency optimal. For the sequential case, Chapter 9 and [14] present an alternate communication-optimal symmetric-indefinite factorization based on a blocked version of Aasen's LTL^T factorization [1] (where T is tridiagonal). While neither Chapter 9 or [14] address the parallel case, the algorithm can be parallelized to minimize communication in the minimal memory case. This algorithm computes the factorization in two steps, first reducing the symmetric matrix to band form and then factoring the band matrix. The first step requires the use of a communication-efficient tall-skinny LU decomposition routine (as used in CALU in Section 8.1.4). The second step can be performed naively with a nonsymmetric band LU factorization (with no parallelism) with computational and communication costs that do not asymptotically exceed the costs of the first step. We leave the details of the parallelization of the reduction to band form and a more complete consideration of efficient parallel methods for factoring the band matrix to future work.

8.1.4 LU Decomposition

As in the symmetric-indefinite case, ScaLAPACK's LU routine (`pxgesv`) minimizes the number of words moved but not the number of messages. Grigori, Demmel, and Xiang [80] propose a communication-optimal algorithm known as Communication-Avoiding LU (CALU) which uses "tournament pivoting," a scheme that makes different pivoting decisions from partial pivoting. The theoretical numerical stability guarantees are slightly weaker for CALU than for algorithms using partial pivoting, though in practice both approaches show similar behavior (see [80] for more details). The use of tournament pivoting allows the overall algorithm to reach both bandwidth and latency cost lower bounds.

In the sequential case, partial pivoting can be maintained while still minimizing both words and messages using a technique known as shape-morphing [32]. Unfortunately, the idea of shape-morphing is unlikely to yield the same benefits in the parallel case. Choosing pivots for each of n columns lies on the critical path of the algorithm and therefore must be done in sequence. Each pivot choice either requires at least one message or for the whole column to reside on a single processor. This seems to require either $\Omega(n)$ messages or $\Omega(n^2)$ words moved, which both asymptotically exceed the respective lower bounds.

8.1.5 QR Decomposition

ScaLAPACK's QR routine (`pxgeqrf`) also minimizes bandwidth cost but not latency cost. It is a parallelization of the LAPACK algorithm, using one Householder vector per column. At nearly the same time as the development of the CALU algorithm, Demmel, Grigori, Hoemmen, and Langou [62] developed the Communication-Avoiding QR (CAQR) algorithm that minimizes both words and messages. Note that the CAQR algorithm satisfies the assumptions of Theorem 4.25—it maintains forward progress and needs not compute T matrices of dimension 2 or greater—and attains both the bandwidth cost lower bound stated in the theorem as well as the latency lower bound corollary. The principal innovation of parallel CAQR is the factorization of a tall-skinny submatrix using only one reduction, for a cost of $O(\log P)$ messages rather than communicating once per column of the submatrix. The algorithm for tall-skinny matrices is called TSQR and is an important subroutine not only for general QR decomposition but also many other computations (see [114], for example). In order to obtain this reduction, the authors abandoned the conventional scheme of computing one Householder vector per column and instead use a tree of Householder transformations. This results in a different representation of the orthogonal factor, though it has the same storage and computational requirements as the conventional scheme. It is possible to perform TSQR and recover the conventional storage scheme without asymptotically increasing communication costs, but we leave details of the approach to future work.

The TSQR operation is effectively a reduction operation, and in the distributed memory parallel case, the optimal reduction tree is binary. Applied to an $m \times n$ matrix such that $m/P > n$, the communication costs of the reduction are $O(n^2 \log P)$ words and $O(\log P)$ messages. As mentioned in the beginning of Section 8.1, it is possible to remove the logarithmic factor in the bandwidth cost for some simple reductions; it is an open question whether this is possible with TSQR.

As in the sequential case, for communication-optimal algorithms performing rank-revealing QR decompositions, see [61] and [17, Algorithm 1].

8.1.6 Symmetric Eigendecomposition and SVD

As in the case of one-sided factorizations, ScaLAPACK's routines for the two-sided factorizations for the symmetric eigendecomposition (`pxsyev`) and SVD (`pxgesvd`) minimize bandwidth cost but fail to attain the latency cost lower bound. However, by using the two-step SBR approach described in Section 7.1.6, we can minimize both words and messages. In the two-step approach, the first step requires an efficient parallel tall-skinny QR factorization, like TSQR. For the second step, the use of the parallel algorithm given in Chapter 10 and [30] ensures the overall algorithm achieves the lower bounds.

8.1.7 Nonsymmetric Eigendecomposition

For the nonsymmetric eigenproblem, ScaLAPACK's routine (`pxgeev`) minimizes neither words nor messages. As in the sequential case, it is an open problem to minimize communication using the standard approach of reduction to Hessenberg form followed by Hessenberg QR iteration. Ongoing algorithmic and implementation development on the ScaLAPACK code has been improving the communication costs and speed of convergence; see [77] for details.

For the purposes of minimizing communication, we consider a different approach based on spectral divide-and-conquer. As in the sequential case, by using the method of [13, 17, 58], we can minimize both words and messages with the use of optimal parallel QR decomposition and matrix multiplication subroutines (see [17] for the communication analysis). Also, computing the eigenvectors from Schur form requires an optimal algorithm which is presented in [17, Section 5].

8.2 Classical Linear Algebra (with Extra Memory)

The lower bounds proved in Chapter 4 depend on the number of flops performed and the size of the local memory. In Section 8.1 we assume the local memory to be fixed at $\Theta(n^2/P)$; however, the actual amount of available local memory is a machine parameter rather than dependent on the problem size. Furthermore, the communication lower bound is proportional to the *inverse* of the square root of the local memory size, so increasing the local memory size (potentially) decreases the communication required of a computation. Thus, the lower bounds suggest that by using local memory, we can decrease the amount of communication performed. Note that the existence of memory-independent lower bounds, given in Section 6.2, provide a limit on how much this tradeoff can be exploited. In this section, we discuss classical algorithms that are able to take advantage of this opportunity.

8.2.1 Matrix Multiplication

The first algorithms to reduce the communication cost of parallel matrix multiplication by using extra memory were developed by Aggarwal, Chandra, and Snir (in the LPRAM model) [4] and Berntsen (on a hypercube network) [35]. Aside from using extra local memory, the main innovation in these algorithms is to divide the work for computing single entries in the output matrix among multiple processors; in the case of Cannon's and other 2D algorithms, a single processor computes all of the scalar multiplications and additions for any given output entry. See [16, Figure 2] for a visual classification of 1D, 2D, and 3D matrix multiplication algorithms based on the assignment of work to processors (note that the classification is applied to sparse algorithms in that paper). For a more complete summary of 3D dense matrix multiplication algorithms, see [95, Section 5].

The extra memory required for 3D algorithms is $O(n^2/P^{2/3})$, or $O(P^{1/3})$ times the minimal amount of memory required to store the input and output matrices. The communication savings, compared to 2D algorithms, is a factor of $O(P^{1/6})$ words and $\tilde{O}(P^{1/2})$ messages. However, these algorithms provide only a binary alternative to 2D algorithms—only if enough memory is available can 3D algorithms be employed. McColl and Tiskin [112] showed how to navigate the tradeoff continuously (in the BSPRAM model): for example, given local memory of size $\Theta(n^2/P^{2\alpha+2/3})$, their algorithm achieves bandwidth cost of $O(n^2/P^{2/3-\alpha})$ for $0 \leq \alpha \leq 1/6$. Later, Solomonik and Demmel [137] developed and implemented a practical version of the algorithm which generalizes the 2D SUMMA algorithm. Because their approach fills the gap between 2D and 3D algorithms, the authors coined "2.5D" to describe their algorithm.

Both of the algorithms exhibiting the continuous tradeoff are iterative algorithms. A recursive algorithm, which is a simple extension of the parallel Strassen algorithm given in Chapter 11 and [20], also achieves the same asymptotic communication costs (see also [105] for more details). The recursive algorithm is generalized to rectangular matrices in [60, 106], and the 2.5D algorithm [137] is generalized to rectangular matrices in [110], though it is not communication optimal in all cases.

8.2.2 Other Linear Algebra Computations

There are only a few known algorithms for the rest of linear algebra that are able to navigate the memory-communication tradeoff, but none as successfully as matrix multiplication. In particular, in the case of matrix multiplication, both bandwidth and latency costs can be reduced with the use of extra memory. In the case of all other algorithms for computations that involve more dependencies than matrix multiplication, there exists a second tradeoff between bandwidth and latency costs. That is, decreasing the bandwidth cost below $\Theta(n^2/P^{1/2})$ increases the latency cost above $\Theta(\sqrt{P})$ (or vice-versa). Therefore, the bandwidth cost and latency cost lower bounds are not simultaneously achieved for $M \gg \Omega(n^2/P)$. It remains an open question whether this is a necessary tradeoff.

Tiskin [140] presents a recursive algorithm in the BSP model for LU decomposition without pivoting that exhibits the same tradeoff between communication and synchronization in that model. This algorithm can be applied to symmetric positive-definite matrices, though it uses explicit triangular matrix inversion and multiplication (ignoring stability issues) and also ignores symmetry. Lipshitz [106] provides a similar algorithm for Cholesky decomposition, along with a recursive algorithm for triangular solve, that has the same asymptotic communication costs but exploits symmetry and maintains the stability of the minimal-memory algorithm. These Cholesky algorithms achieve a bandwidth cost of $O(n^2/P^\alpha)$ and latency cost of $O(P^\alpha)$ for $1/2 \leq \alpha \leq 2/3$.

In a later paper, Tiskin [141] incorporate pairwise pivoting into a new algorithm with the same asymptotic costs as in [140]. While pairwise pivoting is not generally considered stable enough in practice, the approach is generic enough to apply to QR decomposition based on Givens rotations. However, the algorithm seems to be of only theoretical value: for example, constants are generally ignored in the analysis, even for computational costs.

Solomonik and Demmel [137] devise a stable LU factorization algorithm that uses tournament pivoting and achieves the same asymptotic costs as the algorithm in [140]; they demonstrate competitive performance compared to minimal-memory LU algorithms. This work (both algorithm and implementation) is extended to the symmetric positive definite case in [72]. We leave the development of practical algorithms for QR and symmetric-indefinite factorizations (as well as eigenvalue and singular value decompositions) to future work.

8.3 Fast Linear Algebra

Compared to classical linear algebra, much less work has been done on the parallelization of fast linear algebra algorithms. Because there is not as rich a history of minimal-memory fast algorithms, we do not differentiate between minimal-memory and extra-memory algorithms in this section. Note that the fast algorithms that do exist are analogous to the classical extra-memory algorithms of Section 8.2: they can be executed with $M = O(n^2/P)$ if necessary but also can exploit extra memory at reduced communication costs if possible. While Strassen's matrix multiplication has been efficiently parallelized, both in theory and in practice, there are only a few theoretical results for other fast matrix multiplication algorithms and for other linear algebra computations.

McColl and Tiskin [112] present a parallelization of any Strassen-like matrix multiplication algorithm (see Definition 6.1) in the BSPRAM model that achieves a bandwidth cost of $O(n^2/P^{2/\omega_0-\alpha(\omega_0-2)})$ words using local memory of size $O(n^2/P^{2/\omega_0+2\alpha})$, where ω_0 is the exponent of the computational cost of the algorithm and $0 \leq \alpha \leq 1/2 - 1/\omega_0$. This algorithm is communication optimal for any local memory size; in particular, choosing maximum α achieves a minimal-memory algorithm, and choosing minimum α achieves the memory-independent lower bound (given in Theorem 6.8 for Strassen's algorithm). Chapter 11 (see also [20]) presents a more practical version of the algorithm, with communication analysis in the distributed-memory model, described in Section 2.2.2, as well as an implementation with performance results. For more detailed implementation description and performance results, see [105, 106].

We note that the recursive algorithms in Section 8.2.2 that use square matrix multiplication as a subroutine can benefit from a fast matrix multiplication algorithm. In particular, the triangular solve and Cholesky decomposition algorithms of [106, Section 5] and the algorithms of [141] attain the same computational costs as the matrix multiplication algorithm used and similarly navigate the communication-memory tradeoff. However, these algorithms have only been theoretically analyzed—no implementations exist yet. We leave the implementation of these known algorithms and the development of new algorithms for the rest of linear algebra to future work.

8.4 Conclusions and Future Work

As in the sequential case, there are many constant-factor improvements possible for the algorithms discussed in Section 8.1 and presented in Table 8.1. In particular, many implementations are in progress for demonstrating performance benefits of the new algorithms for symmetric-indefinite factorizations and computing the symmetric eigendecomposition and SVD.

Furthermore, the algorithms presented in Table 8.1 assume limitations on local memory that are often not necessary, especially in strong-scaling scenarios. Developing and optimizing extra-memory algorithms is an important area of research for developing scalable algorithms. As mentioned in Section 8.2.2, many open algorithmic problems remain.

Finally, the field of using fast parallel algorithms is wide open. While Strassen's algorithm has been shown to be effective in practice, making it robust for library deployment, applying it to other linear algebra computations, and discovering and developing even faster matrix multiplication algorithms are all areas of future work.

Chapter 9

Communication-Avoiding Symmetric-Indefinite Factorization

The focus of this chapter is a symmetric-indefinite factorization algorithm that minimizes communication costs. The main contributions are

- a new algorithm that is asymptotically more communication efficient than alternative algorithms,
- a proof of its backward stability (subject to a growth factor),
- a proof of its communication optimality in the sequential model, and
- numerical experiments measuring stability of the algorithm with both randomly generated matrices and matrices arising in real applications.

The algorithm is a block variant of Aasen's triangular tridiagonalization algorithm [1]. We designed the algorithm so that it can be implemented by a sequence of operations, each involving a constant number of $b \times b$ submatrices (blocks), where b is a tunable parameter. Most of these block operations perform $\Theta(b^3)$ arithmetic operations, which implies that the computation to communication ratio of the algorithm is $\Theta(b)$. Furthermore, we use a block-contiguous data layout (so that blocks are always contiguous in slow memory, see Section 2.3.1), implying that each block operation requires only $O(1)$ messages and $\Theta(b^2)$ words.

Matrix algorithms with such a structure usually perform well when implemented on sequential or shared-memory parallel computers. They can usually be adapted to distributed-memory parallel computers, but these adaptations are often intricate and far from trivial. The focus here is on the sequential block algorithm and its memory-hierarchy performance. See [15] for a description of a shared-memory parallel implementation of the algorithm and its performance. We do not discuss distributed-memory parallelization in this chapter.

Aasen's algorithm [1] factors

$$A = P^T L T L^T P,$$

where P is a permutation matrix selected for numerical stability, L is lower triangular (with ones on the diagonal and $|L_{ij}| \leq 1$), and T is symmetric and tridiagonal. The algorithm performs $n^3/3+o(n^3)$ arithmetic operations; it improves upon an earlier algorithm by Parlett and Reid that computes the same factorization in $2n^3/3 + o(n^3)$ operations [119]. Neither algorithm is used extensively; a few years later Kaufman and Bunch discovered a similar factorization that proved to be more popular, one in which the tridiagonal T is replaced by a matrix that is block diagonal with 2×2 and 1×1 blocks [47].

Like other early factorizations, the algorithms of Aasen and of Parlett and Reid are not communication efficient even for very simple memory hierarchies. If $M < n^2/8$, both algorithms transfer $\Theta(n^3)$ words between fast and slow memory, which is very inefficient. An implementation of the Bunch-Kaufman factorization that transfers only $O(\min(n^3, \frac{n^2}{M} \cdot n^2) = O(n^4/M)$ words was later discovered,[1] and this implementation was included in LAPACK [8]. More recently, Rozložník, Shklarski and Toledo discovered how to compute the Parlett-Reid-Aasen factorization with the same communication efficiency [125].

In this chapter, we describe and analyze a stable symmetric factorization algorithm that is communication avoiding; it requires only $O(n^3/\sqrt{M})$ words moved. In terms of communication, this is much more efficient than any existing symmetric indefinite factorization. However, the algorithm produces a T that is banded rather than tridiagonal. To achieve this communication efficiency, the half bandwidth of T is $\Theta(\sqrt{M})$. We also show that the resulting T can be factored further in a way that is also communication efficient, and the resulting factorization allows linear systems of equations to be solved quickly.

Our algorithm is fundamentally a block version of Aasen's algorithm, and we will refer to it as the block-Aasen algorithm. While the methodology of producing block matrix algorithms from element-by-element algorithms is well understood, applying it to this case proved to be challenging. The first block-Aasen algorithm that we designed was highly unstable. In Aasen's original algorithm, diagonal elements of T are computed by solving a scalar equation. In the block version, this scalar equation transforms into a linear system of equations whose solution is a diagonal block of T, which is symmetric. But the system itself is unsymmetric and the symmetry of the solution is implicit. When the system is solved in floating point arithmetic, the computed block of T can have a non-negligible skew-symmetric component in addition to its symmetric part, and this excites an instability. To address this difficulty, we designed an algorithm that produces a symmetric T even in floating point.

The rest of the chapter is organized as follows. We present the block-Aasen algorithm in Section 9.1. Section 9.2 analyzes the stability of the algorithm, and Section 9.3 its computation and communication complexity. Numerical experiments presented in Section 9.4 provide additional insights into the behavior of the algorithm. Section 9.5 presents our conclusions from this work.

[1]The $O(n^4/M)$ bound is attained when $M \geq n$. In this regime, the algorithm factors panels of roughly M/n columns. Updating a trailing submatrix of dimension $\Theta(n)$ after the factorization of $\Theta(n/(M/n))$ such panels transfers $\Theta(n^4/M)$ words. When $M < n$, the algorithm transfers $O(n^3)$ words; in this regime the fast memory has no significant beneficial effect.

All of the results in this chapter appear in [14], written with coauthors Dulceneia Becker, James Demmel, Jack Dongarra, Alex Druinsky, Inon Peled, Oded Schwartz, Sivan Toledo, and Ichitaro Yamazaki. The companion paper [15], written with the same coauthors, describes a shared-memory parallel implementation of the algorithm and presents its performance compared with several alternatives; we do not present that data here. This work received a Best Paper award at the 2013 International Parallel and Distributed Processing Symposium (IPDPS).

9.1 Block-Aasen Algorithm

To keep the notation simple, we initially ignore pivoting in the description of the algorithm. The algorithm factors the $n \times n$ matrix A into

$$A = LTL^T,$$

where L is unit lower triangular and T is symmetric and banded with half bandwidth b (*i.e.*, $T_{ij} = 0$ if $|i - j| > b$). The algorithm processes the matrices in aligned blocks of size $b \times b$ (except for the trailing blocks which might be smaller). The algorithm is a block version of Aasen's algorithm, so we view T as a block tridiagonal matrix with triangular blocks in the positions immediately adjacent to the main diagonal.

To describe the algorithm we must specify three auxiliary matrices. The first is a block upper-triangular matrix with symmetric diagonal blocks R that we require to satisfy

$$R^T + R = T.$$

The superdiagonal blocks in R are the same as the corresponding blocks in T, the diagonal blocks of R are scaled copies of those of T (with scaling $1/2$), and the subdiagonal blocks in R are zero (unlike in T, which is symmetric). The other two matrices are denoted by W and H and are required to satisfy

$$W = RL^T$$
$$H = TL^T.$$

Aasen's original algorithm also computes H (forming it was the key step that allowed Aasen to eliminate half the arithmetic operations from Parlett and Reid's algorithm), but it does not compute W.

We present the algorithm in the form of block-matrix equations each of which defines one or two sets of blocks in these matrices. The blocks that are computed from each equation are underlined, following the notation of [85, Section 11.2]. We use capital I and capital J to denote block indices, and we denote the block dimension of all the matrices by $N = \lceil n/b \rceil$. We denote blocks of matrices using indexed notation with block indices. For example, the submatrix that is specified by $A_{1+(I-1)b:Ib,1+(J-1)b:Jb}$ in scalar-index colon notation is denoted $A_{I,J}$.

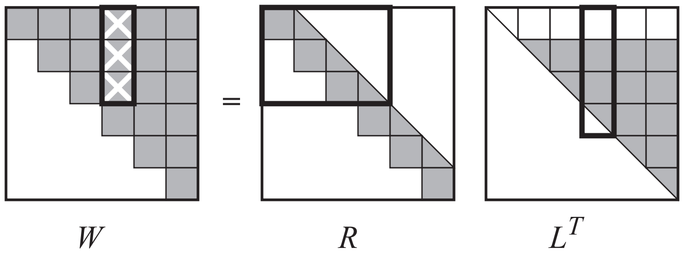


Figure 9.1: An illustration of computing superdiagonal blocks of W via matrix multiplication in Equation (9.1). Here $N = 6$ and $J = 4$. The blocks that participate in the equation are enclosed in thick rectangles, and the blocks that are computed using this equation are crossed. The same notation is used in other diagrams in this section.

The initialization step of the algorithm assigns

$$\underline{L_{1:N,1}} = (\text{identity matrix})_{1:N,1}.$$

That is, the first b columns of L have ones on the diagonal and zeros everywhere else. After this initialization, the algorithm computes a block column of each of the matrices in every step. Step J computes block column $J+1$ of L and block columns J of T, H, and W (diagonal blocks of W are never needed so they are not computed) according to the formulas:

$$\underline{W_{1:J-1,J}} = R_{1:J-1,1:J}(L_{J,1:J})^T \tag{9.1}$$

$$A_{J,J} = L_{J,1:J-1}W_{1:J-1,J} + (W_{1:J-1,J})^T(L_{J,1:J-1})^T + L_{J,J}\underline{T_{J,J}}(L_{J,J})^T \tag{9.2}$$

$$\underline{H_{1:J,J}} = T_{1:J,1:J}(L_{J,1:J})^T \tag{9.3}$$

$$A_{J+1:N,J} = L_{J+1:N,1:J}H_{1:J,J} + \underline{L_{J+1:N,J+1}}\,\underline{H_{J+1,J}} \tag{9.4}$$

$$H_{J+1,J} = \underline{T_{J+1,J}}(L_{J,J})^T. \tag{9.5}$$

9.1.1 Correctness

We now show that the algorithm is correct. Verifying that the blocks that are computed in each equation depend only on blocks that are already known is trivial. Therefore, we focus on showing that $A = LTL^T$ whenever L and T are computed in exact arithmetic. The analysis also constitutes a more detailed presentation of the algorithm.

Equation (9.1) computes a block column of W, except for the diagonal block, by multiplying two submatrices, using the equation $W = RL^T$, as shown in Figure 9.1. This guarantees that $W_{I,J} = (RL^T)_{I,J}$ for all $I < J$. The diagonal blocks of W are not computed and we define for convenience $W_{J,J} = (RL^T)_{J,J}$ for all J. Because all other blocks of W and RL^T are zero, $W = RL^T$.

Equation (9.2) computes a diagonal block of T by solving a two-sided triangular linear system. This linear system can be solved by one of the existing solvers which we describe

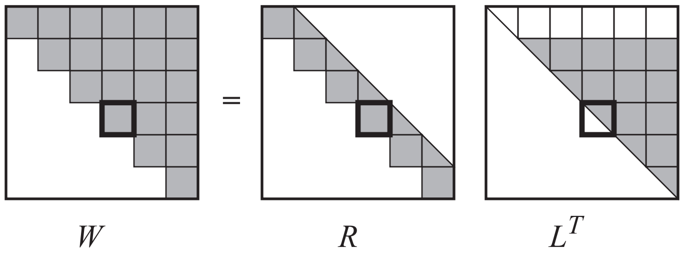


Figure 9.2: An expression for the diagonal blocks of W.

below. The right-hand side matrix in this system,

$$A_{J,J} - L_{J,1:J-1}W_{1:J-1,J} - (L_{J,1:J-1}W_{1:J-1,J})^T,$$

must be computed symmetrically; this may be done using the BLAS routine SYR2K [45]. The equation guarantees that

$$A_{J,J} = L_{J,1:J-1}W_{1:J-1,J} + (L_{J,1:J-1}W_{1:J-1,J})^T + L_{J,J}T_{J,J}(L_{J,J})^T.$$

By noting that

$$\begin{aligned} L_{J,J}T_{J,J}(L_{J,J})^T &= L_{J,J}(R_{J,J} + (R_{J,J})^T)(L_{J,J})^T \\ &= L_{J,J}R_{J,J}(L_{J,J})^T + (L_{J,J}R_{J,J}(L_{J,J})^T)^T \end{aligned}$$

and that the diagonal blocks of $W = RL^T$ are $W_{J,J} = R_{J,J}(L_{J,J})^T$, as shown in Figure 9.2, we can transform (9.2) into

$$\begin{aligned} A_{J,J} &= L_{J,1:J-1}W_{1:J-1,J} + (L_{J,1:J-1}W_{1:J-1,J})^T + L_{J,J}W_{J,J} + (L_{J,J}W_{J,J})^T \\ &= L_{J,1:J}W_{1:J,J} + (L_{J,1:J}W_{1:J,J})^T \\ &= (LW + (LW)^T)_{J,J}. \end{aligned}$$

Substituting $W = RL^T$ we obtain

$$\begin{aligned} A_{J,J} &= (LRL^T + (LRL^T)^T)_{J,J} \\ &= (L(R + R^T)L^T)_{J,J} \\ &= (LTL^T)_{J,J}. \end{aligned}$$

Equation (9.3) computes a block column of H, except for the subdiagonal block, by multiplying matrices, as shown in Figure 9.4. Equation (9.4) multiplies blocks of L and H, subtracts the product from a block of A, and factors the difference using an LU factorization. Equation (9.5) solves a triangular linear system with a triangular right-hand side to compute a subdiagonal block of T.

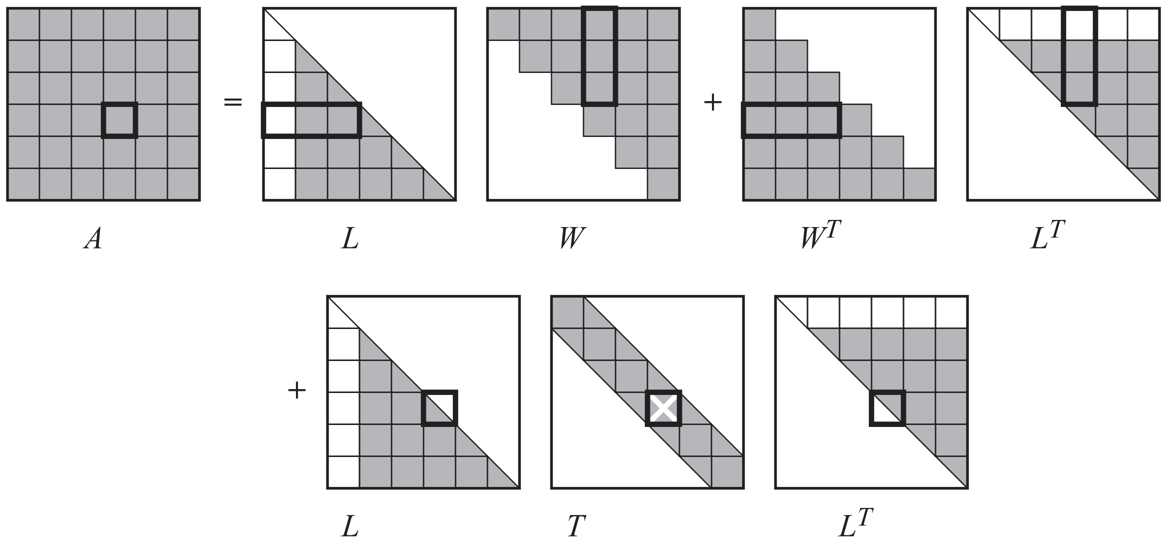


Figure 9.3: Computing a diagonal block of T in Equation (9.2) by updating the corresponding block of A and solving a two-sided triangular system. The letters below each matrix describe only the matrices involved in the expression for $A_{J,J}$; they do not constitute a matrix equation.

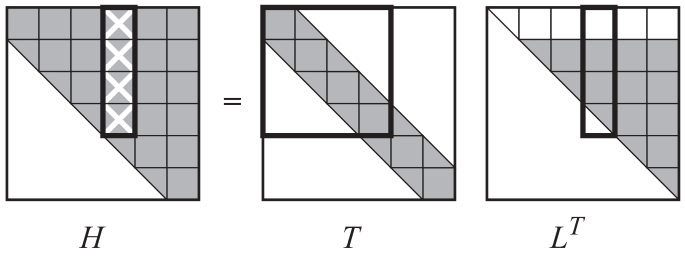


Figure 9.4: Computing blocks of H via matrix multiplication in Equation (9.3).

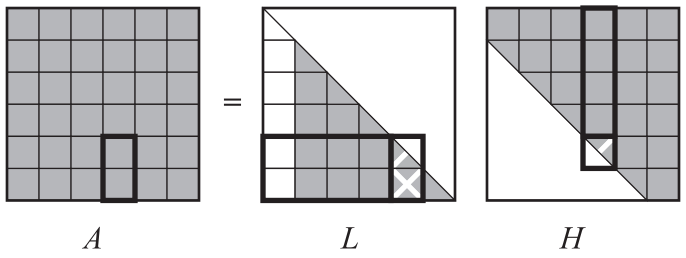


Figure 9.5: Computing a block column of L and a subdiagonal block of H in Equation (9.4) via the LU factorization of an updated submatrix of A.

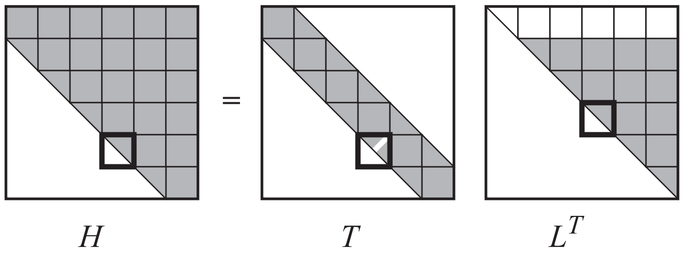


Figure 9.6: Computing a block of T by solving a triangular system in Equation (9.5).

Equations (9.3) and (9.5) guarantee that $H_{I,J} = (TL^T)_{I,J}$ for all $I \leq J$ and for all $I = J + 1$ respectively, and because all other blocks of H and TL^T are zero, $H = TL^T$. Equation (9.4) ensures that $A_{I,J} = (LH)_{I,J}$ for all $I > J$, and substituting $H = TL^T$ shows that $A_{I,J} = (LTL^T)_{I,J}$ for all $I > J$. Because both A and LTL^T are symmetric, this holds for all $I < J$ as well, and thus $A = LTL^T$.

9.1.2 Solving Two-Sided Triangular Linear Systems

We now describe the procedure that solves the two-sided triangular linear system in Equation (9.2). The method is not new; it is used to reduce symmetric generalized eigenproblems to standard eigenproblems and is available in LAPACK and ScaLAPACK under the name SYGST [44, 134]. Even though we are focused on SYGST in this paper, other solvers that produce a symmetric solution would also be suitable for the task. Examples of such solvers are the subroutine REDUC in EISPACK [111, 135] and the algorithms implemented in the Elemental library [120, 121]. Because we apply the solver to block-sized problems, its flops and communications costs do not have a substantial impact on the overall costs of the algorithm. The stability of the solver is important, but as long as it satisfies a bound similar to the one we prove for SYGST in Section 9.2, the impact on the overall block-Aasen algorithm is limited to the size of the constant in the backward stability bound.

To the best of our knowledge the stability of SYGST has not been previously analyzed. In order to analyze the algorithm we will now describe the relevant details of how it works. The equation that defines $T_{J,J}$ is of the form $LXL^T = B$ with a symmetric right-hand side B.[2] A trivial way to solve such systems is using a conventional triangular solver twice. That is, to first solve for $L^{-1}B$ and to then solve for $X = (L^{-1}B)L^{-T}$. This method produces a solution X that is not exactly symmetric in finite precision, and is thus not suitable for use in the block-Aasen algorithm. Another approach, which leads to an exactly symmetric X and which performs only half the arithmetic, is an algorithm that we now describe. We partition all of the matrices that we introduce in this section such that they are all 2×2

[2]This section uses self-contained notation, for simplicity. The matrix that we call L here is a diagonal block of the lower-triangular factor in the overall block-Aasen factorization. In addition, the auxiliary matrices H and W, the dimension of the problem n and the block size b, all of which we define later in this section, are also distinct.

block matrices with first diagonal blocks of dimensions $b \times b$ and second diagonal blocks of dimensions $(n-b) \times (n-b)$. To describe the algorithm we must define an auxiliary matrix Y, which we require to be a block upper triangular matrix with symmetric diagonal blocks that satisfies

$$X = Y^T + Y.$$

Such a matrix must have the form

$$\begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} = \begin{bmatrix} 0.5X_{11} & X_{12} \\ 0 & 0.5X_{22} \end{bmatrix}.$$

We also need two additional auxiliary matrices H and W, which we will require to satisfy

$$H = XL^T, \qquad\qquad W = YL^T.$$

The algorithm works by solving for the underlined blocks in the following equations:

$$\begin{aligned} B_{11} &= L_{11}\underline{X_{11}}(L_{11})^T & (9.6)\\ B_{12} &= L_{11}\underline{H_{12}} & (9.7)\\ H_{12} &= 0.5X_{11}(L_{21})^T + \underline{W_{12}} & (9.8)\\ W_{12} &= 0.5X_{11}(L_{21})^T + \underline{X_{12}}(L_{22})^T & (9.9)\\ B_{22} &= L_{21}W_{12} + (L_{21}W_{12})^T + L_{22}\underline{X_{22}}(L_{22})^T. & (9.10) \end{aligned}$$

The key in this algorithm is to compute $B_{22} - L_{21}W_{12} - (L_{21}W_{12})^T$ in (9.10) symmetrically, which allows the algorithm to compute X_{22} symmetrically as well. Note that the block $0.5X_{11}(L_{21})^T$ is computed twice; the algorithm trades off additional computation for a reduction in workspace requirements.

We derived the equations in (9.6)–(9.10) by considering specific blocks of the equations

$$B = LXL^T, \quad B = LH, \quad B = LW + (LW)^T, \quad H = Y^TL^T + W, \quad W = YL^T.$$

The derivation is described by diagrams in Figures 9.7–9.11.

We will now verify the correctness of the algorithm, meaning that $LXL^T = B$ whenever X is computed in exact arithmetic. The algorithm computes the diagonal and superdiagonal blocks of X and the superdiagonal blocks of H and W. The subdiagonal block of X is not computed because X is symmetric. The diagonal and subdiagonal blocks of H and W are also not computed, and thus we are free to define them so that our notation is simplified. We define the uncomputed blocks of H and W such that the corresponding blocks of the equations $H = XL^T$ and $W = YL^T$ hold.

We start by verifying that $(LXL^T)_{12} = B_{12}$. Equation (9.9) ensures that $W_{12} = (YL^T)_{12}$ and thus $W = YL^T$, due to the way we defined the uncomputed blocks of W. Equation (9.8) guarantees that $H_{12} = (Y^TL^T + W)_{12}$. Substituting $W = YL^T$ and noting that $Y^TL^T + YL^T = XL^T$ shows that $H_{12} = (XL^T)_{12}$ and thus $H = XL^T$, again due to our definition

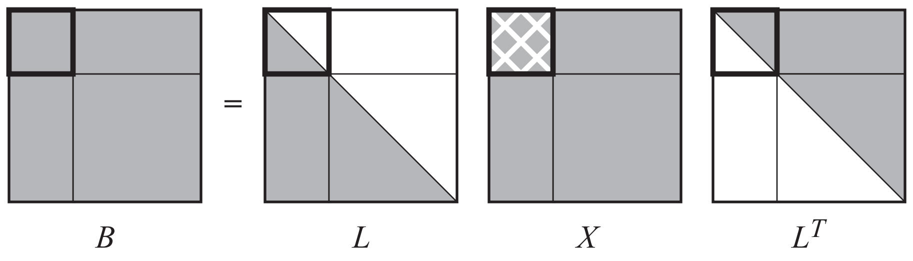


Figure 9.7: Computing the first diagonal block of X by solving a smaller two-sided triangular system in Equation (9.6).

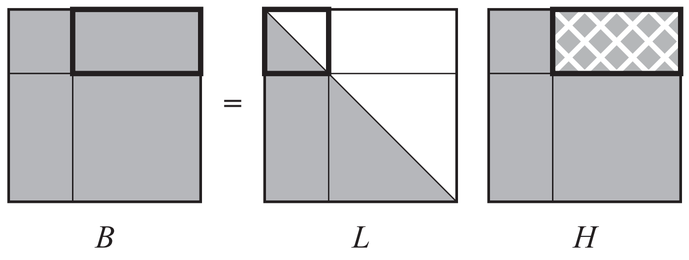


Figure 9.8: Solving a triangular system to compute the superdiagonal block of H in Equation (9.7).

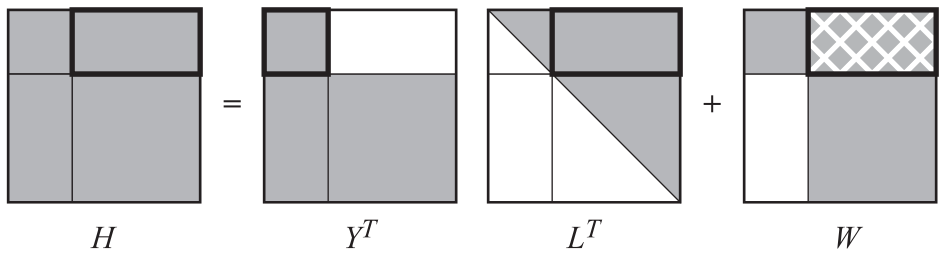


Figure 9.9: Computing the superdiagonal block of W in Equation (9.8) by updating the corresponding block of H.

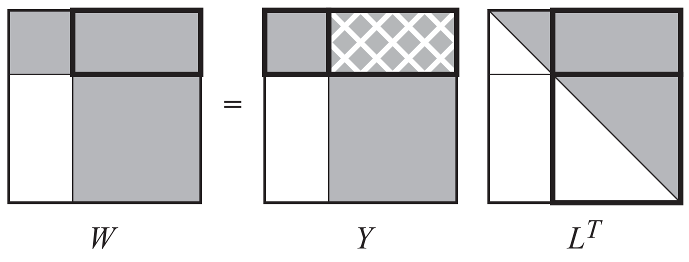


Figure 9.10: Computing the superdiagonal block of Y in Equation (9.9) by updating the corresponding block of W and solving a triangular system.

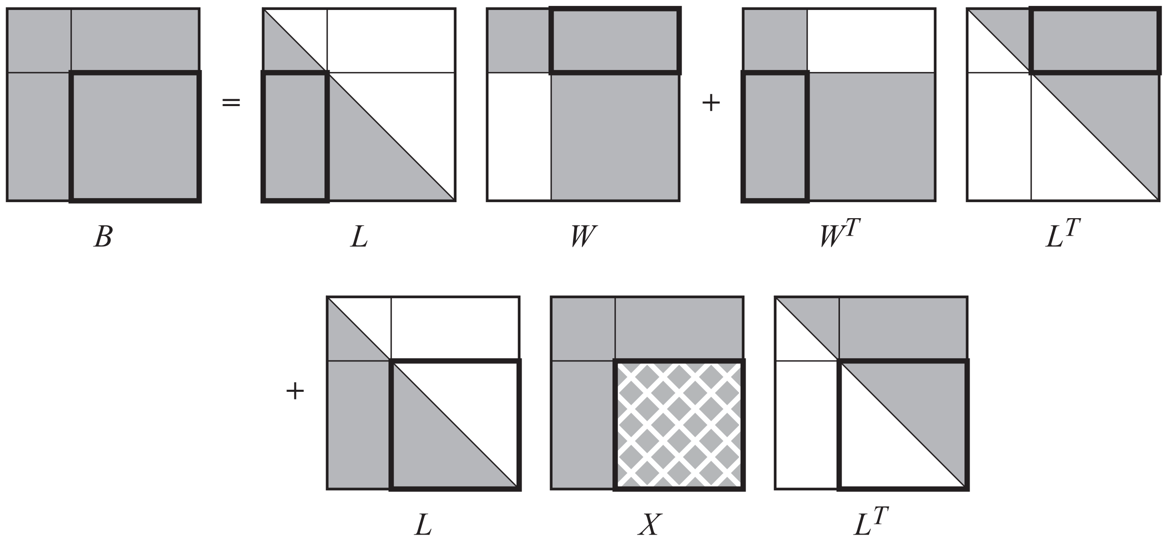


Figure 9.11: Computing the second diagonal block of X in Equation (9.10) by updating the corresponding block of B and solving a smaller two-sided triangular system.

of the uncomputed blocks of H. Finally, Equation (9.7) ensures that $B_{12} = (LH)_{12}$, and substituting $H = XL^T$ shows that $B_{12} = (LXL^T)_{12}$.

Next we verify that $(LXL^T)_{22} = B_{22}$ by transforming the equation in Equation (9.10):

$$\begin{aligned}
B_{22} &= L_{21}W_{12} + (L_{21}W_{12})^T + L_{22}X_{22}(L_{22})^T \\
&= L_{21}W_{12} + (L_{21}W_{12})^T + L_{22}Y_{22}(L_{22})^T + (L_{22}Y_{22}(L_{22})^T)^T \\
&= L_{21}W_{12} + (L_{21}W_{12})^T + L_{22}W_{22} + (L_{22}W_{22})^T \\
&= (LW + (LW)^T)_{22} \\
&= (LYL^T + LY^TL^T)_{22} \\
&= (L(Y + Y^T)L^T)_{22} \\
&= (LXL^T)_{22}.
\end{aligned}$$

Finally, Equation (9.6) explicitly ensures that $(LXL^T)_{11} = B_{11}$ and thus $LXL^T = B$.

We did not specify the dimensions of the blocks; different choices yield different algorithms. If we choose $b = 1$, we end up with an algorithm that computes the columns of X one at a time, in which (9.10) iterates over remaining columns. This version is called SYGS2 in LAPACK and ScaLAPACK. The costs in this partitioning are dominated by the triangular solve in (9.9) and the symmetric update in (9.10) which require $(n-i)^2$ and $2(n-i)^2$ flops, respectively. Thus, the leading term in flop cost is given by

$$F_1(n) = \sum_{i=1}^{n-1} 3(n-i)^2 = 3\sum_{i=1}^{n-1} i^2 = n^3 + o(n^3).$$

If instead of $b = 1$ we choose some fixed $b > 1$, we obtain SYGST, which works by computing a total of b columns of X at a time. Equation (9.6) here corresponds to a call to SYGS2 and Equation (9.10) iterates over remaining block columns. As long as $n \gg b$, the costs are again dominated by the triangular solve in (9.9) and the symmetric update in (9.10) which require $(n-ib)^2b$ and $2(n-ib)^2b$ flops, respectively (here i iterates over block columns). Thus, the leading term in the flop cost is given by

$$F_b(n) = \sum_{i=1}^{n/b-1} 3(n-ib)^2b = 3b^3 \sum_{i=1}^{n/b-1} i^2 = n^3 + o(n^3).$$

We can also formulate the algorithm recursively, with X_{11} being $\lfloor \frac{n}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor$. This recursive version is new and it is communication avoiding and cache oblivious even for large matrices (we use it only on blocks, so this is not useful for the blocked Aasen algorithm). Equations (9.6) and (9.10) are recursive calls. The triangular solves in steps (9.7) and (9.9) and the block multiplications in steps (9.8), (9.9) and (9.10) all contribute to the leading term in the flop cost. The product $X_{11}(L_{21})^T$ is a SYMM BLAS call which costs $2(n/2)^3$. The triangular solves are TRSM calls that cost $(n/2)^3$ each, and the product $L_{21}W_{12}$ is a GEMM call that costs $2(n/2)^3$ (we can subtract the transpose after computing and subtracting the product). If we store the $X_{11}(L_{21})^T$ and reuse it in both step (9.8) and step (9.9), the recurrence is

$$F_R(n) = 2F_R\left(\frac{n}{2}\right) + 6\left(\frac{n}{2}\right)^3$$

which again yields

$$F_R(n) = \frac{3}{4}n^3 \sum_{i=0}^{\log n-1} \left(\frac{1}{4}\right)^i = \frac{3}{4} \cdot \frac{4}{3}n^3 + o(n^3) = n^3 + o(n^3).$$

If we chose to recompute $X_{11}(L_{21})^T$ in order to run the algorithm in place, the flop count increases but is still $O(n^3)$.

9.1.3 Pivoting

Without pivoting, the algorithm can break down or become unstable, just like the classical element-wise Aasen algorithm. In the new algorithm, blocks $L_{J+1:N,J+1}$ and $H_{J+1,J}$ are computed using an LU factorization, and without pivoting, the factorization may fail to exist or may be unstable. Clearly, we need to pivot in Equation (9.4). It turns out that this stabilizes the algorithm, as in the element-wise algorithm.

We use row pivoting, meaning that step J factors

$$L_{J+1:N,J+1}H_{J+1,J} = P_J\left(A_{J+1:N,J} - L_{J+1:N,1:J}H_{1:J,J}\right),$$

where P_J is a permutation matrix, determined by partial pivoting, for example. There are several ways to compute this LU factorization in a way that ensures that the block-Aasen algorithm is communication avoiding; we list and analyze them in Section 9.3.2.2 below.

Once the LU factorization is computed, we also apply P_J to $L_{J+1:N,1:J}$ and to the trailing submatrix $A_{J+1:N,J+1:N}$. Applying the permutation to L and to the trailing submatrix is not trivial to do in a communication-avoiding way, especially since only the upper or lower triangle of the trailing submatrix is stored. The details are explained below, in Section 9.3.2.3.

9.1.4 Computing W and H

As we show in Section 9.3, the arithmetic and communication costs of our algorithm are asymptotically dominated by the computation that corresponds to Equation (9.4). Nevertheless, if optimizing the computation that corresponds to the other equations can yield any savings, then pursuing such optimizations would be desirable from a practical standpoint. It turns out that savings are possible in Equations (9.1) and (9.3). These equations state that individual blocks of H and W are computed according to the formulas:

$$\underline{H_{I,J}} = T_{I,I-1} \left(L_{J,I-1}\right)^T + T_{I,I} \left(L_{J,I}\right)^T + T_{I,I+1} \left(L_{J,I+1}\right)^T$$

and

$$\begin{aligned}\underline{W_{I,J}} &= R_{I,I} \left(L_{J,I}\right)^T + R_{I,I+1} \left(L_{J,I+1}\right)^T \\ &= 0.5\, T_{I,I} \left(L_{J,I}\right)^T + T_{I,I+1} \left(L_{J,I+1}\right)^T\end{aligned}$$

for all $I < J$. (We encourage the reader to review Figures 9.1 and 9.4 for a visualization of these relations.) The blocks $T_{I,I} \left(L_{J,I}\right)^T$ and $T_{I,I+1} \left(L_{J,I+1}\right)^T$ appear in these equations twice but need to be computed only once. Avoiding the recomputation of these blocks reduces the number of $b \times b$ matrix products required to compute W and H by a ratio of 5:3, thereby making the computation of W essentially free.

9.1.5 The Second Phase of the Algorithm: Factoring T

There are several single-pass algorithms that efficiently factor a banded symmetric matrix. All of these algorithms process $O(b)$ rows and columns at a time, so if we choose a small enough $b = \Theta(\sqrt{M})$, the total number of words moved is $O(nb)$, which makes them communication efficient (because the size of input and output is $O(nb)$).

Algorithms with these properties include the unsymmetric banded LU factorization with partial pivoting, Kaufman's retraction algorithm [99], and Irony and Toledo's snap-back algorithm [94]. Both retraction and snap-back algorithms preserve symmetry (and matrix inertia); the LU algorithm destroys the symmetry of the band. All three perform $O(nb^2)$ flops and produce a factorization that is essentially banded with bandwidth $O(b)$. All of these factorizations can be used to solve linear systems of equations using $O(bn)$ arithmetic per right-hand side and $O(bn)$ words moved (for up to $O(\sqrt{M})$ right-hand sides).

9.2 Numerical Stability

We analyze the stability of the factorization of PAP^T where P is the permutation matrix generated by the selection of pivots. We assume in the analysis that the matrix has been pre-permuted so the algorithm is applied directly to PAP^T (rather than to A) and that it never pivots. The sequence of arithmetic operations in such a run of the algorithm is identical to that of the pivoting version applied to A, except perhaps for the order of summation in inner products. Our analysis does not depend on this ordering so our results apply to the pivoting version. We will use the standard model of floating point computation and several well-known results regarding stability of fundamental operations (see Section 2.6).

9.2.1 Stability of the Two-Sided Triangular Solver

Our notation in this section is the same as in Section 9.1.2: we consider solving an $n \times n$ system and partition all of the matrices such that they are 2×2 block matrices with first diagonal blocks of dimensions $b \times b$ and second diagonal blocks of dimensions $(n-b) \times (n-b)$, where $1 \leq b \leq n-1$. The matrix X and the superdiagonal blocks of H and W represent the computed floating point matrices. We use Y to denote the exact matrix

$$Y = \begin{bmatrix} 0.5X_{11} & X_{12} \\ 0 & 0.5X_{22} \end{bmatrix}$$

and we define the diagonal and subdiagonal blocks of H and W such that the corresponding blocks of $H = XL^T$ and $W = YL^T$ hold.

Lemma 9.1. *If the two-sided $n \times n$ triangular system $LXL^T = B$ is solved in floating point arithmetic using any of the partitioned algorithms given in Section 9.1.2, then the computed X satisfies*

$$B = LXL^T + \Delta, \qquad |\Delta| \leq \gamma_{3n-1}|L||X||L^T|.$$

Proof. The proof is by strong induction on n. In the base case we are solving a 1×1 system and the bound clearly holds. To make the inductive step, we first show how the backward error Δ relates to the backward errors of the matrix equations used in the algorithms. We define the matrices $\Delta^{(t)}$ for $t = 1, 2, \ldots, 5$ such that

$$B = LH + \Delta^{(1)}, \quad H = Y^T L^T + W + \Delta^{(2)}, \quad W = YL^T + \Delta^{(3)}, \quad B = LW + (LW)^T + \Delta^{(4)}$$

and

$$H = XL^T + \Delta^{(5)}.$$

Substituting the third formula into the second one, we obtain

$$H = Y^T L^T + YL^T + \Delta^{(2)} + \Delta^{(3)} = XL^T + \Delta^{(2)} + \Delta^{(3)},$$

and then substituting this result into the first formula yields

$$B = LXL^T + \Delta^{(1)} + L(\Delta^{(2)} + \Delta^{(3)}).$$

Similarly, substituting the third formula into the fourth one yields

$$B = LXL^T + L\Delta^{(3)} + (L\Delta^{(3)})^T + \Delta^{(4)}.$$

Thus we have

$$\Delta^{(5)} = \Delta^{(2)} + \Delta^{(3)} \tag{9.11}$$

and two equations for the backward error of the solution of the system:

$$\Delta = \Delta^{(1)} + L\Delta^{(5)} \tag{9.12}$$

$$\Delta = L\Delta^{(3)} + (L\Delta^{(3)})^T + \Delta^{(4)}. \tag{9.13}$$

We're now ready to consider the computations involved in the algorithm. Recall from Section 9.1.2 that we partition all matrices into 2×2 blocks so that the top left block is $b \times b$ for some $1 \le b \le n-1$. Applying the lemmas of rounding-error analysis (given in Section 2.6) and the inductive hypothesis to equations (9.6)–(9.10) allows us to derive the bounds

$$|\Delta_{11}| \le \gamma_{3b-1}(|L||X||L|^T)_{11} \tag{9.14}$$

$$|\Delta_{12}^{(1)}| \le \gamma_b(|L||H|)_{12} \tag{9.15}$$

$$|\Delta^{(2)}| \le \gamma_b(|Y^T||L^T| + |W|) \tag{9.16}$$

$$|\Delta^{(3)}| \le \gamma_n|Y||L^T| \tag{9.17}$$

$$|\Delta_{22}^{(4)}| \le \gamma_{2b}(|L_{21}||W_{12}| + (|L_{21}||W_{12}|)^T) + \gamma_{2b+3(n-b)-1}|L_{22}||X_{22}||L_{22}|^T. \tag{9.18}$$

For example, Equation (9.14) is the result of applying the inductive hypothesis to the solution of a $b\times b$ system (the first step in the algorithm), and Equation (9.15) is the result of applying Lemma 2.16 to the second step of the algorithm, a $b \times b$ triangular solve. We omit the derivation of Equations (9.16)–(9.18) because they are similar.

Applying (9.16) and (9.17) to (9.11), substituting $W = YL^T + \Delta^{(3)}$, and then bounding again yields

$$|\Delta^{(5)}| \le \gamma_b|Y^T||L^T| + (\gamma_b + \gamma_n + \gamma_b\gamma_n)|Y||L^T|$$

and bounding $\gamma_b \le \gamma_{n+b}$ and $\gamma_b + \gamma_n + \gamma_b\gamma_n \le \gamma_{n+b}$ yields

$$|\Delta^{(5)}| \le \gamma_{n+b}|X||L^T|. \tag{9.19}$$

Substituting (9.15) and (9.19) into (9.12), further substituting $H = XL^T + \Delta^{(5)}$, and then applying (9.19) again yields

$$|\Delta_{12}| \le (\gamma_b + \gamma_{n+b} + \gamma_b\gamma_{n+b})(|L||X||L^T|)_{12} \le \gamma_{n+2b}(|L||X||L^T|)_{12}. \tag{9.20}$$

Next we bound Δ_{22} using (9.13), starting with the first two terms in that equation. We defined W such that $\Delta_{22}^{(3)}$ is zero and thus

$$(|L||\Delta^{(3)}| + (|L||\Delta^{(3)}|)^T)_{22} = |L_{21}||\Delta_{12}^{(3)}| + (|L_{21}||\Delta_{12}^{(3)}|)^T.$$

Further applying (9.17) yields

$$\begin{aligned}(|L||\Delta^{(3)}| + (|L||\Delta^{(3)}|)^T)_{22} &\le \gamma_n|L_{21}||Y_{1,:}||L_{2,:}|^T + \gamma_n(|L_{21}||Y_{1,:}||L_{2,:}|^T)^T \\ &= \gamma_n((|L||X||L^T|)_{22} - |L_{22}||X_{22}||L_{22}|^T). \qquad (9.21)\end{aligned}$$

Substituting $W = YL^T + \Delta^{(3)}$ in the first two terms of (9.18) and using the same argument that we used to produce (9.21) yields

$$|L_{21}||W_{12}| + (|L_{21}||W_{12}|)^T \le (1+\gamma_n)((|L||X||L^T|)_{22} - |L_{22}||X_{22}||L_{22}|^T). \qquad (9.22)$$

Substituting (9.22) into (9.18), and then substituting the result together with (9.21) into (9.13) yields

$$\begin{aligned}|\Delta_{22}| \le (\gamma_n + \gamma_{2b} + \gamma_{2b}\gamma_n)((|L||X||L^T|)_{22} - |L_{22}||X_{22}||L_{22}|^T) \\ + \gamma_{2b+3(n-b)-1}|L_{22}||X_{22}||L_{22}|^T.\end{aligned}$$

Because $1 \le b \le n-1$,

$$\begin{aligned}\gamma_n + \gamma_{2b} + \gamma_{2b}\gamma_n &\le \gamma_{n+2b} \le \gamma_{3n-2} \\ \gamma_{2b+3(n-b)-1} &= \gamma_{3n-b-1} \le \gamma_{3n-2},\end{aligned}$$

and thus

$$|\Delta_{22}| \le \gamma_{3n-2}(|L||X||L^T|)_{22}. \qquad (9.23)$$

Combining bounds (9.14), (9.20) and (9.23) we see that $|\Delta_{I,J}| \le C_{I,J}(|L||X||L^T|)_{I,J}$, where

$$C = \begin{bmatrix} \gamma_{3b-1} & \gamma_{n+2b} \\ \gamma_{n+2b} & \gamma_{3n-2} \end{bmatrix},$$

and because $C_{I,J} \le \gamma_{3n-2}$ for all I and J, we conclude that $|\Delta| \le \gamma_{3n-2}(|L||X||L^T|)$. The exception to this is the case $n = 1$, which requires the larger constant γ_{3n-1} in the statement of the lemma. □

9.2.2 Stability of the Block-Aasen Algorithm

We now show that the block-Aasen algorithm is backward stable. The analysis relies on lemmas from Section 2.6 and on Lemma 9.1. We use the symbols L, T, H and W to denote the corresponding floating point matrices and not their abstract exact equivalents. The exception to this is the diagonal blocks of W, which are not computed by the algorithm and which we define for convenience as being exactly

$$W_{J,J} = (RL^T)_{J,J} = R_{J,J}L^T_{J,J}.$$

Similarly, R is also not computed by the algorithm due to the optimization described in Section 9.1.4. We define R as the block upper-triangular matrix with symmetric diagonal blocks that satisfies $R^T + R = T$. Its superdiagonal blocks are exactly those of the computed T and its diagonal blocks are obtained from those of the computed T by scaling them by 0.5.

We break the main content of the proof into two lemmas, bounding the backward error in the off-diagonal blocks in Lemma 9.2 and bounding the backward error in the diagonal blocks in Lemma 9.3. Combining these results, we obtain the backward stability of the block-Aasen algorithm, stated in Theorem 9.4.

Lemma 9.2. *The computed factors satisfy $A = LTL^T + \Delta$, where*

$$|\Delta_{I,J}| \leq \gamma_{n+2b}(|L||T||L^T|)_{I,J}$$

whenever $I \neq J$.

Proof. Let the matrices $\Delta^{(1)}$ and $\Delta^{(2)}$ be such that

$$A = LH + \Delta^{(1)}, \qquad H = TL^T + \Delta^{(2)}.$$

Substituting the second expression into the first one yields

$$A = LTL^T + L\Delta^{(2)} + \Delta^{(1)},$$

and thus

$$\Delta = \Delta^{(1)} + L\Delta^{(2)}. \tag{9.24}$$

Bounding Δ requires that we obtain bounds on $\Delta^{(1)}$ and $\Delta^{(2)}$.

Let us bound the subdiagonal blocks of $\Delta^{(1)}$ by considering the computation that corresponds to Equation (9.4). In that equation we form the matrix $X = A_{J+1:N,J} - L_{J+1:N,1:J}H_{1:J,J}$ and then compute its LU factorization $X = L_{J+1:N,J+1}H_{J+1,J}$. Let $\Gamma^{(1)}$ and $\Gamma^{(2)}$ be such that

$$A_{J+1:N,J} = L_{J+1:N,1:J}H_{1:J,J} + X + \Gamma^{(1)} \tag{9.25}$$

$$X = L_{J+1:N,J+1}H_{J+1,J} + \Gamma^{(2)}. \tag{9.26}$$

Substituting the second expression into the first one yields

$$\begin{aligned} A_{J+1:N,J} &= L_{J+1:N,1:J}H_{1:J,J} + L_{J+1:N,J+1}H_{J+1,J} + \Gamma^{(1)} + \Gamma^{(2)} \\ &= L_{J+1:N1:J+1}H_{1:J+1,J} + \Gamma^{(1)} + \Gamma^{(2)}, \end{aligned}$$

because $H_{J+2:N,J}$ is zero,

$$A_{J+1:N,J} = (LH)_{J+1:N,J} + \Gamma^{(1)} + \Gamma^{(2)},$$

and therefore

$$\Delta^{(1)}_{J+1:N,J} = \Gamma^{(1)} + \Gamma^{(2)}. \tag{9.27}$$

We analyze the accuracy of forming X using Lemma 2.14, which yields the bound

$$|\Gamma^{(1)}| \leq \gamma_{Jb}(|L_{J+1:N,1:J}||H_{1:J,J}| + |X|).$$

However, because $L_{2:N,1}$ is zero, the inner dimension of the product $L_{J+1:N,1:J}H_{1:J,J}$ is effectively $(J-1)b$ instead of Jb, and therefore

$$|\Gamma^{(1)}| \leq \gamma_{(J-1)b}(|L_{J+1:N,1:J}||H_{1:J,J}| + |X|). \tag{9.28}$$

The accuracy of the LU factorization of X can be analyzed using Lemma 2.17, which yields

$$|\Gamma^{(2)}| \leq \gamma_b |L_{J+1:N,J+1}||H_{J+1,J}|. \tag{9.29}$$

Substituting (9.28) and (9.29) into (9.27) yields

$$|\Delta^{(1)}_{J+1:N,J}| \leq \gamma_{(J-1)b}(|L_{J+1:N,1:J}||H_{1:J,J}| + |X|) + \gamma_b |L_{J+1:N,J+1}||H_{J+1,J}|,$$

and further substituting (9.26) and using (9.29) again yields

$$|\Delta^{(1)}_{J+1:N,J}| \leq \gamma_{(J-1)b}|L_{J+1:N,1:J}||H_{1:J,J}| + (\gamma_{(J-1)b} + \gamma_b + \gamma_{(J-1)b}\gamma_b)|L_{J+1:N,J+1}||H_{J+1,J}|.$$

Bounding the constants in this expression according to

$$\begin{aligned} \gamma_{(J-1)b} &\leq \gamma_{Jb} \\ \gamma_{(J-1)b} + \gamma_b + \gamma_{(J-1)b}\gamma_b &\leq \gamma_{Jb}, \end{aligned}$$

where the second bound is justified by Lemma 2.12, yields

$$\begin{aligned} |\Delta^{(1)}_{J+1:N,J}| &\leq \gamma_{Jb}|L_{J+1:N,1:J}||H_{1:J,J}| + \gamma_{Jb}|L_{J+1:N,J+1}||H_{J+1,J}| \\ &= \gamma_{Jb}(|L||H|)_{J+1:N,J} \end{aligned}$$

and therefore

$$|\Delta^{(1)}_{I,J}| \leq \gamma_{Jb}(|L||H|)_{I,J} \tag{9.30}$$

for all $I > J$.

We bound the diagonal and superdiagonal blocks of $\Delta^{(2)}$ by considering the computation that corresponds to Equation (9.3). In that equation we compute blocks of H by forming the corresponding blocks of TL^T, and as we discuss in Section 9.1.4, these blocks are formed according to the formula

$$H_{I,J} = T_{I,I-1}(L_{J,I-1})^T + T_{I,I}(L_{J,I})^T + T_{I,I+1}(L_{J,I+1})^T.$$

This is equivalent to multiplying the $b \times 3b$ matrix $T_{I,I-1:I+1}$ by the $3b \times b$ matrix $(L_{J,I-1:I+1})^T$, and the accuracy of this computation is bounded in Lemma 2.13, which guarantees that

$$|\Delta^{(2)}_{I,J}| \leq \gamma_{3b}(|T||L^T|)_{I,J}$$

for all $I \leq J$. The blocks $\Delta^{(2)}_{J+1,J}$ correspond to Equation (9.5), which solves the triangular system $H_{J+1,J} = T_{J+1,J}(L_{J,J})^T$. This is analyzed in Lemma 2.16, which guarantees that

$$|\Delta^{(2)}_{J+1,J}| \leq \gamma_b(|T||L^T|)_{J+1,J}.$$

All other blocks of $\Delta^{(2)}$ are zero, and thus

$$|\Delta^{(2)}| \leq \gamma_{3b}|T||L^T|. \tag{9.31}$$

Substituting (9.30) and (9.31) into (9.24) yields

$$|\Delta_{I,J}| \leq \gamma_{Jb}(|L||H|)_{I,J} + \gamma_{3b}(|L||T||L^T|)_{I,J}$$

for all $I > J$. Further substituting $H = TL^T + \Delta^{(2)}$ and using (9.31) once again yields

$$|\Delta_{I,J}| \leq (\gamma_{Jb} + \gamma_{3b} + \gamma_{Jb}\gamma_{3b})(|L||T||L^T|)_{I,J} \leq \gamma_{Jb+3b}(|L||T||L^T|)_{I,J}.$$

The constant γ_{Jb+3b} is maximized when $J = N - 1$, which yields the required bound for all $I > J$. As for $I < J$, the same bound holds because Δ is the difference of the two symmetric matrices A and LTL^T and is thus itself symmetric. □

Lemma 9.3. *The computed factors satisfy $A = LTL^T + \Delta$, where*

$$|\Delta_{J,J}| \leq \gamma_{2n-b-1}(|L||T||L^T|)_{J,J}$$

for $J = 1, 2, \ldots, N$.

Proof. Let the matrices $\Delta^{(1)}$ and $\Delta^{(2)}$ be such that

$$A = LW + (LW)^T + \Delta^{(1)}, \qquad W = RL^T + \Delta^{(2)}.$$

Substituting the second expression into the first one yields

$$\begin{aligned} A &= LRL^T + (LRL^T)^T + L\Delta^{(2)} + (L\Delta^{(2)})^T + \Delta^{(1)} \\ &= L(R + R^T)L^T + L\Delta^{(2)} + (L\Delta^{(2)})^T + \Delta^{(1)} \\ &= LTL^T + L\Delta^{(2)} + (L\Delta^{(2)})^T + \Delta^{(1)} \end{aligned}$$

and therefore

$$\Delta = \Delta^{(1)} + L\Delta^{(2)} + (L\Delta^{(2)})^T. \tag{9.32}$$

Equation (9.2) computes $T_{J,J}$ by forming $X = A_{J,J} - L_{J,1:J-1}W_{1:J-1,J} - (L_{J,1:J-1}W_{1:J-1,J})^T$ and then solving $L_{J,J}T_{J,J}(L_{J,J})^T = X$. Let $\Gamma^{(1)}$ and $\Gamma^{(2)}$ be such that

$$A_{J,J} = L_{J,1:J-1}W_{1:J-1,J} + (L_{J,1:J-1}W_{1:J-1,J})^T + X + \Gamma^{(1)} \tag{9.33}$$

$$X = L_{J,J}T_{J,J}(L_{J,J})^T + \Gamma^{(2)}. \tag{9.34}$$

Substituting (9.34) into (9.33) yields

$$A_{J,J} = L_{J,1:J-1}W_{1:J-1,J} + (L_{J,1:J-1}W_{1:J-1,J})^T + L_{J,J}T_{J,J}(L_{J,J})^T + \Gamma^{(1)} + \Gamma^{(2)}.$$

Rewriting the term $L_{J,J}T_{J,J}(L_{J,J})^T$ according to

$$\begin{aligned} L_{J,J}T_{J,J}(L_{J,J})^T &= L_{J,J}(R_{J,J} + (R_{J,J})^T)(L_{J,J})^T \\ &= L_{J,J}R_{J,J}(L_{J,J})^T + (L_{J,J}R_{J,J}(L_{J,J})^T)^T \\ &= L_{J,J}W_{J,J} + (L_{J,J}W_{J,J})^T \end{aligned}$$

gives

$$\begin{aligned} A_{J,J} &= L_{J,1:J-1}W_{1:J-1,J} + (L_{J,1:J-1}W_{1:J-1,J})^T + L_{J,J}W_{J,J} + (L_{J,J}W_{J,J})^T + \Gamma^{(1)} + \Gamma^{(2)} \\ &= L_{J,1:J}W_{1:J,J} + (L_{J,1:J}W_{1:J,J})^T + \Gamma^{(1)} + \Gamma^{(2)} \\ &= (LW + (LW)^T)_{J,J} + \Gamma^{(1)} + \Gamma^{(2)} \end{aligned}$$

and thus

$$\Delta^{(1)}_{J,J} = \Gamma^{(1)} + \Gamma^{(2)}. \tag{9.35}$$

The accuracy of forming X and then solving for $T_{J,J}$ can be bounded using Lemmas 2.15 and 9.1, which guarantee that

$$\begin{aligned} |\Gamma^{(1)}| &\le \gamma_{2(J-2)b}(|L_{J,1:J-1}||W_{1:J-1,J}| + (|L_{J,1:J-1}||W_{1:J-1,J}|)^T + |X|) \\ |\Gamma^{(2)}| &\le \gamma_{3b-1}|L_{J,J}||T_{J,J}||L_{J,J}|^T. \end{aligned}$$

Substituting these bounds into (9.35), further substituting (9.34) and bounding again yields

$$\begin{aligned} |\Delta^{(1)}_{J,J}| &\le \gamma_{2(J-2)b}(|L_{J,1:J-1}||W_{1:J-1,J}| + (|L_{J,1:J-1}||W_{1:J-1,J}|)^T) \\ &\qquad + (\gamma_{2(J-2)b} + \gamma_{3b-1} + \gamma_{2(J-2)b}\gamma_{3b-1})|L_{J,J}||T_{J,J}||L_{J,J}|^T \\ &\le \gamma_{2(J-2)b}(|L_{J,1:J-1}||W_{1:J-1,J}| + (|L_{J,1:J-1}||W_{1:J-1,J}|)^T) \\ &\qquad + \gamma_{2(J-2)b+3b-1}|L_{J,J}||T_{J,J}||L_{J,J}|^T, \end{aligned} \tag{9.36}$$

which is the bound we require for $\Delta^{(1)}$.

The superdiagonal blocks of $\Delta^{(2)}$ correspond to Equation (9.1). That equation states that blocks of W are computed by forming the corresponding blocks of the product RL^T, which are formed according to the formula

$$W_{I,J} = 0.5(T_{I,I}(L_{J,I})^T) + T_{I,I+1}(L_{J,I+1})^T,$$

as we explain in Section 9.1.4. Because of the scaling by 0.5 we cannot apply Lemma 2.13 directly to this formula. Instead we must bound the errors resulting from forming the two single-block products separately, and then use assumptions (2.4) and (2.3) to account for the effects of scaling and summation respectively. We skip the details; the resulting bound is

$$|\Delta^{(2)}_{I,J}| \le \gamma_{b+1}(|R||L^T|)_{I,J} \tag{9.37}$$

for all $I < J$.

Next we return to bounding (9.32), starting with the last two terms. The diagonal and subdiagonal blocks of W are defined such that the corresponding blocks of $\Delta^{(2)}$ are zero, and therefore

$$(|L||\Delta^{(2)}| + (|L||\Delta^{(2)}|))_{J,J} = |L_{J,1:J-1}||\Delta^{(2)}_{1:J-1,J}| + (|L_{J,1:J-1}||\Delta^{(2)}_{1:J-1,J}|)^T.$$

Substituting (9.37) yields

$$\begin{aligned}(|L||\Delta^{(2)}| + (|L||\Delta^{(2)}|))_{J,J} \le \gamma_{b+1}(&|L_{J,1:J-1}||R_{1:J-1,1:J}||L_{J,1:J}|^T \\ &+ (|L_{J,1:J-1}||R_{1:J-1,1:J}||L_{J,1:J}|^T)^T),\end{aligned}$$

which can be further simplified according to

$$\begin{aligned}&|L_{J,1:J-1}||R_{1:J-1,1:J}||L_{J,1:J}|^T + (|L_{J,1:J-1}||R_{1:J-1,1:J}||L_{J,1:J}|^T)^T \\ &\quad = 2\sum_{I=1}^{J-1} |L_{J,I}||R_{I,I}||L_{J,I}|^T + \sum_{I=1}^{J-1} |L_{J,I}||R_{I,I+1}||L_{J,I+1}|^T + \sum_{I=1}^{J-1} |L_{J,I+1}||R_{I,I+1}|^T|L_{J,I}|^T \\ &\quad = \sum_{I=1}^{J-1} |L_{J,I}||T_{I,I}||L_{J,I}|^T + \sum_{I=1}^{J-1} |L_{J,I}||T_{I,I+1}||L_{J,I+1}|^T + \sum_{I=1}^{J-1} |L_{J,I+1}||T_{I+1,I}||L_{J,I}|^T \\ &\quad = (|L||T||L|^T)_{J,J} - |L_{J,J}||T_{J,J}||L_{J,J}|^T,\end{aligned}$$

yielding

$$(|L||\Delta^{(2)}| + (|L||\Delta^{(2)}|)^T)_{J,J} \le \gamma_{b+1}((|L||T||L^T|)_{J,J} - |L_{J,J}||T_{J,J}||L_{J,J}|^T). \tag{9.38}$$

To bound the first term in (9.36) we substitute $W = RL^T + \Delta^{(2)}$ and apply the same arguments we used to produce (9.38), obtaining

$$\begin{aligned}|L_{J,1:J-1}||W_{1:J-1,J}| + (|L_{J,1:J-1}||W_{1:J-1,J}|)^T \le (1+\gamma_{b+1})((&|L||T||L^T|)_{J,J} \\ &- |L_{J,J}||T_{J,J}||L_{J,J}|^T).\end{aligned} \tag{9.39}$$

Finally, substituting (9.39) into (9.36) and then substituting the result together with (9.38) into (9.32) yields

$$\begin{aligned}|\Delta_{J,J}| \leq (\gamma_{2(J-2)b} + \gamma_{b+1} + \gamma_{2(J-2)b}\gamma_{b+1})((|L||T||L^T|)_{J,J} - |L_{J,J}||T_{J,J}||L_{J,J}|^T) \\ + \gamma_{2(J-2)b+3b-1}|L_{J,J}||T_{J,J}||L_{J,J}|^T.\end{aligned}$$

Because $b \geq 1$, we can bound

$$\gamma_{2(J-2)b} + \gamma_{b+1} + \gamma_{2(J-2)b}\gamma_{b+1} \leq \gamma_{2(J-2)b+b+1} \leq \gamma_{2(J-2)b+3b-1},$$

which allows us to cancel the two instances of $|L_{J,J}||T_{J,J}||L_{J,J}|^T$ and obtain

$$|\Delta_{J,J}| \leq \gamma_{2(J-2)b+3b-1}(|L||T||L^T|)_{J,J}.$$

The constant $\gamma_{2(J-2)b+3b-1}$ is maximized when $J = N$, which yields the required bound. □

With these lemmas, we can now state the backward stability of the overall algorithm.

Theorem 9.4. *The computed factors satisfy $A = LTL^T + \Delta$, where*

$$|\Delta| \leq \gamma_{2n-b-1}|L||T||L^T|$$

if $n > 3b$ and

$$|\Delta| \leq \gamma_{n+2b}|L||T||L^T|$$

otherwise.

Proof. Lemmas 9.2 and 9.3 state that

$$|\Delta_{I,J}| \leq \gamma_{n+2b}(|L||T||L^T|)_{I,J}$$

whenever $I \neq J$, and

$$|\Delta_{I,J}| \leq \gamma_{2n-b-1}(|L||T||L^T|)_{I,J}$$

whenever $I = J$, and therefore the bound

$$|\Delta_{I,J}| \leq \max\{\gamma_{n+2b}, \gamma_{2n-b-1}\}(|L||T||L^T|)_{I,J}$$

holds for all I and J. The quantity γ_n increases monotonically with n (so long as $nu < 1$) and therefore $\gamma_{2n-b-1} \geq \gamma_{n+2b}$ whenever $2n - b - 1 \geq n + 2b$, which occurs whenever $n > 3b$. □

9.2.3 Growth

The stability of the factorization algorithm depends on the magnitude of L and T relative to that of A. How large can L and T get? In the element-wise Aasen algorithm, the magnitude of elements of L is bounded by 1 (because the algorithm uses partial pivoting), and it is easy to show that $|T_{ij}| \leq 4^{n-2} \max_{ij} |A_{ij}|$ [86]; the argument is essentially the same as the one that establishes the bound on $|U_{ij}|$ in Gaussian elimination with partial pivoting. Furthermore, this bound is attained by a known matrix of order $n = 3$, although larger matrices that attain the bound are not known [85, p. 224].

It is important to interpret this bound correctly. The actual expression 4^{n-2} is not important, because it does not indicate the growth that is normally attained. The same is true for LU with partial pivoting; it is stable *in spite* of the fact that the growth factor bound is as large as 2^{n-1}, not because this bound is small (it is not; it is huge). Two other things are important. One is that the bound shows that growth is not related to the condition number of A. The second is that growth in practice is small. The reasons for this are complex and not completely understood even in LU with partial pivoting, but this is the reality; for deeper analyses and discussion, see [128, 144] and [85, Section 9.4].

If we compute the factorization in Equation (9.4) using LU with partial pivoting (GEPP), essentially the same bounds hold for our block algorithm. The block columns of the L factor are generated by Gaussian elimination with partial pivoting, so the same two-sided doubling-up argument shows that the growth factor for T is bounded by 4^{n-b-1} (since the first columns of L are unit vectors and additions/subtractions start only in column $b+1$). We provide numerical experiments in Section 9.4 to illustrate the backward stability and growth using GEPP within the block-Aasen algorithm in practice.

When the factorization in Equation (9.4) is computed in a communication-avoiding way using the tall-skinny LU factorization [80] (TSLU), L is still bounded, but the bound is 2^{bh}, where h is a parameter of TSLU that normally satisfies $h = O(\log n)$. This can obviously be much larger than 1, although experiments indicate that L is usually much smaller. This implies that growth in T is still bounded, but the bound is now 4^{nbh}. This is worse than with GEPP, but as we explained above, this theoretical bound is not what normally governs the stability of the algorithm. We leave numerical experiments with TSLU in the block-Aasen algorithm to future work. Also, the recently-developed panel rank-revealing LU factorization [100] may improve the growth bounds in our algorithm, but we have not fully explored this.

9.3 Sequential Complexity Analyses

In this section we analyze the costs of the sequential algorithm. We begin with an analysis of the computational complexity and then analyze the communication costs of the algorithm in the sequential model.

9.3.1 Computational Cost

Our goal in this subsection is to show that the new blocked algorithm performs the same number of arithmetic operations as the element-wise one, up to lower order terms.

In order to determine the arithmetic complexity of the algorithm, we consider only Equations (9.1)–(9.5) (the computational cost of pivoting is negligible). Letting J denote the index of the outermost loop of the algorithm and b be the block size, the arithmetic cost of Equations (9.1)–(9.3) is $O(Jb^3)$ flops. This is because each equation involves $O(J)$ block multiplications of $b \times b$ blocks (some of which are triangular). Note that in Equation (9.2), the dominant cost is in computing the product of the block row of L with the block column of W; the arithmetic cost of the two-sided symmetric solve is $O(b^3)$. Similarly, Equation (9.5) is a triangular solve involving one block and has an arithmetic cost of $O(b^3)$. The dominant arithmetic cost for the block-Aasen algorithm comes from Equation (9.4), which consists of two subcomputations: a matrix multiplication involving L and H and an LU decomposition of a block column. The arithmetic cost of the LU decomposition is $O(Jb^3)$. The matrix multiplication step multiplies an $(N-J)b \times Jb$ submatrix of L by a $Jb \times b$ submatrix of H. At the Jth step of the algorithm, this arithmetic cost is $2(N-J)Jb^3$ flops, ignoring lower order terms. Summing over the outermost loop and using the fact that $N = n/b$, we have a total arithmetic cost of

$$\sum_{J=2}^{N} \left(2(N-J)Jb^3 + O(Jb^3)\right) = \frac{1}{3}n^3 + o(n^3).$$

9.3.2 Communication Costs

To determine the communication complexity of the algorithm, we must consider Equations (9.1)–(9.5) as well as the cost of applying symmetric permutations to the trailing matrix. We analyze the three parts of the algorithm separately: block operations (all of the computations described in Equations (9.1)–(9.5) with the exception of the LU decomposition), LU decomposition of block columns, and application of the permutations. We assume the matrix is stored in block-contiguous format with block size b, the same as the algorithmic block size. In block contiguous format, $b \times b$ blocks are stored contiguously in memory (see Section 2.3.1. We assume column-major ordering of elements within blocks and of the blocks themselves. In the following analysis, we assume $b \leq \sqrt{M/3}$ so that three blocks fit simultaneously in fast memory.

9.3.2.1 Block Operations

By excluding the LU decomposition, all the other computations in Equations (9.1)–(9.5) involve block operations–either block multiplication (sometimes involving triangular or symmetric matrices), block triangular solve, or block two-sided symmetric triangular solve. For

Algorithm	Words	Messages
RLU	$O\left(\frac{nb^2}{\sqrt{M}} + nb\log b\right)$	$O\left(\min\left(n, \frac{n^2}{M}\right)\right)$
SMLU [32]	$O\left(\frac{nb^2}{\sqrt{M}} + nb\log b \log\frac{nb}{M}\right)$	$O\left(\frac{nb^2}{M^{3/2}} + \frac{nb}{M}\log b\log\frac{nb}{M}\right)$
TSLU [80]	$O\left(\frac{nb^2}{\sqrt{M}}\right)$	$O\left(\frac{nb^2}{M^{3/2}}\right)$

Table 9.1: Communication costs of LU decomposition algorithms applied to an $n \times b$ matrix stored in $b \times b$ block contiguous storage, assuming $b \leq \sqrt{M/3}$.

example, in Equation (9.3), we compute

$$\underline{H_{I,J}} = T_{I,I-1}(L_{J,I-1})^T + T_{I,I}(L_{J,I})^T + (T_{I+1,,I})^T(L_{I+1,J})^T$$

(assuming only the lower halves of T and L are stored). Each of the three multiplications involve $b \times b$ blocks, so by the assumption that $b \leq \sqrt{M/3}$, the operations can be performed by reading contiguous input blocks of size b^2 words into fast memory, performing $O(b^3)$ floating point operations, and then writing the output block back to slow memory. This implies that the number of messages is proportional to the number of block operations, which is $O((\text{computational cost})/b^3) = O(n^3/b^3)$ and the number of words moved is $O((\text{computational cost})/b) = O(n^3/b)$.

9.3.2.2 Panel Decomposition

We now consider algorithms for the LU decomposition of the column panel. Note that the $O(N)$ LU factorizations, each involving $O(Nb^3)$ flops, contribute altogether only an $O(n^2b)$ term to the computational complexity of the overall block-Aasen algorithm, a lower order term. Thus, attaining the communication lower bound for the overall algorithm does not require attaining optimal data re-use within panel factorizations. For example, using a naive algorithm and achieving only constant re-use of data during the LU factorization translates to a total of $O(n^2b)$ words moved during LU factorizations, which is dominated by the communication complexity of the block operations, $O(n^3/b)$ words, in the case where $n \gg b^2$. However, to ensure that both bandwidth and latency costs of the LU factorizations do not asymptotically exceed the costs of the rest of the block-Aasen algorithm for all matrix dimensions, we need algorithms that achieve better than constant re-use (though the algorithms need not be asymptotically optimal).

We choose to use the recursive algorithm (RLU) of [83, 142] for panel factorizations, updated slightly to match the block-contiguous data layout. The algorithm works by splitting the matrix into left and right halves, factoring the left half recursively, updating the right half, and then factoring the trailing matrix in the right half recursively. In order to match the block-contiguous layout, the update of the right half (consisting of a triangular solve and matrix multiplication) should be performed block by block. The bandwidth cost of this algorithm for $n \times b$ matrices is analyzed in [142], and the latency cost can be bounded by

Algorithm	Words	Messages
Direct	$O(nb)$	$O(nb)$
Blocked	$O(n^2)$	$O\left(\frac{n^2}{b^2}\right)$

Table 9.2: Communication costs of symmetric pivoting schemes

the recurrence $L(n, b) \leq 2L(n, b/2) + O(N)$. The $O(N)$ term comes from the update of the right half of the matrix, which involves reading contiguous chunks of each of the $O(N)$ blocks in the panel. The base case occurs either when the subpanel fits in memory ($nb < M$) or when $b = 1$. The cost of the recursive algorithm is dominated by its leaves, each of which requires $O(N)$ messages. Depending on the relative sizes of n and M, there are either nb/M or b leaves starting with an $n \times b$ matrix. The latency cost becomes the minimum of two terms: $O(n)$ or $O(n^2/M)$. The bandwidth and latency costs are summarized in the first row of Table 9.1.

In order to determine the contribution of LU factorizations to the costs of the overall block-Aasen algorithm, we must multiply the cost of the $n \times b$ factorization by N, the number of panel factorizations. Using the RLU algorithm, this yields a bandwidth cost of $O(n^2 b/\sqrt{M} + n^2 \log b)$ words and a latency cost of $O(\min(n^2/b, n^3/(bM))$ messages. With the exception of the $O(n^2 \log b)$ term in the bandwidth cost, these costs are always asymptotically dominated by the costs of the block operations.

While the RLU algorithm is sufficient for minimizing communication in the block-Aasen algorithm, there are algorithms which require fewer messages communicated. The Shape-Morphing LU algorithm (SMLU) [32] is an adaptation of RLU that changes the matrix layout on the fly to reduce latency cost. The algorithm and its analysis are provided in [32], and the communication costs are given in the second row of Table 9.1. SMLU uses partial pivoting and incurs a slight bandwidth cost overhead compared to RLU (an extra logarithmic factor on one term). Another algorithm which reduces latency cost even further is the communication-avoiding tall-skinny LU algorithm (TSLU) [80], as described in Section 7.1.4. The algorithm can be applied to general matrices, but the main innovation focuses on tall-skinny matrices. TSLU uses tournament pivoting, a different scheme than partial pivoting, which has slightly weaker theoretical numerical stability properties. The algorithm and analysis are provided in [80], and the communication costs are given in the third row of Table 9.1. The communication costs of TSLU are optimal with respect to each panel factorization.

9.3.2.3 Applying Symmetric Permutations

After each LU decomposition of a block column, we apply the internal permutation to the rest of the matrix. This permutation involves back-pivoting, or swapping rows of the already factored L matrix, and forward-pivoting of the trailing symmetric matrix. Applying the symmetric permutations to the trailing matrix includes swapping elements within a given set of rows and columns, as shown in Figure 9.12. For example, applying the transposition

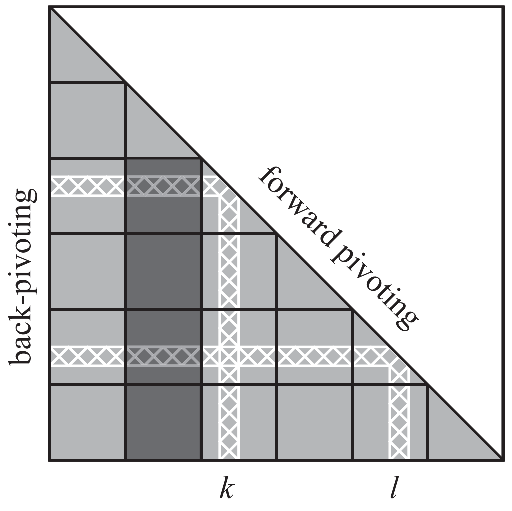


Figure 9.12: Exchanging rows and columns k and l. The second (dark) block column is the block column of the reduced matrix whose LU factorization was just computed. The first block column is a block column of L; the algorithm applies back pivoting to it. Block columns 3 to 6 are part of the trailing submatrix; the algorithm applies forward pivoting to them. The trailing submatrix is square and symmetric, but only its lower triangle is stored, so a row that needs to be exchanged is represented as a partial row (up to the diagonal) and a partial column, as shown here.

(k, l) implies that the L-shaped set of elements in the kth row and kth column (to the left and below the diagonal) is swapped with the L-shaped set of elements in the lth row and lth column, such that element a_{kk} is swapped with element a_{ll} and element a_{lk} stays in place (see Figure 9.12).

Since there are at most b swaps that must be performed for a given LU decomposition, and each swap consists of $O(n)$ data, the direct approach of swapping L-shaped sets of elements one at a time has a bandwidth cost of $O(nb)$ words. However, no matter how individual elements within blocks are stored, because the permutations involve accessing both rows and columns, at least half of the elements will be accessed non-contiguously, so the latency cost of the direct approach is also $O(nb)$ messages. Since there are $N = n/b$ symmetric permutations to be applied, these costs amount to a total of $O(n^2)$ words and $O(n^2)$ messages. While the bandwidth cost is a lower order term with respect to the rest of the algorithm, the latency cost of the permutations exceeds the rest of the algorithm, except when $n \gg M^{3/2}$. This approach is the symmetric analogue of Variant 1 in [80].

In order to reduce the latency cost, we use a blocked approach which will require greater bandwidth cost than the direct approach but will not increase the asymptotic bandwidth cost of the block-Aasen algorithm. The blocked approach accesses contiguous $b \times b$ blocks, but it may permute only a few rows or columns of the blocks. This approach is the symmetric analogue of Variant 2 in [80].

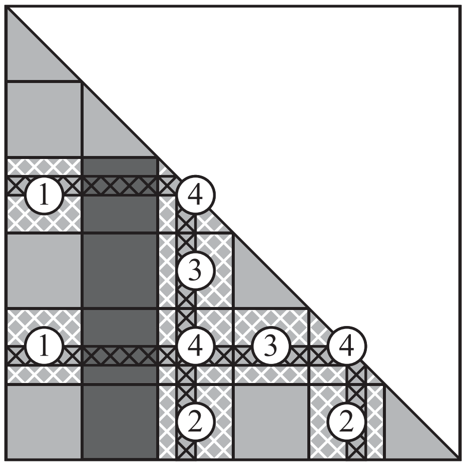


Figure 9.13: Exchanging a pair of rows in the blocked approach. Numbers indicate sets of blocks that are held simultaneously in fast memory: all the blocks marked "1" are held in fast memory simultaneously, later all the blocks marked "2", and so on.

The algorithm works as follows: for each block in the LU factorization panel that includes a permuted row, we update the N pairs of blocks shown in Figure 9.13. The updates include back-pivoting (updating parts of the L matrix that have already been computed) and forward-pivoting (updating the trailing matrix). Nearly all the updates involve pairs of blocks, which fit in fast memory simultaneously. Pairs of blocks involved in back-pivoting are not affected by column permutations and swap only rows (those marked "1" in Figure 9.13). Some pairs of blocks involved in forward-pivoting are not affected by row permutations and swap only columns (those marked "2" in Figure 9.13). Because only half of the matrix is stored, some pairs of blocks in the trailing matrix will swap columns for rows (those marked "3" in Figure 9.13). The more complicated updates involve blocks that are affected by both row and column permutations: the two diagonal blocks and the corresponding off-diagonal block, marked "4" in Figure 9.13. In order to apply the two-sided permutation to these blocks, all three blocks are read into fast memory and updated at once. Since there are $O(N)$ blocks in each LU factorization panel, and each block with a permuted row requires accessing $O(N)$ blocks to apply the symmetric permutation, for a given LU factorization, the number of words moved in applying the associated permutation is $O(N^2b^2) = O(n^2)$, and the total number of messages moved is $O(N^2) = O(n^2/b^2)$. The communication costs of the two approaches are summarized in Table 9.2.

9.3.2.4 Communication Optimality of the Block-Aasen Algorithm

Combining the analysis of the three sections above, we obtain the communication costs of the overall algorithm. Assuming block-contiguous layout, the communication costs of the block operations are $O(n^3/b)$ words and $O(n^3/b^3)$ messages, the costs of the panel factorizations using the RLU algorithm are $O(n^2b/\sqrt{M}+n^2\log b)$ words and $O(\min\{n^2/b, n^3/(bM)\})$

messages, and the costs of the pivoting using the blocked approach are $O(n^3/b)$ words and $O(n^3/b^3)$ messages. Choosing a block size of $b = \Theta(\sqrt{M})$, we obtain communication costs of the block-Aasen algorithm of

$$W = O\left(\frac{n^3}{\sqrt{M}} + n^2 \log M\right)$$

and

$$S = O\left(\frac{n^3}{M^{3/2}}\right).$$

Given the communication lower bound for this factorization (Corollary 4.14), this algorithm is communication optimal except in the tiny range $M \ll n^2 \ll M \log^2 M$. If TSLU is used for the panel factorization, the algorithm is optimal for all n.

9.4 Numerical Experiments

Next we describe a set of numerical experiments that provide further insight into the numerical behavior of the algorithm. We used a block size $b = 16$ in all the experiments.

We carried out two sets of experiments: one set involving random matrices and another involving matrices from the University of Florida Sparse Matrix Collection [55]. In the first set of experiments we generated a sequence of random square symmetric matrices of order n for 100 distinct values of n, linearly spaced in the interval $100 \leq n \leq 5{,}000$. The elements of these matrices are distributed normally and independently (preserving symmetry, of course) with mean 0 and standard deviation 1. In all of our experiments we used GEPP for panel factorizations. We leave experiments using the tall-skinny *LU* (TSLU) algorithm to future work. For each matrix we measured three parameters: the growth factor, the backward error of the factorization, and the backward error in the solution of a linear system of equations $Ax = b$, where b is the sum of the columns of A (so x is the vector of all ones).

We define the growth factor as the number

$$\frac{\left\| |L|\,|T|\,|L|^T \right\|_\infty}{\|A\|_\infty},$$

a definition that is justified by Theorem 9.4. The factorization error is defined as

$$\max_{i,j} \frac{\left|PAP^T - LTL^T\right|_{i,j}}{\left(|L|\,|T|\,|L|^T\right)_{i,j}},$$

using the convention $0/0 = 0$. We compute the backward error of the floating point solution $\hat{x}$ to the system $Ax = b$ according to

$$\frac{\|A\hat{x} - b\|_\infty}{\|A\|_\infty \|\hat{x}\|_\infty + \|b\|_\infty},$$

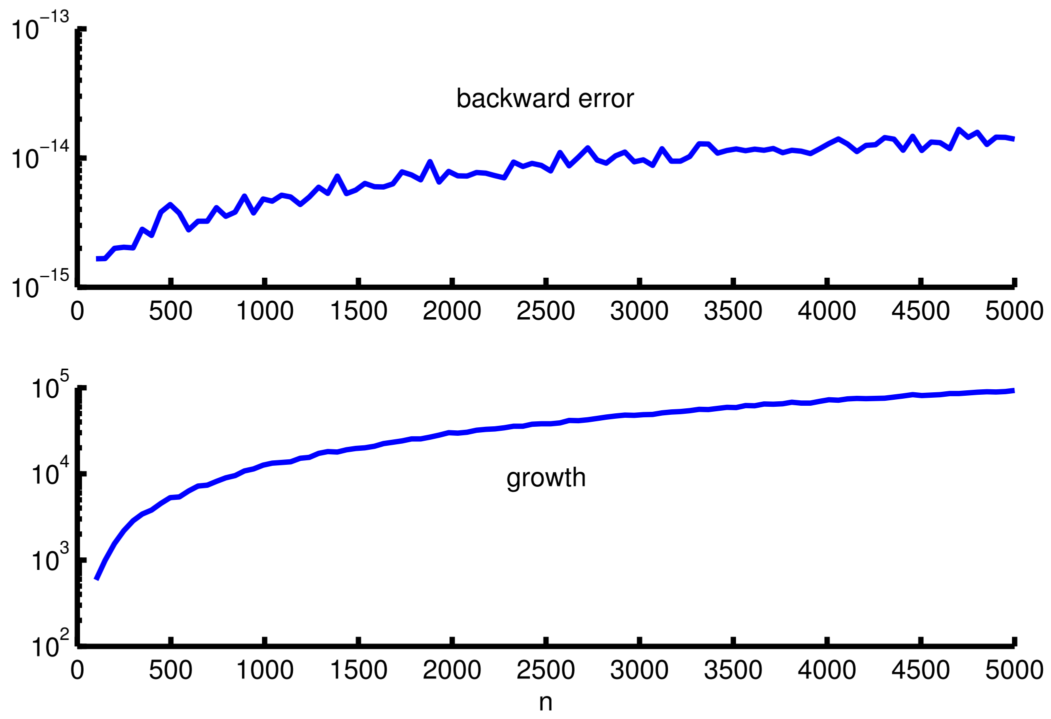


Figure 9.14: Backward errors in the solution of $Ax = b$ and the growth factors in the factorization of random matrices. The matrices are ordered by growth.

	n	κ	$\mathtt{nnz}/n$
minimum	64	5.0×10^0	0.46
1st quartile	800	8.4×10^3	6.81
median	2,000	2.5×10^6	12.94
3rd quartile	4,581	3.2×10^{10}	23.93
maximum	8,140	`inf`	378.19

Table 9.3: Statistics of the University of Florida matrix set.

where we use an unsymmetric LU algorithm to solve the band system.

The factorizations of random matrices were completely backward stable, with backward errors between $1.1u$ and $2.4u$, with a median of $1.9u$. The stability of solutions to linear systems and the growth factors are shown in Figure 9.14. The backward errors are moderate, varying between 1.7×10^{-15} and 1.7×10^{-14}. The backward error is increasing with n but at a rate that is clearly slower than linear. The growth factor is strongly correlated with the error, which is consistent with the bound in Theorem 9.4. The error does not seem to depend on n outside of the implicit dependence through the growth factor, in contrast with the bound in Theorem 9.4.

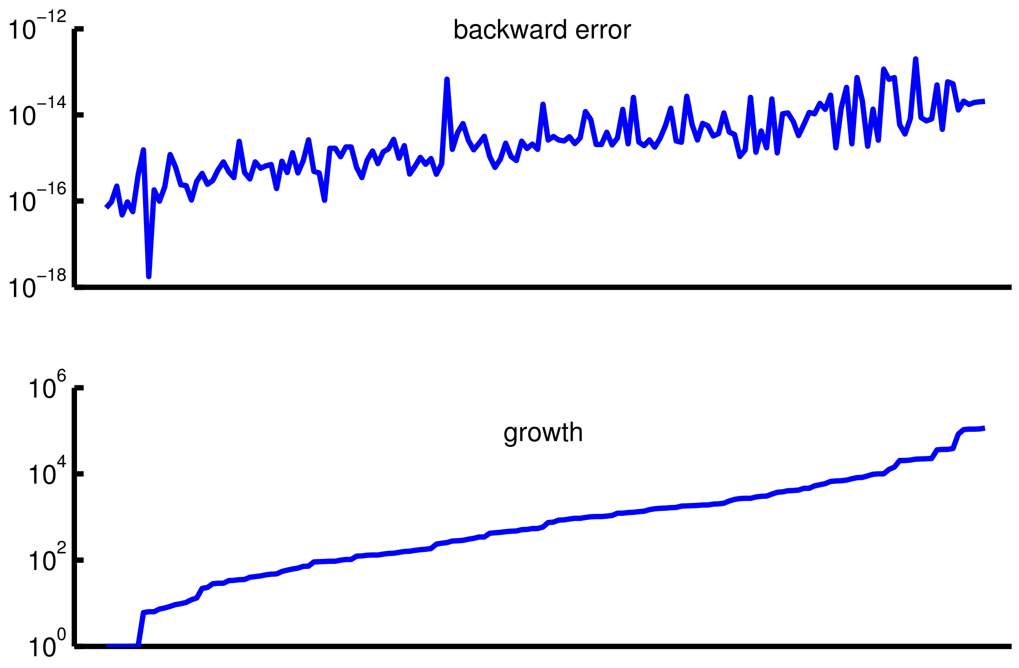


Figure 9.15: Backward errors and growth factors on matrices from the University of Florida Collection. The matrices are ordered by growth.

The second set of experiments factored 180 matrices from the University of Florida Sparse Matrix Collection. We chose for this experiment *all* of the symmetric, real, non-binary matrices of order $64 \le n \le 8{,}192$, with the exception of matrices with bandwidth b or less. Matrices with low bandwidth were omitted because they are their own T factors and therefore do not require factorization. This set of 180 matrices is further described in Table 9.3. The experiment was conducted according to the same scheme as the experiment involving the random matrices.

The algorithm experienced difficulties on 14 of the matrices; they are discussed later in this section.

On the 166 matrices on which the algorithm produced good results, we obtained stable factorizations with backward errors of less than $11u$. The stability of the linear solver and the growth are shown in Figure 9.15. The linear-solver backward errors are in the interval $[1.8 \times 10^{-18}, 2.0 \times 10^{-13}]$ with a median of 2.1×10^{-15}.

On 14 matrices, the linear solver that we used to solve banded systems involving T failed to produce a solution. For solving such systems we use the LAPACK subroutine `GBSV`, which is a banded implementation of GEPP. The source of the problem is that when T is rank deficient, `GBSV` produces a U factor with zeros on the diagonal, and this factor cannot be used to solve linear systems. In all 14 matrices the root cause was structural or numerical rank deficiency of A (Matlab reported condition numbers larger than 10^{20}). Our factorization algorithm produced stable factorizations, with backward errors of order u, well conditioned L's, and mild growth (up to 1.2×10^6).

9.5 Conclusions

We have shown that a block variant of Aasen's factorization algorithm can reduce a symmetric matrix into a symmetric banded form in a communication-avoiding way. No prior symmetric reduction algorithm achieves similar efficiency bounds. We show in [15] that the shared-memory parallel algorithm performs well in practice on a multi-core machine; here we focused on complete analyses of the sequential algorithm's communication costs, arithmetic costs, and numerical stability.

Chapter 10

Communication-Avoiding Successive Band Reduction

In this chapter, we present new sequential and parallel algorithms for tridiagonalizing a symmetric band matrix in order to compute its eigendecomposition. In order to preserve band structure, band reduction algorithms based on orthogonal similarity transformations proceed by an annihilate-and-chase approach. Annihilating entries within the band creates fill-in (bulges); to preserve sparsity, these bulges are chased off the band before annihilating subsequent entries. A general framework for this procedure, known as successive band reduction (SBR), appears in [38].

The main contributions of this chapter are the following:

- we describe new techniques for avoiding communication in the context of SBR,
- we present both new sequential algorithms and improvements on existing ones that asymptotically reduce both bandwidth and latency costs,
- we introduce a new parallel algorithm that requires asymptotically fewer messages than previous approaches, and
- we describe how the new sequential and parallel band reduction algorithms can be used in the context of the dense problem to attain the corresponding communication lower bounds.

Although no communication lower bound is known for the band reduction problem in isolation, we demonstrate that previous approaches communicate asymptotically more than necessary. Our results also prove that the assumption of forward progress (Definition 4.18) is a necessary condition for the lower bound of Theorem 4.25. That is, our algorithms break the assumption and beat the lower bound, attaining asymptotically better data re-use than the lower bound allows (see Section 10.1.4 for more details).

While the symmetric band eigenproblem is interesting in its own right, this work is motivated by the high communication costs of the standard algorithms for solving the full (dense) symmetric problem via tridiagonalization. Greater efficiency than the standard approach can be obtained if the tridiagonalization procedure is split into two steps: reducing the full matrix to band form and then reducing the band matrix to tridiagonal form. Thus, by reducing the communication and improving the algorithm for tridiagonalizing a band matrix, we can also improve algorithms for tridiagonalizing a full matrix. In fact, we can attain the communication lower bound that applies to the dense problem by using the two-step approach and the algorithms described in this chapter for the band reduction step. While we focus on real symmetric matrices in this chapter, the ideas here can be readily applied to tridiagonalization of complex Hermitian matrices as well as bidiagonalization of general matrices (for singular value problems).

The rest of the chapter is organized as follows. In Section 10.2, we extend the band reduction algorithm design space with new techniques for avoiding communication. The main novel contribution is the idea of chasing multiple bulges in the context of SBR. In Section 10.3, we give an asymptotic complexity analysis of previous approaches, and show how our new techniques can be used to improve their communication costs. We also introduce a new algorithm, CASBR, which communicates asymptotically less than all other approaches. In Section 10.4, we extend CASBR to a distributed-memory parallel algorithm which communicates asymptotically fewer messages than previous approaches.

All of the results in this chapter appear in [30], written with coauthors James Demmel and Nicholas Knight. A preliminary version of the work appeared in [31]. The multiple bulge chasing approach and sequential CASBR algorithm (for eigenvalues only) first appeared in that paper. We also showed how to extend the sequential algorithm to a shared-memory parallel environment, and our implementations obtained $2-6\times$ speedups over state-of-the-art library implementations. This chapter extends those results in two ways: we discuss distributed-memory algorithms and consider computing both eigenvalues and eigenvectors, though we do not give implementation details or performance results here.

10.1 Preliminaries

10.1.1 Eigendecomposition of Band Matrices

In this chapter, we are interested in computing the eigenvalues (and possibly the eigenvectors) of a symmetric band matrix via tridiagonalization. Let $A \in \mathbb{R}^{n\times n}$ be a symmetric band matrix with *bandwidth* b (*i.e.*, having $2b+1$ nonzero diagonals). Because we preserve symmetry, it is sufficient to store and operate on only the lower $b+1$ diagonals of A. We reduce A to a symmetric tridiagonal matrix T via orthogonal similarity transformations which comprise an orthogonal matrix Q such that $Q^TAQ = T$. We refer to this process as the *band reduction* phase. We assume the eigendecomposition of the tridiagonal matrix T is computed via an efficient algorithm such as Bisection/Inverse Iteration, MRRR, Divide-and-Conquer, or QR Iteration (*e.g.*, see [63]), and we ignore the computation and communication costs of this phase.

If only eigenvalues are desired, the eigenvalues of T are the eigenvalues of A, so no extra computation is required and Q need not be computed or stored. If eigenvectors are also desired, then a *back-transformation* phase is needed to reconstruct the eigenvectors of A from the eigenvectors of T. That is, if the eigendecomposition of T is given by $T = V\Lambda V^T$, then the eigendecomposition of A is $A = (QV)\Lambda(QV)^T$, so to compute the eigenvectors of A, we must compute QV. There are a range of possibilities for computing QV: if we form Q and V explicitly, then this can be done with matrix multiplication; if we store Q implicitly (*e.g.*, as a set of Householder vectors), then it can be applied to V after V is formed explicitly; if QR Iteration is used to compute the eigendecomposition of T, then Q should be formed explicitly so that V can be applied implicitly to Q from the right as it is computed; or Q and V can be left implicit, allowing us to multiply by them when needed.

In many applications only a subset of eigenpairs are desired. The cost of the back-transformation can be made proportional to the number of eigenpairs desired; this can significantly improve the runtime. Here, we consider only the case of computing all n eigenpairs.

One of the most important applications of solving the symmetric band eigenproblem is when solving the full symmetric eigenproblem. An efficient alternative to direct tridiagonalization [150] is a two-step approach [38]: (1) reducing the full matrix to a band matrix, and (2) reducing the band matrix to tridiagonal form. Both direct and two-step tridiagonalization approaches use orthogonal similarity transformations. The remainder of this chapter concerns step (2); we discuss step (1) briefly in Section 10.5.

10.1.2 SBR Notation

We follow notation from [38] and the authors' related papers to describe the terminology associated with successive band reduction (SBR), our approach for reducing A to T. While we do not give a complete description of SBR here, Figure 2 in [38] is particularly helpful for visualizing the framework.

To exploit symmetry, we store and operate on only the lower triangle of the band matrix, though analogous algorithms apply to the upper triangle. When we refer to a column of the band, we mean the entries of the column on and below the diagonal.

In a given *sweep*, SBR eliminates d subdiagonals in sets of c columns,[1] using an annihilate-and-chase approach. We assume Householder transformations are used; each set of transformations eliminates a d-by-c *parallelogram* of nonzeros but creates trapezoidal-shaped fill (a *bulge*). Using analogous orthogonal similarities, SBR chases each bulge off the end of the band, translating the bulge b columns to the right with each *bulge chase*. Figure 10.1 shows the data access pattern of a single bulge chase. A QR decomposition of the $(d+1+c)$-by-c matrix (QR region in Figure 10.1) containing the parallelogram computes the orthogonal matrix that annihilates the parallelogram; the corresponding rows (PRE region) are updated with a premultiplication of the orthogonal matrix; the corresponding columns (POST

[1]We depart from the LAPACK-style notation `nb` of [38].

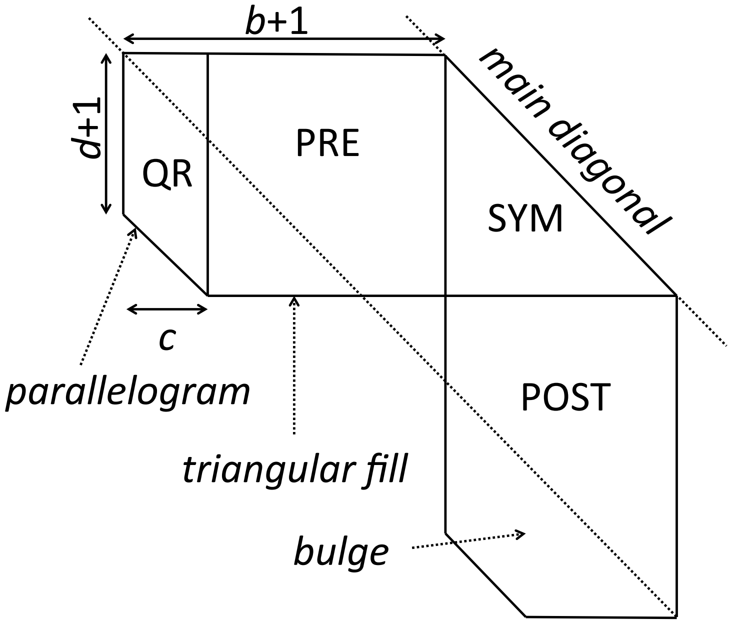


Figure 10.1: Anatomy of the bulge chasing operation. Following the notation of [38], the bulge chasing operation based on an orthogonal similarity transformation can be decomposed into four parts. There are d diagonals in each bulge and c is the number of columns annihilated during a bulge chase which leaves behind triangular fill.

region) are updated with a postmultiplication by the transpose of the orthogonal matrix, creating the next bulge; and the lower half of the corresponding symmetric submatrix on the diagonal (SYM region) is updated from both the left and right.

We define the *working bandwidth* $b + d + 1$ to be the number of subdiagonals necessary to store the $b + 1$ diagonals of the matrix as well as to store the d diagonals that hold temporary fill-in during the course of a sweep. As observed in [115], we note that an entire bulge need not be eliminated; only the first c columns of the bulge must be annihilated to prevent subsequent bulges from introducing nonzeros beyond the working bandwidth. This results in temporary *triangular fill*.

We index sweeps with an integer i, where $i = 1$ is the first sweep, so $b_1 = b$ is the initial bandwidth. We index the parallelograms which initiate each bulge chase by j and the sequence of following bulges by the ordered pairs (j, k): j is the parallelogram index and k is the bulge index, as in [41].

10.1.3 Related Work

In this section we discuss the previous approaches for band reduction in both sequential and parallel cases. For the most competitive algorithms, we provide more detailed communication cost analyses in later sections. See Tables 10.1–10.4 for summaries and comparisons of communication costs.

10.1.3.1 Sequential Algorithms

The two papers of Bischof, Lang, and Sun [38, 43] provide a general framework of sequential SBR algorithms. Their approach first appeared in [40] and generalizes most of the related work described in this section.

The annihilate-and-chase strategy began with Rutishauser and Schwarz in 1963. Rutishauser [126] identified two extreme points in the SBR algorithm design space: (1) a Givens rotation-based approach with b sweeps and $c_i = d_i = 1$ for each i and (2) a column-based approach with one sweep where $c_1 = 1$ and $d_1 = b - 1$. Rutishauser's first approach considered only pentadiagonal matrices; Schwarz [132] generalized the algorithm to arbitrary bandwidths. Later, Schwarz [133] proposed a different algorithm based on Givens rotations which does not fit in the SBR framework. This algorithm eliminates entries by column rather than by diagonal and does not generalize to parallelograms.

Murata and Horikoshi [115] improved on Rutishauser's column-based algorithm by noting that computation can be saved by eliminating only the first column of the triangular bulge rather than the entire triangle. If eigenvectors are desired, Bischof, Lang, and Sun [41] showed that, with this approach, the Householder vectors comprising Q can be stored in a lower triangular n-by-n matrix and applied to V in a different order than they were computed, yielding higher performance during the back-transformation phase.

Kaufman [98] vectorized the Rutishauser/Schwarz algorithm [126, 132], chasing multiple single-element bulges in each vector operation. Her motivation for chasing multiple bulges was not locality but rather to increase the length of the vector operation beyond the bandwidth b. Several years later, Kaufman [97] took the approach of [133] in order to maximize the vector operation length (especially in the case of large b) and make use of a BLAS subroutine when appropriate. When eigenvectors are requested, the Q matrix is formed explicitly by applying the updates to an identity matrix. By exploiting sparsity, the flop cost of constructing Q is about $(4/3)n^3$, compared with $2n^3$ if sparsity is ignored. The current LAPACK [8] reference code for band reduction (`sbtrd`) is based on [97].

More recently, Rajamanickam [123] proposed and implemented a different way of eliminating a parallelogram and chasing its fill. His algorithm uses Givens rotations to eliminate the individual entries of a parallelogram, and instead of creating a large bulge, the update rotations are pipelined such that as soon as an element is filled in outside the band, it is immediately annihilated. The rotations are carefully ordered to obtain temporal and sequential locality. By avoiding the fill-in, this algorithm does up to 50% fewer flops than the Householder-based elimination of parallelograms within SBR and requires minimal working bandwidth.

10.1.3.2 Parallel Algorithms

Lang [103, 102] implemented a distributed-memory parallel version of the band reduction algorithm in [115], although he did not consider computing Q. Bichof et al. [39] implemented a distributed-memory parallel instance of the SBR framework in the context of tridiagonalizing a full matrix. A subsequent paper [41] extended this implementation to reorganize and block the orthogonal updates comprising Q.

Luszczek et al. [109] implemented the band reduction algorithm from [115] as part of a two-step shared-memory tridiagonalization algorithm in the PLASMA library [6], using dynamic DAG-scheduling of tile-based tasks. They distinguished between "right-looking" and "left-looking" variants: right-looking algorithms chase a bulge entirely off the band before eliminating the next parallelogram, left-looking algorithms chase bulges only far enough to allow for the next bulge to be created (see Constraint 2). For example, the SBR framework [38] is right-looking while Kaufman's algorithm [97] is left-looking. In [109], they found improved performance with a left-looking variant. Later, Haidar et al. [84] reduced the runtime of [109]; the improvements in the band-to-tridiagonal step include using an algorithm-specific (static) scheduler, "grouping" related tasks, and avoiding fill-in using pipelined Givens rotations (a single-sweep version of the approach in [123]).

Auckenthaler et al. [10, 11, 12] have implemented a two-step distributed-memory tridiagonalization algorithm as part of a solver for the generalized symmetric eigenproblem. Their band-to-tridiagonal step uses an improved version of Lang's algorithm [102], which performs one sweep. They give a new algorithm for orthogonal updates which uses a 2D processor layout instead of a 1D layout. Their implementation also supports taking multiple sweeps when eigenvectors are not requested; however, this algorithm is not given explicitly.

10.1.4 Related Lower Bounds

No communication lower bound has been established for annihilate-and-chase band reduction algorithms, so we cannot conclude that our new algorithms are communication optimal in an asymptotic sense. In fact, Theorem 4.25 in Section 4.3, which applies to many algorithms that use orthogonal transformations, does not apply to SBR or its variants because they fail to satisfy forward progress (Definition 4.18). That is, the lower bound proof there requires that an orthogonal transformation algorithm not fill in a previously created zero—this occurs frequently in SBR, unlike QR decomposition.

The main results of Chapter 4 state that an applicable algorithm that performs G flops must move $\Omega(G/\sqrt{M})$ words and send $\Omega(G/M^{3/2})$ messages for sufficiently large problems. For most dense matrix algorithms, the number of flops is $G = O(n^3/P)$, where $P = 1$ for the sequential case. In the parallel case, if we assume minimal local memory is used (*i.e.*, $M = \Theta(n^2/P)$, or just enough to store the input and output matrices), the the lower bounds simplify to $\Omega(n^2/\sqrt{P})$ words and $\Omega(\sqrt{P})$ messages.

Since our new sequential algorithm (see Algorithm 10.1 and Table 10.1) performs $O(n^2b)$ flops, moves $O(n^2b^2/M)$ words, and sends $O(n^2b^2/M^2)$ messages, its bandwidth and latency costs drop below the lower bounds by a factor of $O(\sqrt{M}/b)$ for $2 \leq b \leq \sqrt{M}/3$. For small b and large n (such that the band does not fit entirely in fast memory), this discrepancy is as much as $O(\sqrt{M})$. Similarly, our new parallel algorithm (see Algorithm 10.3 and Table 10.3) also beats the lower bounds for bandwidth and latency costs, and the discrepancy is largest for small bandwidths. Thus, our algorithms show that not only does the lower bound proof technique not apply to annihilate-and-chase algorithms, the bound itself must not apply.

10.2 Avoiding Communication in Successive Band Reduction

The goal of our algorithms is to avoid communication by reorganizing computation, extending the SBR framework to obtain greater data locality. In the sequential case, we can asymptotically reduce the number of words and messages that must be moved between fast and slow memory during the execution of the algorithm; in the parallel case, we can asymptotically reduce the number of messages sent between processors. We achieve data locality (*i.e.*, avoid communication) using two techniques described in Sections 10.2.1 and 10.2.2. We navigate the constraints and tradeoffs that arise using a successive halving approach, described in Section 10.2.3.

10.2.1 Applying Multiple Householder Transformations

The first means of achieving data locality is within a single bulge chase (see Figure 10.1). Since c Householder vectors are computed to eliminate the first c columns of the bulge (QR region), every entry in the PRE, SYM, and POST regions is updated by c left and/or right Householder transformations. These transformations may be applied one at a time or blocked (*e.g.*, via [131]). Assuming all the data involved in a single bulge chase reside in fast or local memory, $O(c)$ flops are performed for every entry read from slow memory.

We identify the following algorithmic constraint. If it is violated, then the parallelogram annihilated by the left update will be (partially) refilled by the right update (*i.e.*, the SYM and POST regions overlap the QR region)—this implies wasted computation.

Constraint 1. *To annihilate a parallelogram within the SBR framework, the dimensions of the parallelogram must satisfy*

$$c + d \leq b.$$

While increasing c improves data locality, it limits the size of d due to Constraint 1. Because d is the number of diagonals eliminated in a sweep, this constraint creates a tradeoff between locality and progress towards tridiagonal form.

10.2.2 Chasing Multiple Bulges

The second means of achieving data locality is across bulge chases. If ω bulges can be chased through the same set of columns without data movement, then we have achieved $O(\omega)$ reuse of those columns. Recall that we refer to columns as the subset of column entries on and below the diagonal. We first establish the following constraint.

Constraint 2. *No bulge may be chased into a set of columns still occupied by a previously created bulge.*

If this constraint is violated, then the fill will expand beyond the working bandwidth of the sweep. While it is possible to eliminate this extra fill, we wish to avoid the extra computation and storage necessary to do so. Chasing the first c columns of a bulge and leaving behind the triangular fill is the least amount of work required to prevent the fill from exceeding the working bandwidth.

We state the following lemmas regarding parallelograms, bulges, sets of bulges, and the working set (measured in columns) for chasing a set of bulges. We assume in both cases that Constraints 1 and 2 are satisfied.

Lemma 10.1. *Given a sweep of SBR with parameters b, c, and d,*

(a) the jth parallelogram occupies columns $1 + (j-1)c$ through jc,

(b) bulge (j,k) occupies columns $1 + (j-1)c + kb - d$ through $jc + kb$,

(c) bulges (j,k) and $(j+1, k-2)$ do not overlap.[2]

Lemma 10.2. *Chasing the set of ω bulges*

$$\{(j,k),(j+1,k-2),\ldots,(j+\omega-1,k-2(\omega-1))\}$$

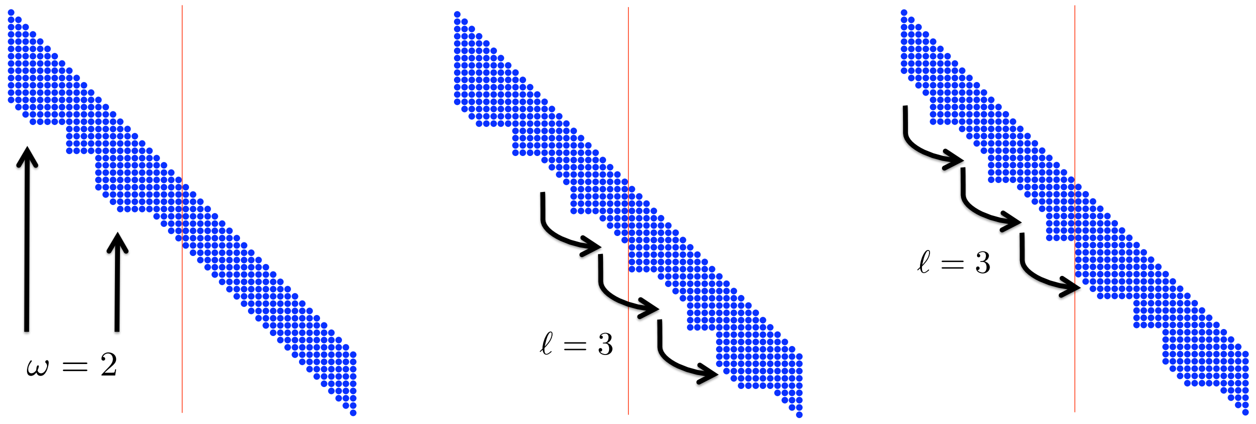
each ℓ times requires a working set of $(\omega-1)(2b-c)+c+d+b\ell$ columns.

Proof. By Lemma 10.1(c), this set of bulges is nonoverlapping. If the bulges are chased in turn ℓ times each, starting with the right-most bulge (j,k) and ending with the left-most bulge $(j+\omega-1, k-2(\omega-1))$, then there is no violation of Constraint 2. The conclusion follows from Lemma 10.1(b). □

Figure 10.2 demonstrates the working set of 44 columns with $\omega = 2$ bulges chased $\ell = 3$ times each on a matrix with bandwidth $b = 8$ with $c = d = 4$.

Our motivation for defining a working set is to ensure that the operation of chasing ω bulges ℓ times can be done entirely in fast memory (in the sequential case) or local memory (in the parallel case). We will specify the constraints in each case when we present our algorithms below.

[2]Note that if $2c + d \leq b$, bulges (j,k) and $(j+1,k-1)$ also do not overlap.



(a) The $\omega = 2$ bulges occupy 20 of the 24 columns on the left.

(b) The first (right-most) bulge is chased $\ell = 3$ times.

(c) The second bulge is chased $\ell = 3$ times.

Figure 10.2: Chasing a set of bulges. We store and operate on only the lower triangle of the band matrix. The parameters shown are $b = 8$, $c = 4$, and $d = 4$; $\omega = 2$ bulges are chased $\ell = 3$ times each. Only $2b\ell = 48$ columns of the band are shown. The working bandwidth includes the diagonals which contain bulges and triangular fill. Note that the triangular fill left behind by the first bulge does not cause any increase in the working bandwidth as the second bulge is chased.

10.2.3 Successive Halving

We will navigate the tradeoff imposed by Constraint 1 by setting $c_i = d_i = b_i/2$ at each sweep i, reducing to tridiagonal form after $\log b$ sweeps. We call this a *successive halving* approach. We will pick the number of bulges in a set (ω_i) and the number of times each bulge is chased (ℓ_i) such that on each sweep (as the bandwidth is successively halved) we double the number of bulges that we chase in a set, and chase each bulge twice as many times, compared to the previous sweep. While the successive halving approach (and doubling ω_i and ℓ_i) simplifies our asymptotic analysis, in practice the parameters $\{c_i, d_i, \omega_i, \ell_i\}$ should be tuned independently for best performance—we suggested a framework for automatically tuning these parameters in a shared-memory implementation [31, Section 5].

10.3 Sequential Band Tridiagonalization Algorithms

Recall our sequential machine model, where communication is moving data between slow memory of unbounded capacity and a fast memory with a capacity of M words. We will first consider the case of computing eigenvalues only and then extend to the case of computing both eigenvalues and eigenvectors. We will not analyze the solution of the tridiagonal eigenproblem. In each case, we discuss existing approaches, apply our techniques to improve them, and then present our communication-avoiding approach.

For our sequential algorithms, we will assume the initial bandwidth b is bounded above by $\sqrt{M}/3$. As mentioned in Sections 10.1.1 and 10.5, this is a reasonable assumption if the band reduction is used as the second step of a two-step reduction of a full symmetric matrix to tridiagonal form. For larger bandwidths, another approach must be taken to avoid communication (see Section 10.5). We also assume that $nb \gg M$ (the band does not fit in fast memory).

10.3.1 Computing Eigenvalues Only

When only eigenvalues are desired, the runtime is dominated by the band reduction. Computing the eigenvalues of a tridiagonal matrix involves only $O(n)$ data and less computation than the band reduction—$O(n^2)$ as opposed to $O(n^2b)$. While there is a large design space for band reduction, the computational cost ranges from $4n^2b$ to $6n^2b$, a difference of only 50% (as long as a bulge-chasing procedure is used to prevent unnecessary fill). However, the communication cost (and expected performance) has a much larger range.

Under the assumption above, the matrix does not fit in fast memory (otherwise, the communication costs are the same for all algorithms: $O(nb)$). In the case that $n < M$ (*i.e.*, one or more diagonals fit in fast memory), when the bandwidth is reduced such that the remaining band matrix fits in fast memory, the communication cost of remaining sweeps is that of reading the band into fast memory once and writing the tridiagonal output.

Table 10.1 summarizes the computation and communication costs of various algorithms for tridiagonalizing a band matrix (for computing eigenvalues only). Our new approach, CASBR, improves the communication costs compared to the previous approaches. For example, CASBR moves a factor of M/b fewer words than LAPACK or MH, which is at least $\sqrt{M}$ in the range of b considered, and near M for $b = O(1)$. Note that while the computational costs vary only by constant factors, these factors can make a difference in practice. The tradeoffs between different algorithms and between computation and communication should be navigated with autotuning (of algorithms and parameters) in practice. In the context of two-step tridiagonalization of a dense matrix, CASBR is the only approach that always attains (or beats) the lower bounds discussed in Section 10.1.4.

10.3.1.1 Alternative Algorithms

We first consider Kaufman's algorithm [97], which is implemented in the current LAPACK reference code [8], given in the first row of Table 10.1. The algorithm uses Givens rotations and performs $4n^2b$ flops. It is left-looking and chases multiple single-element bulges in order to maximize the vector operation length, but it does not limit the size of the working set to fit in fast memory. As a result, the algorithm has to read (from slow memory) at least one of each pair of entries to be updated by a Givens rotation. Thus, the data reuse is $O(1)$ and the total number of words transferred between fast and slow memory is proportional to the number of flops: $O(n^2b)$. Since fine-grained data access occurs along both rows and

Algorithm	**Flops**	**Words**	**Messages**
LAPACK [97]	$4n^2b$	$O(n^2b)$	$O(n^2b)$
MH [115]	$6n^2b$	$O(n^2b)$	$O\left(\frac{n^2b}{M}\right)$
Improved MH	$6n^2b$	$O\left(\frac{n^2b^3}{M}\right)$	$O\left(\frac{n^2b^3}{M^2}\right)$
SBR [38]	$\sum_{i=1}^{s}\left(4d_i+2\frac{d_i^2}{b_i}\right)n^2$	$O\left(\sum_{i=1}^{t}\left(1+\frac{d_i}{b_i}\right)n^2\right)$	$O\left(\sum_{i=1}^{t}\left(1+\frac{d_i}{b_i}\right)\frac{n^2}{M}\right)$
SBR ($c_i = d_i = b_i/2$)	$5n^2b$	$O(n^2t)$	$O\left(\frac{n^2t}{M}\right)$
CASBR	$5n^2b$	$O\left(\frac{n^2b^2}{M}\right)$	$O\left(\frac{n^2b^2}{M^2}\right)$

Table 10.1: Asymptotic comparison of previous sequential algorithms for tridiagonalization (for eigenvalues only) with our improvements, for symmetric band matrices of n columns and $b+1$ subdiagonals on a machine with fast memory of size M. The table assumes that $nb \gg M$ and that $2 \le b \le \sqrt{M}/3$. The analysis for all algorithms is given in Section 10.3. In the fourth and fifth rows, s is the number of sweeps performed and $t \le s$ is the smallest sweep index such that the subsequent sweeps can be performed in fast memory, or $t = s$ otherwise.

columns, the latency cost is on the same order as the bandwidth cost, assuming LAPACK's column-major layout.

Next, we consider the Householder-based approach of Murata and Horikoshi [115], given as MH in the second row of Table 10.1. In this algorithm, each column is eliminated all at once, and the bulge is chased completely off the band before the next column is eliminated. Because of operations on the triangular fill, the number of flops required increases to $6n^2b$ compared to Givens-based algorithms. Since each bulge is chased entirely off the band, the entire band must be read from slow memory for every column eliminated, a total of $O(n^2b)$ words moved. Assuming column-major layout, the sequence of bulge chases for each column (*i.e.*, bulges (j,k) for fixed j) is executed on contiguous data, and the latency cost is a factor of $O(M)$ less than the bandwidth cost.

In order to reduce communication costs for the MH algorithm it is possible to apply one of the optimizations described in Section 10.2: chasing multiple bulges. From Lemma 10.2, we can chase $O(M/b^2)$ bulges at a time and maintain a working set which fits in fast memory. This results in a reduction of both bandwidth and latency costs by a factor of $O(M/b^2)$. We call this algorithm "Improved MH," given in the third row of Table 10.1.

Consider an algorithm within the SBR framework with parameters $\{(b_i, c_i, d_i)\}_{i=1,2,\ldots,s}$, which does not chase multiple bulges at a time (*i.e.*, $\omega_i = 1$ for every i). This corresponds to the fourth row of Table 10.1. The flop count is given by [38, Equation (3)] (and Lemma 10.7 below). Note that the approach of [123] allows the computational cost to be reduced to $4n^2b$ for all parameter choices. Since the SBR framework is right-looking, the trailing band must be read for each parallelogram eliminated. During the ith sweep, there are $O(n/c_i)$

parallelograms and each parallelogram is chased $O(n/b_i)$ times. The amount of data accessed during one bulge chase is $O(b_i(c_i + d_i))$ words—for example, b_i columns are accessed during the left update and each bulge occupies $c_i + d_i$ rows. Thus, the number of words read during the ith sweep is $O(n^2(1 + d_i/c_i))$. In the best case, the latency cost is a factor of M smaller than the bandwidth cost.

If we apply the successive halving approach ($c_i = d_i = b_i/2$) but do not chase multiple bulges, then the costs of SBR simplify to $O(n^2t)$ words (where $t \leq \log b$ is the smallest sweep index such that $n(b_t + d_t + 1) \leq M$, or $t = \log b$ otherwise) and $O(n^2t/M)$ messages, in the best case. These costs appear in the fifth row of Table 10.1.

10.3.1.2 CASBR

The communication avoiding sequential algorithm, shown in Algorithm 10.1, is based on the framework given in [38], using the successive halving approach (see Section 10.2.3). Our main deviation from the original SBR framework is chasing multiple bulges at a time, as described in Section 10.2.2. Recall that ω_i denotes the number of bulges chased at a time, and ℓ_i the number of times each bulge is chased, during sweep i. We would like to maximize ω_i so that for some $\ell_i \geq 1$, this working set fits in a fast memory of size M words. We ignore the sparsity below the b_ith subdiagonal by assuming each column has $b_i + d_i + 1$ nonzeros (*i.e.*, the working bandwidth). It follows from Lemma 10.2 that we would like to pick positive integers ω_i and ℓ_i such that ω_i is maximized and

$$((\omega_i - 1)(2b_i - c_i) + c_i + d_i + b_i\ell_i)(b_i + d_i + 1) \leq M. \tag{10.1}$$

We use a successive halving approach, as mentioned above. That is, at each sweep i, we cut the remaining bandwidth b_i in half by setting $d_i = b_i/2$. We also set $c_i = b_i/2$ (which satisfies Constraint 1). To simplify the analysis, we assume that the initial bandwidth $b = b_1$ is a power of two.

As in Lemma 10.2, when chasing a set of ω_i bulges, we work right-to-left, chasing each bulge $\ell_i = (3/2)\omega_i$ times in turn. In this way, after all bulges in the set are chased, the set does not overlap the previous columns occupied, and the relative positions of the bulges are maintained. This process is shown in Figure 10.2 and corresponds to line 9 in Algorithm 10.1. Fixing ℓ_i in terms of ω_i also has the benefit of decreasing the latency cost on successive sweeps. While the constant ratio between ω_i and ℓ_i simplifies theoretical analysis, these parameters can be tuned independently in practice.

With these parameter choices and assumptions, inequality (10.1) simplifies, as given in the following constraint.

Constraint 3. *Assuming b and ω are even, $c = d = b/2$, $\ell = (3/2)\omega$, and $b \leq \sqrt{M}/3$, then the number of bulges chased at a time must not exceed $\omega \leq 4M/(9(b+1)^2)$.*

By satisfying Constraint 3, we ensure that the entire operation can be performed on columns which all fit in fast memory simultaneously.

We include explicit memory operations within the algorithm in order to determine the communication costs: *writes* imply moving data from fast memory to slow memory and *reads* imply moving data from slow memory to fast memory.

Algorithm 10.1 Sequential CASBR

```
Require: initial bandwidth $b \leq \sqrt{M}/3$ is a power of 2
1: $t = \min\{\log b, \left\lceil \log \frac{n(b+1)}{M} \right\rceil\}$
2: for $i = 1$ to $t$ do
3:     $b_i = \frac{b}{2^{i-1}}$, $c_i = \frac{b_i}{2}$, $d_i = \frac{b_i}{2}$, $\omega_i = 2\left\lfloor \frac{2M}{9(b_i+1)^2} \right\rfloor$, $\ell_i = \frac{3}{2}\omega_i$
4:     while not reached end of band do
5:         create next set of $\omega_i$ bulges
6:         while not reached end of band do
7:             write previous $\ell_i b_i$ columns of band
8:             read next $\ell_i b_i$ columns of band
9:             chase $\omega_i$ bulges $\ell_i$ times each
10:        end while
11:        chase $\omega_i$ bulges off the end of the band
12:    end while
13:    copy band into data structure with column height $\frac{3}{2}b_{i+1}$
14: end for
15: if $t < \log b$ then
16:    read remaining band into fast memory
17:    reduce band to tridiagonal
18:    write output to slow memory
19: end if
```

We omit the details of creating a set of bulges (line 5) and of chasing bulges at the end of the band (line 11). Both the arithmetic and communication costs of creating ω_i bulges or chasing ω_i bulges off the end of the band are dominated by that of chasing the ω_i bulges ℓ_i times each. Also, since neither operation occurs in the inner loop of the algorithm, they contribute only lower order terms to the costs of the entire algorithm.

The computation of t in line 1 determines the sweep (if any) after which the remaining band fits entirely in fast memory. Note that if $n > M$, then the band will never fit in fast memory and $t = \log b$. If the band becomes small enough to fit in fast memory, then the algorithm will stop the main loop (lines 2–14) and fall to the clean-up code in lines 15–19 which simply reads the band into fast memory, reduces to tridiagonal form, and writes the result back to slow memory.

10.3.1.3 Arithmetic Cost

In order to count the number of flops required by Algorithm 10.1, we first establish two lemmas related to the cost of applying Householder transformations.

Lemma 10.3. *Applying a Householder transformation from the left,* $\mathrm{House}(u) \cdot A$, *costs no more than* $4hc + h - c$ *flops, where* h *is the number of nonzeros in* u *and* A *has* c *columns. Equivalently, applying the transformation from the right,* $A \cdot \mathrm{House}(u)$, *costs no more than* $4hr + h - r$ *flops if* A *has* r *rows.*

Proof. The first statement is verified by counting the operations in $A :- A - (\tau u)\left(u^T A\right)$. The second statement is verified by transposing the first transformation. □

Lemma 10.4. *Applying a Householder transformation symmetrically to an* n*-by-*n *symmetric matrix* A, $\mathrm{House}(u) \cdot A \cdot \mathrm{House}(u)^T$, *costs no more than* $(4h-1)n + 5h$ *flops, where* h *is the number of nonzeros in* u.

Proof. We perform three steps: $y := A(\tau u)$, $v := y - (1/2)\left(y^T u\right) u$, and $A := A - uv^T - vu^T$. The first step costs $(2h-1)n + h$ operations, the second $4h - 1$, and the third $2nh$, if we exploit symmetry. □

Given these lemmas, we can compute the arithmetic cost of a single bulge chase.

Lemma 10.5. *A single bulge chase costs* $8bcd + 4cd^2 + O(bc)$ *operations. Creating a bulge, or clearing a bulge (off the end of the band), is less expensive.*

Proof. We refer to the four operations depicted in Figure 10.1. Let $1 \le m \le c$ index the (unblocked) Householder transformations that eliminate the parallelogram in the QR region. Transformation m is applied from the left to $c-m$ columns in the QR region and $b-c$ columns in the PRE region, from the right to $b-(c-m)$ rows in the POST region, and symmetrically to a $(d+c)$-by-$(d+c)$ symmetric matrix in the SYM region. Applying Lemmas 10.3 and 10.4, transformation m performs $8bd + 4d^2 + O(b)$ flops, and there are c transformations. Creating a bulge is less expensive because the PRE region includes only $b - c - d$ columns. As a result, transformation m does fewer flops. Clearing a bulge is less expensive because there are fewer rows in the POST region. □

See [31, Section 5.3] for a discussion of different approaches to chasing individual bulges and their implications on performance.

We can also count the number of bulge chases that occur during each sweep.

Lemma 10.6. *The number of bulges chased during a sweep with parameters* n, b, c, *and* d *is* $n^2/(2bc) + O(n/b)$.

Proof. For each parallelogram eliminated, the bulge must be chased the length of the trailing band, in increments of b columns. Thus, the total number of bulge chases during a sweep is $\sum_{j=1}^{n/c}(n - jc)/b = n^2/(2bc) + O(n/b)$. □

Lemmas 10.5 and 10.6 together imply the following fact, which agrees with [38, Equation (3)].

Lemma 10.7. *The arithmetic cost of eliminating d diagonals from a matrix with bandwidth b using SBR is $(4d + 2d^2/b)\, n^2 + O(n^2)$.*

The order of operations specified by the algorithm does not affect the arithmetic count, provided Constraints 1 and 2 are satisfied. Given the cost of the ith sweep specified by Lemma 10.7, since $d_i = b_i/2$ and $\sum_i d_i = b - 1$, the arithmetic cost of Algorithm 10.1 (ignoring lower order terms) is

$$\sum_{i=1}^{\log b} \left(4d_i + 2\frac{d_i^2}{b_i} \right) n^2 = 5n^2 b.$$

10.3.1.4 Bandwidth Cost

In determining the communication costs of Algorithm 10.1, we must consider two cases. If $n > M$, then $\log b < \lceil \log(n(b+1)/M) \rceil$ and the main loop (lines 2–14) will be executed $\log b$ times, reducing the band to tridiagonal. However, if $n < M$, then at some point the bandwidth will become small enough such that the entire band fits in fast memory. At this point, the algorithm reduces to lines 15–19 and the only communication required to finish the reduction is that of reading the band into fast memory and writing the tridiagonal output back to slow memory for a cost of $O(nb_{t+1})$ words.

We now consider the ith sweep, where we assume the band is too large to fit in fast memory. The dominant communication cost is in the innermost loop (lines 6–10). The number of words in each column is $(3/2)b_i$, so the bandwidth cost of one iteration of the inner loop is $3\ell_i b_i^2 = O(M)$ words. The inner loop is executed $O(n/(\ell_i b_i))$ times for each set of bulges, and there are $O(n/(c_i\omega_i))$ sets of bulges during the sweep. Thus, the bandwidth cost of one sweep is $O(n^2 b_i^2/M)$ words.

The bandwidth cost (*i.e.*, number of words moved) of Algorithm 10.1 is then

$$\sum_{i=1}^{t} O\left(\frac{n^2 b_i^2}{M} \right) + O(nb_{t+1}) = O\left(\frac{n^2 b^2}{M} + nb \right).$$

10.3.1.5 Latency Cost

We will assume the band matrix is stored in LAPACK symmetric band storage format (column-major with column height equal to the working bandwidth) so that any block of columns of the band will be stored contiguously in slow memory. After each set of subdiagonals is annihilated from a column block, the algorithm packs the remaining diagonals into a smaller data structure (see line 13) to maintain a packed column-major layout for all successive sweeps. This increases the memory footprint by no more than a factor of two and can also be done in place, and it adds only lower order terms to the bandwidth and latency costs.

As in the previous section, if the band becomes small enough to fit in fast memory, then the communication costs of completing the algorithm are reduced to reading the band and writing the tridiagonal output. In this case, the latency cost is 2 messages. When the band is too large to fit in fast memory, the dominant latency cost is that of the innermost loop. Since consecutive columns are stored contiguously, the latency cost per iteration of the innermost loop is 2 messages. As argued above, the inner loop is executed $O(n/(\ell_i b_i))$ times for each set of bulges, and there are $O(n/(c_i \omega_i))$ sets of bulges during the sweep. Thus, the latency cost of one sweep is $O(n^2 b_i^2/M^2)$ messages.

The latency cost (*i.e.*, number of messages moved) of Algorithm 10.1 is then

$$\sum_{i=1}^{t} O\left(\frac{n^2 b_i^2}{M^2}\right) + O(1) = O\left(\frac{n^2 b^2}{M^2} + 1\right).$$

10.3.2 Computing Eigenvalues and Eigenvectors

When only eigenvalues are desired, the orthogonal similarity transformations that reduce the band matrix to tridiagonal form may be discarded. However, when eigenvectors are desired, these transformations must be used to reconstruct the eigenvectors QV of the band matrix from the eigenvectors V of the tridiagonal matrix.

Compared to Section 10.3.1, the main difference between computing eigenvalues and additionally eigenvectors is that the arithmetic cost of computing QV increases with the number of sweeps taken in the band reduction. While the arithmetic cost of the band reduction for the algorithms discussed in Section 10.3.1 ranges from $4n^2b$ to $6n^2b$, that of the back-transformation ranges from $2n^3$ up to $n^3 \log b$.

The orthogonal matrix Q can be constructed explicitly by applying the updates from the band reduction to an n-by-n identity matrix. Some flops may be saved when starting from the identity matrix (compared to applying them to a dense matrix, see *e.g.*, [97]), but the entries fill in quickly after one sweep, and we will ignore this savings in our analysis. Then, the arithmetic cost of computing QV given V is the cost of a matrix multiplication, $2n^3$ flops. However, the cost of this matrix multiplication can be avoided by storing Q implicitly as a set of Householder vectors and applying them to V. While this choice does not affect our theoretical analysis of CASBR, it should be considered in practice. Storing the Householder information for each sweep requires extra memory for at most $n^2/2$ entries per sweep.

Table 10.2 shows the computation and communication costs for various approaches to tridiagonalizing a band matrix (for computing both eigenvalues and eigenvectors).

Recall that one context of this work is two-step tridiagonalization; the communication lower bounds referenced in Section 10.1.4 apply to the first step (full-to-banded), but not the second step. However, note that a lower bound for part of the algorithm gives a valid lower bound for the whole algorithm. So, we will compare the approaches in Table 10.2 (the second step) and see which attain the lower bounds of $\Omega(n^3/\sqrt{M})$ words moved and $\Omega(n^3/M^{3/2})$

Algorithm	Flops	Words	Messages
LAPACK [97]	$2n^3$	$O(n^2b + n^3)$	$O\left(n^2b + \frac{n^3}{M}\right)$
Improved LAPACK	$2n^3$	$O\left(n^2b + \frac{n^3}{\sqrt{M}}\right)$	$O\left(n^2b + \frac{n^3}{M}\right)$
BLS [41]	$2n^3$	$O\left(n^2b + \frac{n^3}{\sqrt{M}}\right)$	$O\left(\frac{n^2b}{M} + \frac{n^3}{M}\right)$
Improved BLS	$2n^3$	$O\left(\frac{n^2b^3}{M} + \frac{n^3}{\sqrt{M}}\right)$	$O\left(\frac{n^2b^3}{M^2} + \frac{n^3}{M^{3/2}}\right)$
CASBR	tn^3	$O\left(\frac{n^2b}{\sqrt{M}} + \frac{tn^3}{\sqrt{M}}\right)$	$O\left(\frac{tn^2}{M} + \frac{tn^3}{M^{3/2}}\right)$

Table 10.2: Asymptotic comparison of previous sequential algorithms for tridiagonalization (for eigenvalues and eigenvectors) with our improvements, for symmetric band matrices of n columns and $b+1$ subdiagonals on a machine with fast memory of size M. We include the cost of the back transformation (but not the cost of the tridiagonal eigendecomposition). The table assumes that $nb \gg M$ and that $2 \le b \le \sqrt{M}/3$. The two terms in the communication costs correspond to the band reduction and back transformation, respectively. In the last row, $t = O(\min\{\log b, \log(nb/M)\})$.

messages sent; both bounds are attainable by the first step by setting $b = \Theta(\sqrt{M})$. We claim that only CASBR attains these expected lower bounds for all ranges of parameters we consider, within a factor of $t = O(\log M)$.

Clearly the costs of LAPACK asymptotically exceed the lower bounds. If $n \ll M$, the bandwidth costs of the band reduction for Improved LAPACK, BLS, and Improved BLS asymptotically exceed the lower bound. If $n \gg M$, then the bandwidth costs of those three approaches match the lower bound, and the latency cost of Improved BLS also matches the lower bound.

Fact 1. *The computational cost of applying all the updates from a single band reduction sweep to a dense n-by-n matrix is $2\frac{d}{b}n^3$, ignoring lower order terms.*

Proof. From Lemma 10.6, there are $n^2/(2bc)$ bulge chases, each consisting of c Householder vectors of length $d+1$. From Lemma 10.3, the cost of applying each Householder transformation to an n-by-n matrix is $4(d+1)n$, so the total arithmetic cost is $4dn \cdot (n^2/(2b)) = 2(d/b)n^3$, ignoring lower order terms. □

10.3.2.1 Alternative Algorithms

As mentioned in Section 10.3.1.1, the current LAPACK reference code for band reduction (`sbtrd`) is based on [97]. When eigenvectors are requested, Q can be either explicitly formed or applied to an input matrix. The LAPACK routine for solving the eigenproblem for a band matrix (`sbevd`) forms Q explicitly and premultiplies V by it. The arithmetic cost of forming Q is approximately $(4/3)n^3$ [97], and the cost of the matrix multiplication is $2n^3$. In Table 10.2, we do not count the cost of computing Q, because the Givens rotations can be stored, reordered, and later applied to V for a total of $2n^3$ flops, although LAPACK does not offer this functionality.

The communication cost of the band reduction is analyzed in Section 10.3.1.1. Assuming the stored Givens rotations are applied to the rows of V one at a time (which is how they are accumulated in Q in LAPACK), at least one of the rows must be read from slow memory, and the data reuse is $O(1)$. This implies that the bandwidth cost of the band reduction, which is $O(n^2b)$, is dominated by the cost of the orthogonal updates. In the best case, if V is stored in row-major order and $n > M$, the latency cost is $O(n^3/M)$.

In [41], the authors consider an alternative approach for computing both eigenvalues and eigenvectors of a band matrix, in the context of a 2-step reduction of a full symmetric matrix. The band reduction scheme follows the algorithm of [115] consisting of one sweep (*i.e.*, $d = b - 1$ and $c = 1$). The key idea from [41] is to store all of the Householder vectors and, instead of applying them to V in exactly the reverse order that they were computed, to use a reordering technique that respects the dependency pattern. This reordering allows for the orthogonal updates to be blocked. See Figure 2 in [41] or Figure 2 in [11] for illustrations of this technique. Since the band reduction is performed in one sweep, the arithmetic cost is $2n^3$. Using the reordering technique with a blocking factor of size $\Theta(\sqrt{M})$, the communication cost of the orthogonal updates is $O(n^3/\sqrt{M})$. While the orthogonal updates are performed efficiently, the data reuse obtained during the band reduction is $O(1)$, as explained in Section 10.3.1.1. Thus, the bandwidth cost of the band reduction is $O(n^2b)$ which dominates the total bandwidth cost for $b \gg n/\sqrt{M}$. In the best case, the latency cost of the band reduction is $O(n^2b/M)$. In order to determine the latency cost of the orthogonal updates, we assume the matrix V is stored in column-major order and the Householder vectors are written to memory in the order they are computed. Then every application of a block of Householder vectors involves $O(\sqrt{M})$ messages, and so the latency cost is a factor of $O(\sqrt{M})$ less than the bandwidth cost. We refer to this as BLS, given in the third row of Table 10.2.

Note that this same reordering optimization from [41] can be used to improve the LAPACK algorithm. That is, the Givens rotations may be reordered and applied to V in a blocked fashion. For examples of implementations for applying blocks of Givens rotations, see [123, 147]. If the right block size is chosen, the bandwidth cost of the orthogonal updates can be reduced to $O(n^3/\sqrt{M})$. We refer to this algorithm as "Improved LAPACK," given in the second row of Table 10.2. Because of better alternatives, we do not discuss improvements in the latency cost.

We can apply two optimizations to reduce the communication costs of the BLS algorithm. First, as noted in Section 10.3.1.1, when $b \ll \sqrt{M}$, the communication costs of the band reduction can be improved by chasing $O(M/b^2)$ bulges at a time, reducing both the bandwidth and latency costs by a factor of $O(M/b^2)$.

Second, we can reduce the latency cost in performing the orthogonal updates by storing the eigenvector matrix V in a block-contiguous layout with block size C-by-C with $C = \Theta(\sqrt{M})$ and by performing a data layout transformation of the temporary data structure of Householder vectors. In order to minimize bandwidth cost, the Householder vectors corresponding to eliminating $\Theta(\sqrt{M})$ columns and chasing their respective bulges off the band should be temporarily stored before applying them to Q.

In order to analyze the data layout transformation, we need to consider the temporary storage of Householder vectors. If we let H be the temporary storage matrix, then we can store each Householder vector associated with the same eliminated column of A in the same column of H. Further, each vector can occupy the rows of H corresponding to the rows of A it updated; in this way, H is an n-by-n lower triangular matrix. If one bulge is chased at a time and Householder vectors are written to H in the order they are computed, then H will have a column-major data layout. However, in order to improve data reuse in applying the vectors to V, we want to apply parallelograms of vectors at a time, so we need those parallelograms to be stored contiguously. The data layout transformation is equivalent to transforming a matrix in column-major layout to a block-contiguous layout. By applying (for example) the `Separate` function given as Algorithm 3 in [32] to each panel of width $\Theta(\sqrt{M})$ a logarithmic number of times, we can convert H from column-major to $\Theta(\sqrt{M})$-by-$\Theta(\sqrt{M})$ block-contiguous layout with total bandwidth cost $O(n^2 \log(n/\sqrt{M}))$ and total latency cost $O((n^2/M)\log(n/\sqrt{M}))$, which are lower order terms for $n \gg \sqrt{M}$.

Note that these two optimizations cannot both be applied straightforwardly to the approach of [41], as H will not be written in column-major order when multiple bulges are chased at a time. We claim that a more complicated data layout transformation is possible in the case that multiple bulges are chased at a time. This costs of this algorithm are given as "Improved BLS" in the fourth row of Table 10.2. We also claim it is possible to apply the second optimization to the LAPACK algorithm, though the order in which the Givens rotations are computed and the method for temporarily storing them is more complicated.

10.3.2.2 CASBR

Algorithm 10.2 is a modification of Algorithm 10.1 which includes the explicit formation of the matrix Q, which we store in a block-contiguous layout with C-by-C blocks. An important difference between the two algorithms is the definition of ω_i, the number of bulges chased at a time. In Algorithm 10.1, ω_i is maximized under the constraint that the working set of data to chase ω_i bulges ℓ_i times each remains of size $O(M)$. In Algorithm 10.2, ω_i is further limited so that the working set of data while applying the Householder updates to a block row of the intermediate Q matrix remains of size $O(M)$. This working set of data now consists of three components: a subset of A, Householder transformations (temporarily stored in a data structure $\mathcal{H}$), and blocks of Q. We will pick ω_i to be approximately the square root of the previous choice so that each of these three components occupies no more than a third of fast memory. Reducing ω_i results in more communication cost during the band reduction, but we will see that this cost is always dominated by that of the orthogonal updates. One advantage of this approach is that, assuming the band is too large to fit into fast memory, Householder information is never written to slow memory: it is computed in fast memory, all updates are applied, and then the Householder entries are discarded.

In order to validate the communication pattern described in Algorithm 10.2, we verify three facts: $2\ell_i b_i$ columns of A fit in one third of fast memory, $\mathcal{H}$ fits in one third of fast memory, and each iteration in the OrthogonalUpdates function involves at most 3 blocks of Q, which fit in one third of fast memory. We will show that this is possible when $\omega_i \leq 2\sqrt{M}/(9(b_i+1))$, $\ell_i = (3/2)\omega_i$, $C = \sqrt{M}/3$, and the assumption from above that $b \leq \sqrt{M}/3$.

Since each column of the band has at most $(3/2)b_i+1$ entries, the total number of words in $2\ell_i b_i$ columns is $\omega_i((9/2)b_i^2 + 3b_i) < M/3$. The $\mathcal{H}$ data structure needs to store Householder information corresponding to chasing ω_i bulges ℓ_i times each, and each bulge consists of $c_i(d_i+1)$ entries. Thus $\mathcal{H}$ occupies $(3/8)\omega_i^2(b_i^2 + 2b_i) < M/3$ words. Finally, we must also verify that the number of columns of Q updated by the $\omega_i\ell_i$ bulge chases (which correspond to the rows of the band that are updated) cannot span more than 3 blocks of Q (*i.e.*, one third of fast memory). By Lemma 10.1, the number of columns is $(3/2)\omega_i(b_i+1) - b_i/2 \leq 2\sqrt{M}/3$; since $C = \sqrt{M}/3$, these columns cannot span more than 3 blocks.

Note that t is defined differently here than in Section 10.3.1.2. Here, since we will eliminate all subdiagonals at once, we need twice the working bandwidth to fit into fast memory.

10.3.2.3 Arithmetic Cost

From Lemma 1, the arithmetic cost of the orthogonal updates is given by $2n^3 \sum_{i=1}^{t} d_i/b_i$, where t is the number of sweeps, and the cost of the band reduction is always a lower order term. By the definition of t and the fact that $d_i = b_i/2$, the arithmetic cost is then $n^3 \min\{\log b, \lceil \log(2n(b+1)/M) \rceil\}$, ignoring lower order terms.

10.3.2.4 Bandwidth Cost

The bandwidth cost can be computed in a similar way to Section 10.3.1.2, though ω_i is defined slightly differently. The dominant communication cost is the call to the function OrthogonalUpdates within the innermost loop (lines 7-12). During the ith sweep, the number of sets of ω_i bulges is $n/(c_i\omega_i)$, and for each set, the innermost loop is executed $O(n/(\ell_i b_i))$ times. Since $\mathcal{H}$ resides in fast memory, the bandwidth cost of the function OrthogonalUpdates is that of reading and writing the row panels of the Q matrix: $O(nC)$ words. Thus, the total bandwidth cost of Algorithm 10.2 is

$$\sum_{i=0}^{t} O\left(\frac{n^3}{\sqrt{M}}\right) = O\left(\frac{tn^3}{\sqrt{M}}\right).$$

Note that due to the change in definition of ω_i, the bandwidth cost of the band reduction is increased from $O(n^2b^2/M)$ (from Section 10.3.1.2) to $O(n^2b/\sqrt{M})$, but this higher cost is still dominated by that of the orthogonal updates.

In the case that $\lceil \log(2n(b+1)/M) \rceil < \log b$, the final step of the algorithm is to read the entire band into memory and reduce all the remaining subdiagonals at once, updating Q

Algorithm 10.2 Sequential CASBR with orthogonal updates

```
Require: initial bandwidth $b \leq \sqrt{M}/3$ is a power of 2, $Q = I_n$ is stored in contiguous
    $C$-by-$C$ blocks, $\mathcal{H}$ is a temporary data structure of size $O(M)$ which resides in fast
    memory
 1: $t = \min\{\log b, \left\lceil \log \frac{2n(b+1)}{M} \right\rceil\}$
 2: for $i = 1$ to $t$ do
 3:     $b_i = \frac{b}{2^{i-1}}$, $c_i = \frac{b_i}{2}$, $d_i = \frac{b_i}{2}$, $\omega_i = 2\left\lfloor \frac{\sqrt{M}}{9(b_i+1)} \right\rfloor$, $\ell_i = \frac{3}{2}\omega_i$
 4:     while not reached end of band do
 5:         create next set of $\omega_i$ bulges , storing Householder entries in $\mathcal{H}$
 6:         OrthogonalUpdates($Q, \mathcal{H}$)
 7:         while not reached end of band do
 8:             write previous $\ell_i b_i$ columns of band
 9:             read next $\ell_i b_i$ columns of band
10:             chase $\omega_i$ bulges $\ell_i$ times each , storing Householder entries in $\mathcal{H}$
11:             OrthogonalUpdates($Q, \mathcal{H}$)
12:         end while
13:         chase $\omega_i$ bulges off the end of the band , storing Householder entries in $\mathcal{H}$
14:         OrthogonalUpdates($Q, \mathcal{H}$)
15:     end while
16:     copy band into data structure with column height $\frac{3}{2}b_{i+1}$
17: end for
18: if $t < \log b$ then
19:     read remaining band into fast memory
20:     reduce band to tridiagonal in one sweep, updating $Q$ with improved BLS algorithm
21:     write output to slow memory
22: end if

23: function OrthogonalUpdates($Q, \mathcal{H}$)
24:     for $i = 1$ to $\frac{n}{C}$ do
25:         read at most 3 blocks from $i$th block column of $Q$ into fast memory
26:         apply Householder updates stored in $\mathcal{H}$ to blocks of $Q$
27:         write blocks of $Q$ back to slow memory
28:     end for
29: end function
```

using the second technique of improving the BLS algorithm (*i.e.*, transforming the column-major H matrix to block-contiguous layout). In this case, the bandwidth cost of reading A is $O(nb)$, and the cost of the orthogonal updates is $O(n^3/\sqrt{M})$ as explained above. Both of these are lower order terms.

10.3.2.5 Latency Cost

The latency cost is also dominated by that of the orthogonal updates. Since Q is stored in C-by-C contiguous blocks, the latency cost of the function OrthogonalUpdates is $O(n/C)$. Thus, the latency cost of Algorithm 10.2 simplifies to $O(tn^3/M^{3/2})$.

Like the bandwidth cost, the latency cost associated with the band reduction is increased by the choice of ω_i, but this higher cost of $O(tn^2/M)$ is still dominated by that of the orthogonal updates. In the case that $\lceil \log(nb/M) \rceil < \log b$, the final step of the algorithm using the improved BLS technique incurs a latency cost which is also a lower order term.

10.4 Parallel Band Tridiagonalization Algorithms

Recall our distributed-memory parallel model described in Section 2.2.2, where we have P processors connected over a network. Again, we will first discuss the case of computing eigenvalues only and then extend to the case of computing both eigenvalues and eigenvectors. The main improvement of our new algorithm over previous approaches is a reduction in latency cost, both in terms of the band reduction and the back-transformation phase (when eigenvectors are desired).

We assume that $b \leq n/(3P)$, where P is the number of processors involved in the band reduction. This is a reasonable assumption in the context of two-step tridiagonalization, in order to minimize the latency cost in the first step. For larger bandwidths, one may use fewer processors on the first sweep(s), or have multiple processors participate in a single bulge chase. The latter approach may incur a higher communication cost—see [102].

10.4.1 Computing Eigenvalues Only

In this section we concern ourselves with the case when only eigenvalues are desired, so the orthogonal updates may be discarded after applying them to the band. We collect the results from the analyses in Sections 10.4.1.1-10.4.1.2 in Table 10.3.

10.4.1.1 Alternate Approaches

The ‘conventional’ distributed memory band tridiagonalization algorithm was introduced in [102], and has been extended several times. This is a parallelization of the MH algorithm, discussed in Section 10.3.1.1, a one-sweep band reduction algorithm (*i.e.*, $d = b - 1$ and $c = 1$). We will refer to this as Lang’s algorithm.

Algorithm	**Flops**	**Words**	**Messages**
Lang [10, 102]	$O\left(\frac{n^2b}{P}\right)$	$O(nb)$	$O(n)$
CASBR	$O\left(\frac{n^2b}{P}\right)$	$O(nb)$	$O(P\log b)$

Table 10.3: Asymptotic comparison of previous parallel algorithms for tridiagonalization (for eigenvalues only) with our improvements, for symmetric band matrices of n columns and $b+1$ subdiagonals on a machine with P processors. The first row assumes that $P \leq n/b$, and the second row assumes $P \leq n/(3b)$. The asymptotic arithmetic and communication costs are determined along the critical path.

We will not present this algorithm and its variants, but instead refer the reader to the detailed complexity analysis (and performance modeling) in [10] (summarized in the papers [11] and [12]). We present their complexity results in asymptotic notation; the hidden constant factors vary depending on the optimizations applied, including 'logical blocking,' which eliminates a factor of 2 idle time along the critical path, and using a cyclic layout, which helps alleviate load imbalance between processors. Along the critical path, their algorithm performs $O(n^2b/P)$ flops and moves $O(nb)$ words. Because there is a communication step for every column in the band, the latency cost is $O(n)$ messages.

Unless multiple bulges are chased at a time, the latency cost of $O(n)$ cannot be asymptotically reduced. That is, if a message is sent along the critical path for every parallelogram annihilated, then the last sweep, which has one parallelogram for each column, will incur $O(n)$ latency cost.

10.4.1.2 CASBR

The parallel CASBR algorithm begins with a similar data layout as Lang's algorithm. Each of the P processors (indexed 0 to $P-1$) owns a contiguous set of $C = n/P$ columns of the lower half of the symmetric band. We use a similar successive halving and multiple bulge chasing approach to the sequential CASBR algorithm. During each sweep, the number of columns per processor stays fixed at $C = n/P$. We assume each of the P processors has $\Omega(nb/P)$ words of memory available, so that the band A can be stored across the machine. To simplify the presentation, we assume that $3bP$ divides n, and that b is a power of 2. This implies that $P \leq n/(3b)$, which is our maximum parallelism. Note that our maximum parallelism is three times smaller than the Lang approach; we conjecture that this constant factor can be improved by exploiting more overlap between the pipeline stages in the bulge chasing procedure. These constant factors will not affect our asymptotic analysis.

Roughly, the parallel algorithm proceeds as each processor chases bulges through its C (local) columns and into the C columns of its right neighbor, and then passes the second set of columns to its right neighbor. This way, each of the P processors accesses only $O(nb/P)$ of A rather than streaming through the entire band. In the algorithm we present below, each processor is active on every other step; we can eliminate this idle time by using logical blocking (as in [102]); we ignore this factor of 2 savings for the purposes of our asymptotic analysis.

At the high level, there are four kernels: create_bulges, pass_bulges, clear_bulges, and create_and_clear_bulges. The create_bulges kernel eliminates ω_i parallelograms (each with c_i columns and d_i diagonals) from the local set of C columns of A and chases the resulting bulges ℓ_i times (on average[3]) into the right neighbor's set of C columns. The pass_bulges kernel chases ω_i bulges (created by the left neighbor) from the local set ℓ_i times into the right neighbor's set. The create_and_clear_bulges and clear_bulges kernels are only executed by the last processor[4] and are analogous to create_bulges and pass_bulges, except the 'second set of columns' is off the end of the band. Both create_bulges and pass_bulges require $2C$ columns to pass information from one processor's columns to the next: the left set of C columns is owned by the processor invoking the kernel, and the right set is owned by the right neighbor. The create_and_clear_bulges and clear_bulges kernels require only the last C columns of the band (its local set).

At any time, a processor will have access to and update only its own C columns and the C columns from its right neighbor. For example, the parallel algorithm begins with processor 1 sending its columns to processor 0. After processor 0 executes the create_bulges kernel, it sends the updated second set of C columns (with bulges) back to processor 1. Processor 1 must then also receive processor 2's C columns in order to execute the pass_bulges kernel. The parallel algorithm ends (on sweep $i = \log b$) with processor $P - 1$ receiving C columns from the left, clearing all bulges, and finally eliminating the last subdiagonal of its local block (via create_and_clear_bulges).

In order for the pass_bulges kernel to pass the bulges into the right neighbor's column block, and for the bulges to retain their respective positions relative to the column blocks, we set $\ell_i = C/b_i$, which is an integer given the assumptions above. Recall that a bulge chase advances a bulge exactly b_i columns.

Our constraint on ω_i, the maximum number of bulges that fits in $C = n/P$ columns, is given by Lemma 10.2:

$$(\omega_i - 1)(2b_i - c_i) + c_i + d_i \leq C.$$

A little more care must be taken when creating bulges to ensure that they do not cross processor boundaries (adjacent sets of C columns). Consulting Lemma 10.1, for the successive halving approach, we arrive at the following lemma.

Lemma 10.8. *Assuming b and ω are even, $c = d = b/2$, and $3b$ divides C, then we can create and chase $\omega = 2C/(3b)$ bulges at a time, and chasing them $\ell = C/b$ times each advances them to the next set of C columns.*

[3]Note that some bulges may need to be chased up to $2\ell_i$ times, some less.

[4]Note that processor $P-2$ may chase some bulges (partially or completely) off the end of the band when invoking pass_bulges and create_bulges, depending on the number of columns owned by processor $P-1$ and the current bandwidth.

As in the sequential case, we fix the parameters to simplify the asymptotic analysis; in practice, the parameters (including the number of processors $P' \le P$ used and the number of columns C a processor owns) should be tuned independently.

Algorithm 10.3 Parallel CASBR

```
Require: $3bP$ divides $n$, $b$ is a power of 2, processor ranks are between 0 and $P-1$, each
    processor owns $C = \frac{n}{P}$ columns of $A$.
 1: for $i = 1$ to $\log b$ do
 2:     $b_i = \frac{b}{2^{i-1}}$, $c_i = \frac{b_i}{2}$, $d_i = \frac{b_i}{2}$, $\omega_i = \frac{2C}{3b_i}$, $\ell_i = \frac{3}{2}\omega_i$.
 3:     if myrank $> 0$ then
 4:         send left: block of $C$ columns
 5:     end if
 6:     for $j = 1$ to $3 \cdot$ myrank do
 7:         receive from left: block of $C$ columns (includes bulges)
 8:         if myrank $= P-1$ then
 9:             clear_bulges
10:         else
11:             receive right: block of $C$ columns
12:             pass_bulges
13:             send right: block of $C$ columns (includes bulges)
14:         end if
15:         if $j < 3 \cdot$ myrank then
16:             send left: block of $C$ columns
17:         end if
18:     end for
19:     for $j = 1$ to $3$ do
20:         if myrank $= P-1$ then
21:             create_and_clear_bulges
22:         else
23:             receive right: block of $C$ columns
24:             create_bulges
25:             send right: block of $C$ columns (includes bulges)
26:         end if
27:     end for
28: end for
```

We analyze the arithmetic, bandwidth, and latency costs along the critical path of the algorithm. That is, we follow the progress of the first ω_1 bulges from processor 0 to processor $P-2$, at which point (exactly) one of processors $P-2$ and $P-1$ is active chasing and/or clearing bulges on every remaining step of every sweep.

10.4.1.3 Arithmetic Cost

From Lemma 10.5, the arithmetic cost of chasing one bulge (a single hop), with parameters b, c, and d, is bounded above by $8bcd + 4cd^2 + O(bc)$ flops, while the cost of creating a bulge and the cost of chasing a bulge partially or completely off the band are less. For our choices $b/2^i = b_i/2 = c_i = d_i$, this cost is $(5/2)b_i^3$ flops.

Every kernel call involves at most ω_i bulges; the calls to create_bulges, pass_bulges, clear_bulges, and create_and_clear_bulges costs each chase the bulges about ℓ_i times, so each kernel invocation costs about $\omega_i \ell_i (5b_i^3/2) = O(n^2 b_i/P^2)$ flops. Following the critical path, there are (fewer than) P kernel invocations while the pipeline fills. At this point, processors $P-2$ and $P-1$ are active for the remainder of the execution, each invoking a kernel on alternating steps. There are $3p$ steps (iterations of the inner two for-loops) per sweep, each with one kernel invocation (along the critical path). Altogether, this is

$$O\left(\frac{n^2 b_1}{P}\right) + \sum_{i=1}^{\log b} O\left(\frac{n^2 b_i}{P}\right) = O\left(\frac{n^2 b}{P}\right)$$

flops. The hidden leading constant is about 20; a cyclic layout and logical blocking as in [102] can be applied here to reduce this constant to between 5 and 10 (note these same strategies reduced the corresponding constant in Lang's algorithm's arithmetic cost from 24 to between 6 and 12).

10.4.1.4 Bandwidth Cost

Every message in the algorithm consists of C columns of the band; because of bulges and triangular fill stored below the b_ith subdiagonal, each message (during the ith sweep) has size (at most) $C(3b_i/2+1) = O(nb_i/P)$ words. Following the critical path as before, we have the upper bound of

$$O\left(nb_1\right) + \sum_{i=1}^{\log b} O\left(nb_i\right) = O\left(nb\right)$$

words moved.

10.4.1.5 Latency Cost

The latency cost analysis is similar to the bandwidth cost analysis, replacing the $O(nb_i)$ terms by $O(1)$; in total, we have $O(P \log b)$ messages. This is asymptotically smaller than the $O(n)$ messages that Lang's algorithm sends: we save a factor of $O(n/(P \log b))$ messages.

10.4.2 Computing Eigenvalues and Eigenvectors

Recall our three steps: first, tridiagonalize $A = QTQ^T$; second, compute the eigendecomposition $T = V\Lambda V^T$ with an efficient algorithm; finally, back-transform the matrix V by

Algorithm	**Flops**	**Words**	**Messages**
Lang [10]	$O\left(\frac{n^2 b}{\sqrt{P}} + \frac{n^3}{P}\right)$	$O\left(nb + \frac{n^2}{\sqrt{P}}\right)$	$O\left(n + \frac{n}{b}\right)$
CASBR	$O\left(\frac{n^2 b}{\sqrt{P}} + \frac{n^3}{P}\log b\right)$	$O\left(nb + \frac{n^2}{\sqrt{P}}\log b\right)$	$O(\sqrt{P}\log b + \sqrt{P}\log b)$

Table 10.4: Asymptotic comparison of previous parallel algorithms for tridiagonalization (for eigenvalues and eigenvectors) with our improvements, for symmetric band matrices of n columns and $b+1$ subdiagonals on a machine with P processors. We assume only $O(\sqrt{P})$ of the processors participate in the band reduction for both algorithms. We include the cost of the back transformation (but not the cost of the tridiagonal eigendecomposition). The first row assumes $\sqrt{P} \leq n/b$, and the second row assumes $\sqrt{P} \leq n/(3b)$. The asymptotic arithmetic and communication costs are determined along the critical path. The two terms in each cost correspond to the band reduction and the back transformation, respectively.

computing QV. We may either store Q implicitly as a collection of Householder vectors, and apply it using a blocked approach, or compute Q explicitly by applying the orthogonal updates (from the band reduction) to an identity matrix, and then compute QV with a matrix multiplication. As in the sequential case, the computation and communication involved in constructing and/or applying Q dominates the costs of the band reduction.

We assume V is distributed in a 2D blocked fashion to all P processors, and that the bandwidth b of A is (at most) 1/3 of the width of a block row of V, *i.e.*, $b \leq n/(3\sqrt{P})$. This means that we will use only $\sqrt{P}$ of the P available processors to perform the band reduction, and all P for the back-transformation. So, we must assume each processor has $\Omega(n^2/P)$ words of memory.

We collect the results from the analyses in Sections 10.4.2.1-10.4.2.2 in Table 10.4. Under our assumptions, for both algorithms, the arithmetic and bandwidth costs of the back-transformation always dominate those of the band reduction. The asymptotic arithmetic costs decrease linearly (in P) as expected. The first step of two-step tridiagonalization can attain the communication lower bounds for parallel dense linear algebra (without extra memory), *i.e.*, $\Omega(n^2/\sqrt{P})$ words moved and $\Omega(\sqrt{P})$ messages, if $b = \Theta(n/\sqrt{P})$. Asymptotically, both algorithms attain the bandwidth lower bound, up to a factor of $\Theta(\log(n/\sqrt{P}))$ in the case of CASBR. However, only CASBR attains the latency lower bound of $\Omega(\sqrt{P})$, again up to a factor of $\Theta(\log(n/\sqrt{P}))$.

10.4.2.1 Alternate Approaches

The approach in [11] stores Q implicitly (as a sequence of Householder transformations) and then applies Q to V in a blocked fashion. The authors give three algorithms to compute QV, with different parallel layouts of the matrix V—we consider only their best approach, based on a 2D layout which is dynamically rebalanced. Before computing QV, we assume each processor owns a $(n/\sqrt{P})$-by-$(n/\sqrt{P})$ block of V. Again, we refer the reader to the detailed analysis in [10]. Along the critical path, the additional costs for the back-transformation are $O(n^3/P)$ flops, $O(n^2/\sqrt{P})$ words moved, and $O(n/b)$ messages.

10.4.2.2 CASBR

As in the sequential case (Section 10.3.2.2), we construct Q explicitly rather than storing it implicitly. The extra cost of the matrix multiplication QV is dominated by the cost of constructing Q and thus will not affect our asymptotic analysis. Again, in practice, this cost can be avoided by storing and applying Q to V as a sequence of Householder transformations.

By the assumption $\sqrt{P} \leq n/(3b)$, we can involve all $\sqrt{P}$ processors in each processor row in a band reduction. Since the arithmetic cost for the band reduction is a lower order term, we can afford to perform the band reduction $\sqrt{P}$ times redundantly (or once, but only on a subset of $\sqrt{P}$ processors). We distribute the band A to each row of the given $\sqrt{P}$-by-$\sqrt{P}$ processor grid; each row performs the band reduction once. Note that each processor owns $C = n/\sqrt{P}$ columns of A, rather than n/P (as before).

We use Algorithm 10.4, a modification of Algorithm 10.3, which simultaneously computes the band reduction and the n-by-n matrix Q. That is, we postmultiply an n-by-n identity matrix I_n by each orthogonal matrix $Q_1, Q_2, \ldots$, generated by the bulge chasing procedure. (To simplify the presentation, we will again refer to the intermediate products $I_n \cdot Q_1 \cdot Q_2 \cdots$ also as Q, and the intermediate band matrices all as A.) These orthogonal updates combine columns of Q (but not rows); thus, each processor row may work independently. Each processor row is assigned C contiguous rows of Q; the columns of this block row are distributed according to the distribution of the band matrix. That is, if processor i (indexed within a given processor row) owns the first element of the jth row of A, then processor i will own the jth column of the corresponding block row of Q. In this way, the communication pattern of the blocks of Q between neighboring processors will exactly match the communication pattern of the blocks of the band. Whenever a processor performs a local kernel on $2C$ columns of the band, it will also apply all of those updates to $2C$ columns of (its block row of) Q. This implies that in sweep i, within each processor row, the first processor owns the first $C+b_i$ columns of the corresponding block row of Q, each subsequent processor owns the next C columns, and the last processor owns the last $C-b_i$ columns. (Note that the first processor does not touch the first $b_i/2$ columns, but rather stores them to be updated in the next sweep.) This distribution also implies that between sweeps i and $i+1$, the Q matrix must be shifted to maintain the relationship between the ownership of rows of the band and the columns of Q. To simplify the presentation, we assume that on each sweep i, Q is padded with b_i zero columns, and that the first processor in each row always sends its rightmost C columns; under these assumptions, each processor always sends/receives C-by-C blocks of Q, avoiding fringe cases for the first and last processors (within each processor row).

For the orthogonal updates of Q, we introduce four new kernels—create_bulges_update, pass_bulges_update, create_and_clear_bulges_update, and clear_bulges_update—which apply the *right* orthogonal updates (as sets of Householder transformations) from the corresponding band reduction kernels to the local blocks of Q.

Again, we do not analyze computing the eigendecomposition of T, but we assume that this step terminates with V distributed across the processor grid with each processor owning a C-by-C block of V. We then compute QV using matrix-matrix multiplication.

In the following complexity analysis, we count only the additional work and communication done for the orthogonal updates. To obtain the results for CASBR in Table 10.4, we simply add the the band reduction costs (Section 10.4.1.2), substituting $\sqrt{p}$ for p (since now we run the band reduction redundantly). Then we add the cost of multiplying QV with Cannon's algorithm [49], which costs $2n^3/p$ flops, $O(n^2/\sqrt{p})$ words moved, and $O(\sqrt{p})$ messages. These are all lower order terms, due to the logarithmic factors in the other costs.

10.4.2.3 Arithmetic Cost

As argued in Section 10.4.1.2, there are at most $\omega_i \ell_i$ bulges chased in the pass_bulges, clear_bulges, and create_and_clear_bulges kernels, and at most $2\omega_i \ell_i$ bulges chased in the create_bulges kernel. Since the number of Householder entries in each bulge chase is $c_i d_i = b_i^2/4$, from Lemma 10.3, the cost of applying the updates from one kernel invocation to $n/\sqrt{P}$ rows of the Q matrix is at most

$$4 \cdot \frac{b_i^2}{4} \cdot \frac{n}{\sqrt{P}} \cdot \omega_i \ell_i = \frac{2n^3}{3P^{3/2}} = O\left(\frac{n^3}{P^{3/2}}\right)$$

flops (and up to 2 times more for create_bulges).

Following the analysis in Section 10.4.1.2, we can upper bound the additional arithmetic performed along the critical path by

$$O\left(\frac{n^3}{P}\right) + \sum_{i=1}^{\log b} O\left(\frac{n^3}{P}\right) = O\left(\frac{n^3 \log b}{P}\right)$$

flops. The costs of the band reduction and multiplication QV are lower order terms.

10.4.2.4 Bandwidth Cost

The communication costs of the orthogonal updates are also analogous to band reduction. As shown in Algorithm 10.4, for every message sent/received containing a block of A, there is a second message containing a block of Q. (The additional message every sweep to shift the block row of Q amounts to a lower order term.) However, while the size of the A messages decreases with the bandwidth, the size of the Q messages remains the same (n^2/P words). The additional bandwidth cost, following the analysis in Section 10.4.1.2, is bounded by

$$O\left(\frac{n^2}{\sqrt{P}}\right) + \sum_{i=1}^{\log b} O\left(\frac{n^2}{\sqrt{P}}\right) = O\left(\frac{n^2 \log b}{\sqrt{P}}\right)$$

words moved. Again, the cost of the band reduction and multiplication QV are lower order terms.

Algorithm 10.4 Parallel SBR with orthogonal updates

```
Require: $3b\sqrt{P}$ divides $n$, $b$ is a power of 2. Processor ranks are with respect to the processor row (i.e., between 0 and $\sqrt{P}-1$). Within each processor row, each processor stores $C = \frac{n}{\sqrt{P}}$ columns of $A$, and $C$-by-$C$ (or $C$-by-$(C \pm b_i)$) block of $Q$, whose column indices correspond to the indices of the local rows of $A$ whose first (leftmost) nonzero is stored locally.
 1: for $i = 1$ to $\log b$ do
 2:     $b_i = \frac{b}{2^{i-1}}$, $c_i = \frac{b_i}{2}$, $d_i = \frac{b_i}{2}$, $\omega_i = \frac{2C}{3b_i}$, $\ell_i = \frac{3}{2}\omega_i$.
 3:     if myrank $> 0$ then
 4:         send left: block of $C$ columns of $A$
 5:         send left: block of $C$ columns and rows of $Q$
 6:     end if
 7:     for $j = 1$ to $3 \cdot$ myrank do
 8:         receive from left: block of $C$ columns of $A$ (includes bulges)
 9:         receive from left: block of $C$ columns and rows of $Q$
10:         if myrank $= \sqrt{P}-1$ then
11:             clear_bulges
12:             clear_bulges_update
13:         else
14:             receive from right: block of $C$ columns of $A$
15:             receive from right: block of $C$ columns and rows of $Q$
16:             pass_bulges
17:             pass_bulges_update
18:             send right: block of $C$ columns of $A$ (includes bulges)
19:             send right: block of $C$ columns and rows of $Q$
20:         end if
21:         if $j < 3 \cdot$ myrank then
22:             send left: block of $C$ columns of $A$
23:             send left: block of $C$ columns and rows of $Q$
24:         end if
25:     end for
26:     for $q = 1$ to 3 do
27:         if myrank $= \sqrt{P}-1$ then
28:             create_and_clear_bulges
29:             create_and_clear_bulges_update
30:         else
31:             receive from right: block of $C$ columns of $A$
32:             receive from right: block of $C$ columns and $C$ rows of $Q$
33:             create_bulges
34:             create_bulges_update
35:             send right: block of $C$ columns of $A$ (includes bulges)
36:             send right: block of $C$ columns and rows of $Q$
37:         end if
38:     end for
39:     if myrank $< \sqrt{P}-1$ then
40:         send right: block of $b_i/2$ columns and $C$ rows of $Q$.
41:     else if myrank $> 0$ then
42:         receive left: block of $b_i/2$ columns and $C$ rows of $Q$.
43:     end if
44: end for
```

10.4.2.5 Latency Cost

The additional latency cost is the same as that for the band reduction (see Section 10.4.1.2) plus the shift (a lower order term), *i.e.*, $O(\sqrt{P}\log b)$ messages. In the more restrictive case $\sqrt{P} \ll n/(b\log b)$, this is an asymptotic improvement compared to Lang's algorithm for just the back-transformation phase; considering also the cost of the band reduction, we always have an asymptotic improvement.

10.5 Conclusions

In theory, both band reduction and dense matrix-matrix multiplication have $O(n)$ possible data reuse in the sequential case, given by the ratio of total flops to size of inputs and outputs. When the problem does not fit in fast memory (of size M words), matrix multiplication can attain only $O(\sqrt{M})$ data reuse [88], while our CASBR algorithm achieves $O(M/b)$ reuse, provided $b \le \sqrt{M}/3$. This constraint on b also ensures that the reuse is always asymptotically at least as large as that of matrix multiplication, and when $b \ll \sqrt{M}$, we can actually attain much better reuse.

Indeed, improved data reuse often translates to better performance. In [31], we observed that using the techniques of reducing communication (even at the expense of some extra arithmetic), as well as a framework that automatically tuned the algorithmic parameters, led to speedups of $2-6\times$ on sequential and shared-memory parallel machines. We believe that these benefits will extend to the distributed-memory case, particularly when performance is latency-bound.

The performance results in [31] focused on the case of computing eigenvalues only and did not include the cost of the back-transformation phase. In that case, the arithmetic cost increased by no more than 50%. As we have seen, the cost of the back-transformation, which dominates that of the band reduction when eigenvectors are requested, increases with the number of sweeps. For example, for the successive halving approach, the increase in arithmetic was a factor of $O(\log b)$. Thus, there exists an important tradeoff between reducing communication in the band reduction phase and the resulting increased costs in the back-transformation phase. Note that when computing partial eigensystems, the costs of the back-transformation can be reduced to be proportional to the number of eigenvectors desired, improving this tradeoff.

W also do not give algorithms or complexity analysis for taking more than 1 and less than $\log b$ sweeps and using the technique of chasing multiple bulges. Indeed, we fixed many parameters here with the sole intention of simplifying the theoretical analysis. In practice, parameters such as the number of sweeps and the number of bulges chased at a time should be autotuned for the target architecture to navigate the tradeoffs mentioned above.

Recall our application of two-step tridiagonalization for the symmetric eigenproblem. The first step (full-to-banded) and its corresponding back-transformation phase, can be performed efficiently [17, 109]. Combined with the approaches here for the second step (and an efficient tridiagonal eigensolver), we have sequential and parallel algorithms for the symmetric eigenproblem that attain the communication lower bounds for dense linear algebra in [28] up to $O(\log b)$ factors: in the sequential case, $\Omega(n^3/\sqrt{M})$ words moved and $\Omega(n^3/M^{3/2})$ messages, in the parallel case (if minimal memory is used), $\Omega(n^2/\sqrt{P})$ words moved and $\Omega(\sqrt{P})$ messages. Even though these lower bounds formally apply to only the first step, they are still valid lower bounds for any algorithm that performs this step.

We also remark that, in the sequential case, similar techniques to those used in a communication-optimal first step can also be applied in the case $b > \sqrt{M}/3$ (violating an assumption in Section 10.3). In fact, for any $b+1 \leq n$, we can reduce to tridiagonal form with communication costs that attain (or beat) the aforementioned bounds.

Chapter 11

Communication-Avoiding Parallel Strassen

In this chapter, we consider the parallelization of Strassen's fast matrix multiplication algorithm. Our main contribution is a new algorithm we call Communication-Avoiding Parallel Strassen, or *CAPS*, that

- perfectly load balances the $\Theta(n^{\lg 7})$ flops across processors,
- is communication optimal, attaining the bandwidth and latency cost lower bounds of Chapter 5 and Section 6.2.1.2 up to a logarithmic factor in the number of processors,
- requires asymptotically less communication than previous parallelizations of Strassen's algorithm,
- requires asymptotically less computation *and* communication than all classical algorithms, and
- outperforms in practice all other known implementations of matrix multiplication, Strassen-based or classical.

The algorithm and its computational and communication cost analyses are presented in Section 11.2. There we show it matches the communication lower bounds. We provide a review and analysis of previous algorithms in Section 11.3. We also consider two natural combinations of previously known algorithms (Sections 11.3.4 and 11.3.5). One of these new algorithms that we call "2.5D-Strassen" performs better than all previous algorithms, but is still not optimal, and is inferior to CAPS.

We discuss our implementations of the new algorithms and compare their performance with previous ones in Section 11.4 to show that our new CAPS algorithm outperforms previous algorithms not just asymptotically, but also in practice. Benchmarking our implementation on a Cray XT4, we obtain speedups over classical and Strassen-based algorithms ranging from 24% to 184% for a fixed matrix dimension $n = 94080$, where the number of nodes ranges from 49 to 7203.

In Section 11.5 we show that our parallelization method applies to other fast matrix multiplication algorithms. It also applies to classical recursive matrix multiplication, thus obtaining a new optimal classical algorithm that matches the communication complexity of the 2.5D algorithm of Solomonik and Demmel [137]. In Section 11.5, we also discuss numerical stability, hardware scaling, and future work.

The main results of this chapter appear in [20], written with coauthors James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. A subsequent paper [105], written with James Demmel, Benjamin Lipshitz, and Oded Schwartz, details the implementation of the algorithm on multiple machines and presents more extensive performance data. Much of that content also appears in [106].

11.1 Preliminaries

11.1.1 Strassen's Algorithm

Strassen showed that 2×2 matrix multiplication can be performed using 7 multiplications and 18 additions, instead of the classical algorithm that does 8 multiplications and 4 additions [139]. By recursive application this yields an algorithm with multiplies two $n \times n$ matrices $O(n^{\omega_0})$ flops, where $\omega_0 = \log_2 7 \approx 2.81$ (see Section 2.4.1). Winograd improved the algorithm to use 7 multiplications and 15 additions in the base case, thus decreasing the hidden constant in the O notation [152]. Our implementation uses the Winograd variant (see Section 2.4.2 for details).

11.1.2 Previous Work on Parallel Strassen

In this section we briefly describe previous efforts to parallelize Strassen. More details, including communication analyses, are in Section 11.3. A summary appears in Table 11.1.

Luo and Drake [108] explored Strassen-based parallel algorithms that use the communication patterns known for classical matrix multiplication. They considered using a classical 2D parallel algorithm and using Strassen locally, which corresponds to what we call the "2D-Strassen" approach (see Section 11.3.2). They also consider using Strassen at the highest levels and performing a classical parallel algorithm for each subproblem generated, which corresponds to what we call the "Strassen-2D" approach. The size of the subproblems depends on the number of Strassen steps taken (see Section 11.3.3). Luo and Drake also analyzed the communication costs for these two approaches.

Soon after, Grayson, Shah, and van de Geijn [78] improved on the Strassen-2D approach of [108] by using a better classical parallel matrix multiplication algorithm and running on a more communication-efficient machine. They obtained better performance results compared to a purely classical algorithm for up to three levels of Strassen's recursion.

Kumar, Huang, Johnson, and Sadayappan [101] implemented Strassen's algorithm on a shared-memory machine. They identified the tradeoff between available parallelism and total memory footprint by differentiating between "partial" and "complete" evaluation of the algorithm, which corresponds to what we call depth-first and breadth-first traversal of the recursion tree (see Section 11.2.1). They show that by using ℓ DFS steps before using BFS steps, the memory footprint is reduced by a factor of $(7/4)^\ell$ compared to using all BFS steps. They did not consider communication costs in their work.

Other parallel approaches [64, 90, 138] have used more complex parallel schemes and communication patterns. However, they restrict attention to only one or two steps of Strassen and obtain modest performance improvements over classical algorithms.

11.1.3 Lower Bounds for Strassen's Algorithm

The main impetus for this work was the observation of the asymptotic gap between the communication costs of existing parallel Strassen-based algorithms and the communication lower bounds given by Theorems 5.10 and 6.8. Because of the attainability of the lower bounds in the sequential case, we hypothesized that the gap could be closed by finding a new algorithm rather than by tightening the lower bounds.

We made three observations from the lower bound results of Chapter 5 that led to the new algorithm. First, the lower bounds for Strassen are lower than those for classical matrix multiplication. This implies that in order to obtain an optimal Strassen-based algorithm, the communication pattern for an optimal algorithm cannot be that of a classical algorithm but must reflect the properties of Strassen's algorithm. Second, the factor $M^{\omega_0/2-1}$ that appears in the denominator of the communication cost lower bound implies that an optimal algorithm must use as much local memory as possible. That is, there is a tradeoff between memory usage and communication (the same is true in the classical case). Third, the proof of the lower bounds shows that in order to minimize communication costs relative to computation, it is necessary to perform each submatrix multiplication of size $\Theta(\sqrt{M}) \times \Theta(\sqrt{M})$ on a single processor.

With these observations and assisted by techniques from previous approaches to parallelizing Strassen, we developed a new parallel algorithm which achieves perfect load balance, minimizes communication costs, and in particular performs asymptotically less computation and communication than is possible using classical matrix multiplication.

11.2 The Algorithm

In this section we present the CAPS algorithm, and prove it is communication optimal. See Algorithm 11.1 for a concise presentation and Algorithm 11.2 for a more detailed description. We use notation for the Strassen-Winograd algorithm from Section 2.4.2.

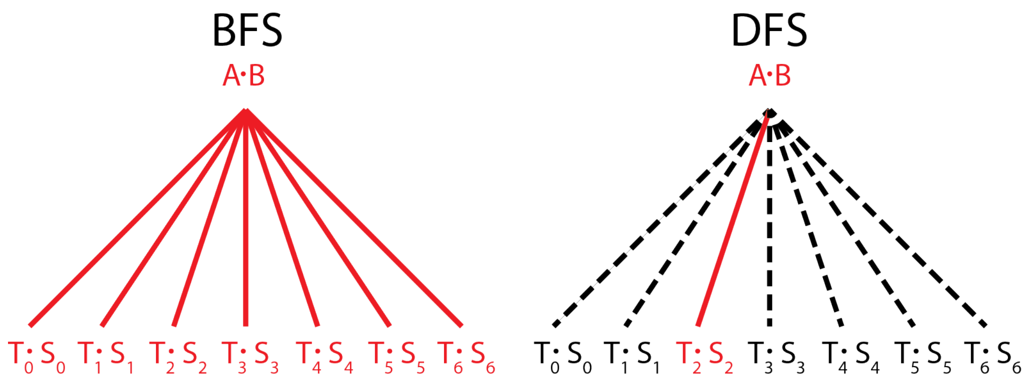

BFS
A·B
T0 S0 T1 S1 T2 S2 T3 S3 T4 S4 T5 S5 T6 S6
DFS
A·B
T0 S0 T1 S1 T2 S2 T3 S3 T4 S4 T5 S5 T6 S6

Figure 11.1: Representation of BFS and DFS steps. In a BFS step, all seven subproblems are computed at once, each on 1/7 of the processors. In a DFS step, the seven subproblems are computed in sequence, each using all the processors. The notation follows that of Section 2.4.2.

11.2.1 Overview of CAPS

Consider the recursion tree of Strassen's sequential algorithm. CAPS traverses it in parallel as follows. At each level of the tree, the algorithm proceeds in one of two ways. A "breadth-first-step" *(BFS)* divides the 7 subproblems among the processors, so that $\frac{1}{7}$ of the processors work on each subproblem independently and in parallel. A "depth-first-step" *(DFS)* uses all the processors on each subproblem, solving each one in sequence. See Figure 11.1.

In short, a BFS step requires more memory but reduces communication costs while a DFS step requires little extra memory but is less communication-efficient. In order to minimize communication costs, the algorithm must choose an ordering of BFS and DFS steps that uses as much memory as possible.

Let $k = \log_7 P$ and $s \geq k$ be the number of distributed Strassen steps the algorithm will take. In this section, we assume that n is a multiple of $2^s 7^{\lceil k/2 \rceil}$. If k is even, the restriction simplifies to n being a multiple of $2^s\sqrt{P}$. Since P is a power of 7, it is sometimes convenient to think of the processors as numbered in base 7. CAPS performs s steps of Strassen's algorithm and finishes the calculation with local matrix multiplication. The algorithm can easily be generalized to other values of n by padding or dynamic peeling.

We consider two simple schemes of traversing the recursion tree with BFS and DFS steps. The first scheme, which we call the Unlimited Memory *(UM)* scheme, is to take k BFS steps in a row. This approach is possible only if there is sufficient available memory. The second scheme, which we call the Limited Memory *(LM)* scheme is to take ℓ DFS steps in a row followed by k BFS steps in a row, where ℓ is minimized subject to the memory constraints.

It is possible to use a more complicated scheme that interleave BFS and DFS steps to reduce communication. We show that the LM scheme is optimal up to a constant factor, and hence no more than a constant factor improvement can be attained from interleaving.

Algorithm 11.1 CAPS, in brief. For more details, see Algorithm 11.2.

```
Require: $A$, $B$, $n$, where $A$ and $B$ are $n \times n$ matrices
        $P$ = number of processors
Ensure: $C = A \cdot B$
            ▷ The dependence of the $S_i$'s on $A$, the $T_i$'s on $B$ and $C$ on the $Q_i$'s follows the
    Strassen-Winograd algorithm. See Section 2.4.2.
 1: procedure C = CAPS($A$, $B$, $n$, $P$)
 2:     if enough memory then                                        ▷ Do a BFS step
 3:         locally compute the $S_i$'s and $T_i$'s from $A$ and $B$
 4:         while $i = 1 \ldots 7$ do
 5:             redistribute $S_i$ and $T_i$
 6:             $Q_i$ = CAPS($S_i$, $T_i$, $n/2$, $P/7$)
 7:             redistribute $Q_i$
 8:         end while
 9:         locally compute $C$ from all the $Q_i$'s
10:     else                                                         ▷ Do a DFS step
11:         for $i = 1 \ldots 7$ do
12:             locally compute $S_i$ and $T_i$ from A and B
13:             $Q_i$ = CAPS($S_i$, $T_i$, $n/2$, $P$)
14:             locally compute contribution of $Q_i$ to $C$
15:         end for
16:     end if
17: end procedure
```

11.2.2 Data Layout

We require that the data layout of the matrices satisfies the following two properties:

1. At each of the s Strassen recursion steps, the data layouts of the four submatrices of each of A, B, and C must match so that the weighted additions of these submatrices can be performed locally. This technique follows [108] and allows communication-free DFS steps.

2. Each of these submatrices must be equally distributed among the P processors for load balancing.

There are many data layouts that satisfy these properties, perhaps the simplest being block-cyclic layout with a processor grid of size $7^{\lfloor k/2 \rfloor} \times 7^{\lceil k/2 \rceil}$ and block size $\frac{n}{2^s 7^{\lfloor k/2 \rfloor}} \times \frac{n}{2^s 7^{\lceil k/2 \rceil}}$. (When $k = \log_7 P$ is even these expressions simplify to a processor grid of size $\sqrt{P} \times \sqrt{P}$ and block size $\frac{n}{2^s \sqrt{P}}$.) See Section 2.3.2 for a description of block-cyclic layout and Figure 2.3 for an example with $P = 49$.

Any layout that we use is specified by three parameters, (n, P, s), and intermediate stages of the computation use the same layout with smaller values of the parameters. A BFS step reduces a multiplication problem with layout parameters (n, P, s) to seven subproblems with layout parameters $(n/2, P/7, s-1)$. A DFS step reduces a multiplication problem with layout parameters (n, P, s) to seven subproblems with layout parameters $(n/2, P, s-1)$.

Note that if the input data is initially load-balanced but distributed using a different layout, we can rearrange it to the above layout using a total of $O\left(\frac{n^2}{P}\right)$ words moved and $O(n^2)$ messages. This has no asymptotic effect on the bandwidth cost but significantly increases the latency cost in the worst case.

11.2.3 Unlimited Memory Scheme

In the UM scheme, we take $k = \log_7 P$ BFS steps in a row. Since a BFS step reduces the number of processors involved in each subproblem by a factor of 7, after k BFS steps each subproblem is assigned to a single processor, and so is computed locally with no further communication costs. We first describe a BFS step in more detail.

The matrices A and B are initially distributed as described in Section 11.2.2. In order to take a recursive step, the 14 matrices $S_1, \ldots S_7, T_1, \ldots, T_7$ must be computed (the notation follows that of Section 2.4.2). Each processor allocates space for all 14 matrices and performs local additions and subtractions to compute its portion of the matrices. Recall that the submatrices are distributed identically, so this step requires no communication. If the layouts of A and B have parameters (n, P, s), the S_i and the T_i now have layout parameters $(n/2, P, s-1)$.

The next step is to redistribute these 14 matrices so that the 7 pairs of matrices (S_i, T_i) exist on disjoint sets of $P/7$ processors. This requires disjoint sets of 7 processors performing an all-to-all communication step (each processor must send and receive a message from each of the other 6). To see this, consider the numbering of the processors base-7. On the mth BFS step, the communication is between the seven processors whose numbers agree on all digits except the mth (counting from the right). After the mth BFS step, the set of processors working on a given subproblem share the same m-digit suffix. After the above communication is performed, the layout of S_i and T_i has parameters $(n/2, P/7, s-1)$, and the sets of processors that own the T_i and S_i are disjoint for different values of i. Note that since each all-to-all only involves seven processors no matter how large P is, this algorithm does not have the scalability issues that typically come from an all-to-all communication pattern.

11.2.3.1 Memory Requirements

The extra memory required to take one BFS step is the space to store all 7 triples S_j, T_j, Q_j. Since each of those matrices is $\frac{1}{4}$ the size of A, B, and C, the extra space required at a given step is 7/4 the extra space required for the previous step. We assume that no extra

Algorithm 11.2 CAPS, in detail

```
Require: $A$, $B$, are $n \times n$ matrices
         $P$ = number of processors
         rank = processor number base-7 as an array
         $M$ = local memory size
Ensure: $C = A \cdot B$
 1: procedure C = CAPS($A$, $B$, $P$, rank, $M$)
 2:     $\ell = \left\lceil \log_2 \frac{4n}{P^{1/\omega_0} M^{1/2}} \right\rceil$
                                        ▷ $\ell$ is number of DFS steps to fit in memory
 3:     $k = \log_7 P$
 4:     call DFS($A$, $B$, $C$, $k$, $\ell$, rank)
 5: end procedure
```

```
 1: procedure DFS($A$, $B$, $C$, $k$, $\ell$, rank)
                                        ▷ Do $C = A \cdot B$ by $\ell$ DFS, then $k$ BFS steps
 2:     if $\ell \le 0$ then call BFS( $A$, $B$, $C$, $k$, rank); return
 3:     end if
 4:     for $i = 1 \dots 7$ do
 5:         locally compute $S_i$ and $T_i$ from $A$ and $B$
                                        ▷ following Strassen-Winograd
 6:         call DFS( $S_i$, $T_i$, $Q_i$, $k$, $\ell - 1$, rank )
 7:         locally compute contribution of $Q_i$ to $C$
                                        ▷ following Strassen-Winograd
 8:     end for
 9: end procedure
```

```
 1: procedure BFS($A$, $B$, $C$, $k$, rank)
                                        ▷ Do $C = A \cdot B$ by k BFS steps, then local Strassen
 2:     if $k == 0$ then call localStrassen($A$, $B$, $C$); return
 3:     end if
 4:     for $i = 1 \dots 7$ do
 5:         locally compute $S_i$ and $T_i$ from $A$ and $B$
                                        ▷ following Strassen-Winograd
 6:     end for
 7:     for $i = 1 \dots 7$ do
 8:         target = rank
 9:         target[$k$] = i
10:         send $S_i$ to target
11:         receive into $L$
                                        ▷ One part of $L$ comes from each of 7 processors
12:         send $T_i$ to target
13:         receive into $R$
                                        ▷ One part of $R$ comes from each of 7 processors
14:     end for
15:     call BFS($L$, $R$, $P$, $k - 1$, rank )
16:     for $i = 1 \dots 7$ do
17:         target = rank
18:         target[$k$] = $i$
19:         send $i$th part of $P$ to target
20:         receive from target into $Q_i$
21:     end for
22:     locally compute $C$ from $Q_i$
                                        ▷ following Strassen-Winograd
23: end procedure
```

memory is required for the local multiplications.[1] Thus, the total local memory requirement for taking k BFS steps is given by

$$\mathrm{Mem}_{\mathrm{UM}}(n,P)=\frac{3n^2}{P}\sum_{i=0}^{k}\left(\frac{7}{4}\right)^i=\frac{7n^2}{P^{2/\omega_0}}-\frac{4n^2}{P}=\Theta\left(\frac{n^2}{P^{2/\omega_0}}\right).$$

11.2.3.2 Computational Costs

The computation required at a given BFS step is that of the local additions and subtractions associated with computing the S_i and T_i and updating the output matrix C with the Q_i. Since the Strassen-Winograd algorithm performs 15 additions and subtractions, the computational cost recurrence is

$$F_{\mathrm{UM}}(n,P)=15\left(\frac{n^2}{4P}\right)+F_{\mathrm{UM}}\left(\frac{n}{2},\frac{P}{7}\right)$$

with base case $F_{\mathrm{UM}}(n,1)=c_s n^{\omega_0}-6n^2$, where c_s is the constant of Strassen's algorithm. See Section 2.4.2 for more details. The solution to this recurrence is

$$F_{\mathrm{UM}}(n,P)=\frac{c_s n^{\omega_0}-6n^2}{P}=\Theta\left(\frac{n^{\omega_0}}{P}\right).$$

11.2.3.3 Communication Costs

Consider the communication costs associated with the UM scheme. Given that the redistribution within a BFS step is performed by an all-to-all communication step among sets of 7 processors, each processor sends 6 messages and receives 6 messages to redistribute $S_1,\dots,S_7$, and the same for $T_1,\dots,T_7$. Each processor can pack the S_i and T_i data for a single other processor into one message. After the products $Q_i=S_iT_i$ are computed, each processor sends 6 messages and receives 6 messages to redistribute $Q_1,\dots,Q_7$. The size of each message varies according to the recursion depth, and is the number of words a processor owns of any S_i, T_i, or Q_i, namely $\frac{n^2}{4P}$ words.

As each of the Q_i is computed simultaneously on disjoint sets of $P/7$ processors, we obtain a cost recurrence for the entire UM scheme:

$$W_{\mathrm{UM}}(n,P)=36\frac{n^2}{4P}+W_{\mathrm{UM}}\left(\frac{n}{2},\frac{P}{7}\right)$$
$$S_{\mathrm{UM}}(n,P)=24+S_{\mathrm{UM}}\left(\frac{n}{2},\frac{P}{7}\right)$$

[1]If one does not overwrite the input, it is impossible to run Strassen in place; however using a few temporary matrices affects the analysis here by a constant factor only.

with base case $S_{\mathrm{UM}}(n,1) = W_{\mathrm{UM}}(n,1) = 0$. Thus

$$\begin{aligned} W_{\mathrm{UM}}(n,P) &= \frac{12n^2}{P^{2/\omega_0}} - \frac{12n^2}{P} = \Theta\left(\frac{n^2}{P^{2/\omega_0}}\right) \\ S_{\mathrm{UM}}(n,P) &= 24\log_7 P = \Theta\left(\log P\right). \end{aligned} \tag{11.1}$$

11.2.4 Limited Memory Scheme

In this section we discuss a scheme for traversing Strassen's recursion tree in the context of limited memory. In the LM scheme, we take ℓ DFS steps in a row followed by k BFS steps in a row, where ℓ is minimized subject to the memory constraints. That is, we use a sequence of DFS steps to reduce the problem size so that we can use the UM scheme on each subproblem without exceeding the available memory.

Consider taking a single DFS step. Rather than allocating space for and computing all 14 matrices $S_1, T_1, \ldots, S_7, T_7$ at once, the DFS step requires allocation of only one subproblem, and each of the Q_i will be computed in sequence.

Consider the ith subproblem: as before, both S_i and T_i can be computed locally. After Q_i is computed, it is used to update the corresponding quadrants of C and then discarded so that its space in memory (as well as the space for S_i and T_i) can be re-used for the next subproblem. In a DFS step, no redistribution occurs. After S_i and T_i are computed, all processors participate in the computation of Q_i.

We assume that some extra memory is available. To be precise, assume the matrices A, B, and C require only $\frac{1}{3}$ of the available memory:

$$\frac{3n^2}{P} \le \frac{1}{3}M. \tag{11.2}$$

In the LM scheme, we set

$$\ell = \max\left\{0, \left\lceil \log_2 \frac{4n}{P^{1/\omega_0}M^{1/2}} \right\rceil\right\}. \tag{11.3}$$

The following subsection shows that this choice of ℓ is sufficient not to exceed the memory capacity.

11.2.4.1 Memory Requirements

The extra memory requirement for a DFS step is the space to store one subproblem. Thus, the extra space required at this step is 1/4 the space required to store A, B, and C. The local memory requirements for the LM scheme is given by

$$\begin{aligned}
\mathrm{Mem}_{\mathrm{LM}}(n,P) &= \frac{3n^2}{P}\sum_{i=0}^{\ell-1}\left(\frac{1}{4}\right)^i + \mathrm{Mem}_{\mathrm{UM}}\left(\frac{n}{2^\ell},P\right)\\
&\le \frac{M}{3}\sum_{i=0}^{\ell-1}\left(\frac{1}{4}\right)^i + \frac{7\left(\frac{n}{2^\ell}\right)^2}{P^{2/\omega_0}}\\
&\le \frac{127}{144}M < M,
\end{aligned}$$

where the last line follows from (11.3) and (11.2). Thus, the limited memory scheme does not exceed the available memory.

11.2.4.2 Computational Costs

As in the UM case, the computation required at a given DFS step is that of the local additions and subtractions associated with computing the S_i and T_i and updating the output matrix C with the Q_i. However, since all processors participate in each subproblem and the subproblems are computed in sequence, the recurrence is given by

$$F_{\mathrm{LM}}(n,P) = 15\left(\frac{n^2}{4P}\right) + 7\cdot F_{\mathrm{LM}}\left(\frac{n}{2},P\right).$$

After ℓ steps of DFS, the size of a subproblems is $\frac{n}{2^\ell}\times\frac{n}{2^\ell}$, and there are P processors involved. We take k BFS steps to compute each of these 7^ℓ subproblems. Thus

$$F_{\mathrm{LM}}\left(\frac{n}{2^\ell},P\right) = F_{\mathrm{UM}}\left(\frac{n}{2^\ell},P\right),$$

and

$$F_{\mathrm{LM}}(n,P) = \frac{15n^2}{4P}\sum_{i=0}^{\ell-1}\left(\frac{7}{4}\right)^i + 7^\ell\cdot F_{\mathrm{UM}}\left(\frac{n}{2^\ell},P\right) = \frac{c_s n^{\omega_0} - 6n^2}{P} = \Theta\left(\frac{n_0^{\omega}}{P}\right).$$

11.2.4.3 Communication Costs

Since there are no communication costs associated with a DFS step, the recurrence is simply

$$\begin{aligned}
W_{\mathrm{LM}}(n,P) &= 7\cdot W_{\mathrm{LM}}\left(\frac{n}{2},P\right)\\
S_{\mathrm{LM}}(n,P) &= 7\cdot S_{\mathrm{LM}}\left(\frac{n}{2},P\right)
\end{aligned}$$

with base cases

$$\begin{aligned}
W_{\mathrm{LM}}\left(\frac{n}{2^\ell},P\right) &= W_{\mathrm{UM}}\left(\frac{n}{2^\ell},P\right)\\
S_{\mathrm{LM}}\left(\frac{n}{2^\ell},P\right) &= S_{\mathrm{UM}}\left(\frac{n}{2^\ell},P\right).
\end{aligned}$$

Thus the total communication costs are given by

$$\begin{aligned} W_{\text{LM}}(n,P) &= 7^\ell \cdot W_{\text{UM}}\left(\frac{n}{2^\ell},P\right) \leq \frac{12\cdot 4^{\omega_0-2}n^{\omega_0}}{PM^{\omega_0/2-1}} = \Theta\left(\frac{n^{\omega_0}}{PM^{\omega_0/2-1}}\right) \\ S_{\text{LM}}(n,P) &= 7^\ell \cdot S_{\text{UM}}\left(\frac{n}{2^\ell},P\right) \leq \frac{(4n)^{\omega_0}}{PM^{\omega_0/2}} 24\log_7 P = \Theta\left(\frac{n^{\omega_0}}{PM^{\omega_0/2}}\log P\right). \end{aligned} \tag{11.4}$$

11.2.5 Communication Optimality

Theorem 11.1. *CAPS has computational cost $\Theta\left(\frac{n^{\omega_0}}{P}\right)$, bandwidth cost*

$$W = \Theta\left(\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2-1}}, \frac{n^2}{P^{2/\omega_0}}\right\}\right),$$

and latency cost

$$S = \Theta\left(\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2}}\log P, \log P\right\}\right).$$

Proof. In the case that $M \geq \text{Mem}_{\text{UM}}(n,P) = \Omega\left(\frac{n^2}{P^{2/\omega_0}}\right)$ the UM scheme is possible. Then the communication costs are given by (11.1) which matches the lower bound of Theorem 6.8. Thus the UM scheme is communication-optimal (up to a logarithmic factor in the latency cost and assuming that the data is initially distributed as described in Section 11.2.2). For smaller values of M, the LM scheme must be used. Then the communication costs are given by (11.4) and match the lower bound of Theorem 5.10, so the LM scheme is also communication-optimal. $\square$

By Theorems 5.10 and 6.8, we see that CAPS has optimal computational and bandwidth costs, and that its latency cost is at most $\log P$ away from optimal.

We note that for the LM scheme, since both the computational and communication costs are proportional to $\frac{1}{P}$, we can expect perfect strong scaling: given a fixed problem size, increasing the number of processors by some factor will decrease each cost by the same factor. However, this strong scaling property has a limited range. As P increases, holding everything else constant, the global memory size PM increases as well. The limit of perfect strong scaling is exactly when there is enough memory for the UM scheme. See Section 6.2.2 for details.

11.3 Analysis of Other Algorithms

In the section we detail the asymptotic communication costs of other matrix multiplication algorithms, both classical and Strassen-based. These communication costs and the corresponding lower bounds are summarized in Table 11.1.

Many of the algorithms described in this section are hybrids of two different algorithms. We use the convention that the names of the hybrid algorithms are composed of the names

		Flops	**Bandwidth**	**Latency**
Classical	Lower Bound [19, 95]	$\frac{n^3}{P}$	$\max\left\{\frac{n^3}{PM^{1/2}}, \frac{n^2}{P^{2/3}}\right\}$	$\max\left\{\frac{n^3}{PM^{3/2}}, 1\right\}$
	2D [49, 71]	$\frac{n^3}{P}$	$\frac{n^2}{P^{1/2}}$	$P^{1/2}$
	3D [2, 35]	$\frac{n^3}{P}$	$\frac{n^2}{P^{2/3}}$	$\log P$
	2.5D (optimal) [137]	$\frac{n^3}{P}$	$\max\left\{\frac{n^3}{PM^{1/2}}, \frac{n^2}{P^{2/3}}\right\}$	$\frac{n^3}{PM^{3/2}} + \log P$
Strassen-based	Lower Bound [19, 25]	$\frac{n^{\omega_0}}{P}$	$\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2-1}}, \frac{n^2}{P^{2/\omega_0}}\right\}$	$\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2}}, 1\right\}$
	2D-Strassen [108]	$\frac{n^{\omega_0}}{P^{(\omega_0-1)/2}}$	$\frac{n^2}{P^{1/2}}$	$P^{1/2}$
	Strassen-2D [78, 108]	$\left(\frac{7}{8}\right)^\ell \frac{n^3}{P}$	$\left(\frac{7}{4}\right)^\ell \frac{n^2}{P^{1/2}}$	$7^\ell P^{1/2}$
	2.5D-Strassen	$\max\left\{\frac{n^3}{PM^{3/2-\omega_0/2}}, \frac{n^{\omega_0}}{P^{\omega_0/3}}\right\}$	$\max\left\{\frac{n^3}{PM^{1/2}}, \frac{n^2}{P^{2/3}}\right\}$	$\frac{n^3}{PM^{3/2}} + \log P$
	Strassen-2.5D	$\left(\frac{7}{8}\right)^\ell \frac{n^3}{P}$	$\max\left\{\left(\frac{7}{8}\right)^\ell \frac{n^3}{PM^{1/2}}, \left(\frac{7}{4}\right)^\ell \frac{n^2}{P^{2/3}}\right\}$	$\left(\frac{7}{8}\right)^\ell \frac{n^3}{PM^{3/2}} + 7^\ell \log P$
	CAPS (optimal)	$\frac{n^{\omega_0}}{P}$	$\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2-1}}, \frac{n^2}{P^{2/\omega_0}}\right\}$	$\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2}} \log P, \log P\right\}$

Table 11.1: Asymptotic matrix multiplication computational and communication costs of algorithms and corresponding lower bounds. Here $\omega_0 = \lg 7 \approx 2.81$ is the exponent of Strassen; ℓ is the number of Strassen steps taken. None of the Strassen-based algorithms except for CAPS attain the lower bounds of Chapter 5 or Section 6.2.1.2, see Section 11.3 for a discussion of each.

of the two component algorithms, hyphenated. The first name describes the algorithm used at the top level, on the largest problems, and the second describes the algorithm used at the base level on smaller problems.

11.3.1 Classical Algorithms

Classical algorithms must communicate asymptotically more than an optimal Strassen-based algorithm. To compare the lower bounds, it is necessary to consider three cases for the memory size: when the memory-dependent bounds dominate for both classical and Strassen, when the memory-dependent bound dominates for classical, but the memory-independent bound dominates for Strassen, and when the memory-independent bounds dominate for both classical and Strassen. This analysis is detailed in [105, Appendix B]. Briefly, the factor by which the classical bandwidth cost exceeds the Strassen bandwidth cost is P^a where a ranges from $\frac{2}{\omega_0} - \frac{2}{3} \approx 0.046$ to $\frac{3-\omega_0}{2} \approx 0.10$ depending on the relative problem size. The same sort of analysis is used throughout Section 11.3 to compare each algorithm with the Strassen-based lower bounds.

Various parallel classical matrix multiplication algorithms minimize communication relative to the classical lower bounds for certain amounts of local memory M. For example, Cannon's algorithm [49] minimizes communication for $M = O(n^2/P)$. Several more practical algorithms exist (such as SUMMA [71]) which use the same amount of local memory and have the same asymptotic communication costs. We call this class of algorithms "2D" because the communication patterns follow a two-dimensional processor grid.

Another class of algorithms, known as "3D" [35, 2] because the communication pattern maps to a three-dimensional processor grid, uses more local memory and reduces communication relative to 2D algorithms. This class of algorithms minimizes communication relative to the classical lower bounds for $M = \Omega(n^2/P^{2/3})$. As shown in Section 6.2.1.1, it is not possible to use more memory than $M = \Theta(n^2/P^{2/3})$ to reduce communication.

Recently, a more general algorithm has been developed which minimizes communication in all cases. Because it reduces to a 2D and 3D for the extreme values of M but interpolates for the values between, it is known as the "2.5D" algorithm [137].

11.3.2 2D-Strassen

One idea to parallelize Strassen-based algorithms is to use a 2D classical algorithm for the inter-processor communication, and use the fast matrix multiplication algorithm locally [108]. We call such an algorithm "2D-Strassen". It is straightforward to implement, but cannot attain all the computational speedup from Strassen since it uses a classical algorithm for part of the computation. In particular, it does not use Strassen for the largest matrices, when Strassen provides the greatest reduction in computation. As a result, the computational cost exceeds $\Theta(n^{\omega_0}/P)$ by a factor of $P^{(3-\omega_0)/2} \approx P^{0.10}$. The 2D-Strassen algorithm has the same communication cost as 2D algorithms, and hence does not match the communication costs of CAPS. In comparing the 2D-Strassen bandwidth cost, $\Theta(n^2/P^{1/2})$, to the CAPS bandwidth cost in Section 11.2, note that for the problem to fit in memory we always have $M = \Omega(n^2/P)$. The bandwidth cost exceeds that of CAPS by a factor of P^a, where a ranges from $(3-\omega_0)/2 \approx .10$ to $2/\omega_0 - 1/2 \approx .21$, depending on the relative problem size. Similarly, the latency cost, $\Theta(P^{1/2})$, exceeds that of CAPS by a factor of P^a where a ranges from $(3-\omega_0)/2 \approx .10$ to $1/2 = .5$.

11.3.3 Strassen-2D

The "Strassen-2D" algorithm applies ℓ DFS steps of Strassen's algorithm at the top level, and performs the 7^ℓ smaller matrix multiplications using a 2D algorithm. By choosing certain data layouts as in Section 11.2.2, it is possible to do the additions and subtractions for Strassen's algorithm without any communication [108]. However, Strassen-2D is also unable to match the communication costs of CAPS. Moreover, the speedup of Strassen-2D in computation comes at the expense of extra communication. For large numbers of Strassen steps ℓ, Strassen-2D can approach the computational lower bound of Strassen, but each step increases the bandwidth cost by a factor of $\frac{7}{4}$ and the latency cost by a factor of 7. Thus the bandwidth cost of Strassen-2D is a factor of $\left(\frac{7}{4}\right)^\ell$ higher than 2D-Strassen, which is already higher than that of CAPS. The latency cost is even worse: Strassen-2D is a factor of 7^ℓ higher than 2D-Strassen.

One can reduce the latency cost of Strassen-2D at the expense of a larger memory footprint. Since Strassen-2D runs a 2D algorithm 7^ℓ times on the same set of processors, it is possible to pack together messages from independent matrix multiplications. In the best case, the latency cost is reduced to the cost of 2D-Strassen, which is still above that of CAPS, at the expense of using a factor of $\left(\frac{7}{4}\right)^\ell$ more memory.

11.3.4 2.5D-Strassen

A natural idea is to replace a 2D classical algorithm in 2D-Strassen with the superior 2.5D classical algorithm to obtain an algorithm we call 2.5D-Strassen. This algorithm uses the 2.5D algorithm for the inter-processor communication, and then uses Strassen for the local computation. When $M = \Theta(n^2/P)$, 2.5D-Strassen is exactly the same as 2D-Strassen, but when there is extra memory it both decreases the communication cost and decreases the computational cost since the local matrix multiplications are performed (using Strassen) on larger matrices. To be precise, the computational cost exceeds the lower bound by a factor of P^a where a ranges from $1-\frac{\omega_0}{3} \approx 0.064$ to $\frac{3-\omega_0}{2} \approx 0.10$ depending on the relative problem size. The bandwidth cost exceeds the bandwidth cost of CAPS by a factor of P^a where a ranges from $\frac{2}{\omega_0} - \frac{2}{3} \approx 0.046$ to $\frac{3-\omega_0}{2} \approx 0.10$. In terms of latency, the cost of $\frac{n^3}{PM^{3/2}} + \log P$ exceeds the latency cost of CAPS by a factor ranging from $\log P$ to $P^{(3-\omega_0)/2} \approx P^{0.10}$, depending on the relative problem size.

11.3.5 Strassen-2.5D

Similarly, by replacing a 2D algorithm with 2.5D in Strassen-2D, one obtains the new algorithm we call Strassen-2.5D. First one takes ℓ DFS steps of Strassen, which can be done without communication, and then one applies the 2.5D algorithm to each of the 7^ℓ subproblems. The computational cost is exactly the same as Strassen-2D, but the communication cost will typically be lower. Each of the 7^ℓ subproblems is multiplication of $n/2^\ell \times n/2^\ell$ matrices. Each subproblem uses only $1/4^\ell$ as much memory as the original problem. Thus there may be a large amount of extra memory available for each subproblem, and the lower communication costs of the 2.5D algorithm help. The choice of ℓ that minimizes the bandwidth cost is

$$\ell_{\text{opt}} = \max\left\{0, \left\lceil \log_2 \frac{n}{M^{1/2}P^{1/3}} \right\rceil\right\}.$$

The same choice minimizes the latency cost. Note that when $M \geq \frac{n^2}{P^{2/3}}$, taking zero Strassen steps minimizes the communication within the constraints of the Strassen-2.5D algorithm. With $\ell = \ell_{\text{opt}}$, the bandwidth cost is a factor of $P^{1-\omega_0/3} \approx P^{0.064}$ above that of CAPS. Additionally, the computational cost is not optimal, and using $\ell = \ell_{\text{opt}}$, the computational cost exceeds the optimal by a factor of $P^{1-\omega_0/3}M^{3/2-\omega_0/2} \approx P^{0.064}M^{0.096}$.

It is also possible to take $\ell > \ell_{\text{opt}}$ steps of Strassen to decrease the comptutational cost further. However the decreased computational cost comes at the expense of higher communication cost, as in the case of Strassen-2D. In particular, each extra step over ℓ_{opt}

increases the bandwidth cost by a factor of $\frac{7}{4}$ and the latency cost by a factor of 7. As with Strassen-2D, it is possible to use extra memory to pack together messages from several subproblems and decrease the latency cost, but not the bandwidth cost.

11.4 Performance Results

We have implemented CAPS using MPI on a Cray XT4, and compared it to various previous classical and Strassen-based algorithms. The benchmarking data is shown in Figure 11.2.

11.4.1 Experimental Setup

The nodes of the Cray XT4 have 8GB of memory and a quad-core AMD "Bupdapest" processor running at 2.3GHz. We treat the entire node as a single processor, and when we use the classical algorithm we call the optimized threaded BLAS in Cray's LibSci to provide parallelism between the four cores in a node. The peak flop rate is 9.2 GFLOPS per core, or 36.8 GFLOPS per node. The machine consists of 9,572 nodes. All the data in Figure 11.2 is for multiplying two square matrices with $n = 94080$.

11.4.2 Performance

Note that the vertical scale of Figure 11.2 is "effective GFLOPS", which is a useful measure for comparing classical and fast matrix multiplication algorithms. It is calculated as

$$\text{Effective GFLOPS} = \frac{2n^3}{(\text{Execution time in seconds})10^9}. \tag{11.5}$$

For classical algorithms, which perform $2n^3$ floating point operations, this gives the actual GFLOPS. For fast matrix multiplication algorithms it gives the performance relative to classical algorithms, but does not accurately represent the number of floating point operations performed. For this problem size and number of Strassen steps taken, the actual number of flops is about 45% of that of the classical algorithms.

Our algorithm outperforms all previous algorithms, and attains performance as high as 49.1 effective GFLOPS/node, which is 33% above the theoretical maximum for all classical algorithms. Compared with the best classical implementation, our speedup ranges from 51% for small values of P up to 94% when using most of the machine. Compared with the best previous parallel Strassen algorithms, our speedup ranges from 24% up to 184%. Unlike previous Strassen algorithms, we are able to attain substantial speedups over the entire range of processors.

11.4.3 Strong Scaling

Figure 11.2 is a strong scaling plot: the problem size is fixed and each algorithm is run with P ranging from the minimum that provides enough memory up to the largest allowed

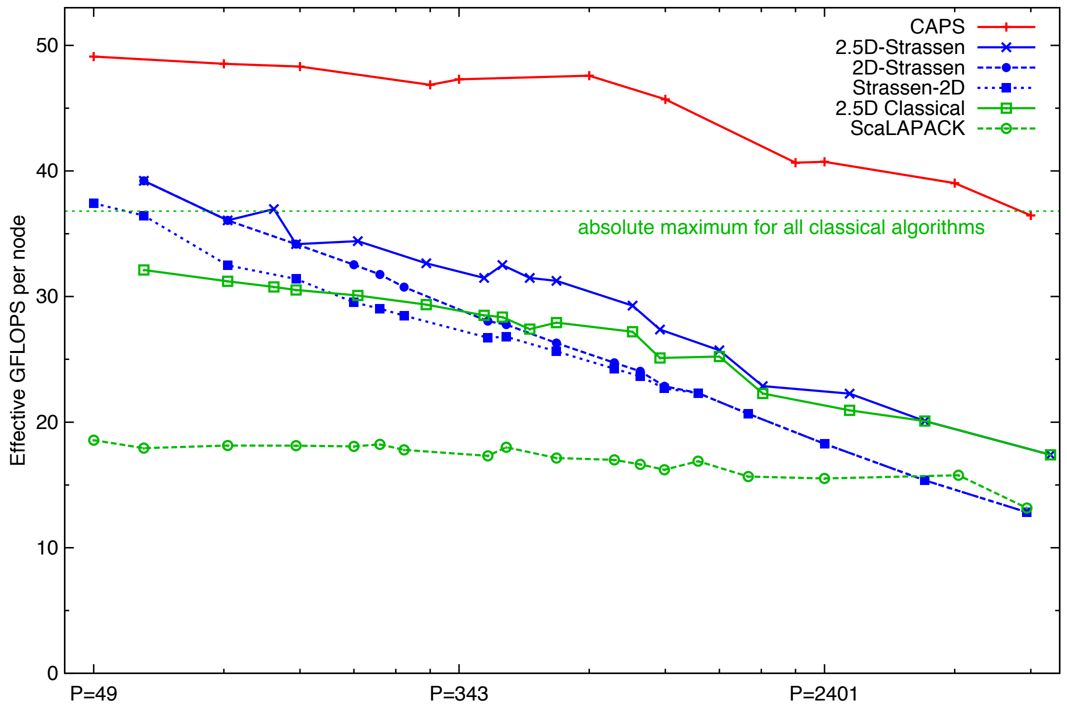


Figure 11.2: Strong scaling performance of various matrix multiplication algorithms on a Cray XT4 for fixed problem size $n = 94080$. The top line is CAPS as described in Section 11.2, and substantially outperforms all the other classical and Strassen-based algorithms. The horizontal axis is the number of nodes in log-scale. The vertical axis is effective GFLOPS, which are a performance measure rather than a flop rate, as discussed in Section 11.4.2. See Section 11.4.4 for a description of each implementation.

value of P smaller than the size of the machine. Perfect strong scaling corresponds to a horizontal line in the plot. As the communication analysis predicts, CAPS exhibits better strong scaling than any of the other algorithms (with the exception of ScaLAPACK, which obtains very good strong scaling by having poor performance for small values of P).

11.4.4 Details of the Implementations

11.4.4.1 CAPS

This implementation is the CAPS algorithm, with a few modifications from the presentation in Section 11.2. First, when computing locally it switches to classical matrix multiplication below some size n_0. Second, it is generalized to run on $P = c7^k$ processors for $c \in \{1, 2, 3, 6\}$ rather than just 7^k processors. As a result, the base-case classical matrix multiplication is done on c processors rather than 1. Finally, implementation uses the Winograd variant of Strassen; see Section 2.4.2 for more details. Every point in the plot is tuned to use the best interleaving pattern of BFS and DFS steps, and the best total number of Strassen steps. For points in the figure, the optimal total number of Strassen steps is always 5 or 6.

11.4.4.2 ScaLAPACK

We use ScaLAPACK [44] as optimized by Cray in LibSci. This is an implementation of the SUMMA algorithm, and can run on an arbitrary number of processors. It should give the best performance if P is a perfect square so the processors can be placed on a square 2D grid. All the runs shown in Figure 11.2 are with P a perfect square.

11.4.4.3 2.5D Classical

This is the code of [137]. It places the P processors in a grid of size $\sqrt{P/c} \times \sqrt{P/c} \times c$, and requires that $\sqrt{P/c}$ and c are integers with $1 \leq c \leq P^{1/3}$, and c divides $\sqrt{P/c}$. Additionally, it gets the best performance if c is as large as possible, given the constraint that c copies of the input and output matrices fit in memory. In the case that $c = 1$ this code is an optimized implementation of SUMMA. The values of P and c for the runs in Figure 11.2 are chosen to get the best performance. The optimal permissible values of c ranged from 1 on 64 processors to 20 on 8000 processors.

11.4.4.4 Strassen-2D

Following the algorithm of [78, 108], this implementation uses the DFS code from the implementation of CAPS at the top level, and then uses the optimized SUMMA code from the 2.5D implementation with $c = 1$. Since the classical code requires that P is a perfect square, this requirement applies here as well. The number of Strassen steps taken is tuned to give the best performance for each P value, and the optimal number varies from 0 to 2.

11.4.4.5 2D-Strassen

Following the algorithm of [108], the 2D-Strassen implementation is analogous to the Strassen-2D implementation, but with the classical algorithm run before taking local Strassen steps. Similarly, the same code is used for local Strassen steps here and in our implementation of CAPS. This code also requires that P is a perfect square. The number of Strassen steps is tuned for each P value, and the optimal number varies from 0 to 3.

11.4.4.6 2.5D-Strassen

This implementation uses the 2.5D implementation to reduce the problem to one processor, then takes several Strassen steps. The processor requirements are the same as for the 2.5D implementation. The number of Strassen steps is tuned for each number of processors, and the optimal number varies from 0 to 3. We also tested the Strassen-2.5D algorithm, but its performance was always lower than 2.5D-Strassen in our experiments.

11.5 Conclusions and Future Work

11.5.1 Stability of Fast Matrix Multiplication

CAPS has the same stability properties as sequential versions of Strassen. For discussion of the stability of fast matrix multiplication algorithms, see [59, 85]. We highlight a few main points here. The tightest error bounds for classical matrix multiplication are component-wise: $|C - \hat{C}| \leq n\epsilon|A| \cdot |B|$, where $\hat{C}$ is the computed result and ϵ is the machine precision. Strassen and other fast algorithms do not satisfy component-wise bounds but do satisfy the slightly weaker norm-wise bounds: $\|C - \hat{C}\| \leq f(n)\epsilon\|A\|\|B\|$, where $\|A\| = \max_{i,j} A_{ij}$ and f is polynomial in n [85]. Accuracy can be improved with the use of diagonal scaling matrices: $D_1CD_3 = D_1AD_2 \cdot D_2^{-1}BD_3$. It is possible to choose D_1, D_2, D_3 so that the error bounds satisfy either $|C_{ij} - \hat{C}_{ij}| \leq f(n)\epsilon\|A(i,:)\|\|B(:,j)\|$ or $\|C - \hat{C}\| \leq f(n)\epsilon\||A| \cdot |B|\|$. By scaling, the error bounds on Strassen become comparable to those of many other dense linear algebra algorithms, such as LU and QR decomposition [58]. Thus using Strassen for the matrix multiplications in a larger computation will often not harm the stability at all.

11.5.2 Hardware Scaling

Although Strassen performs asymptotically less computation and communication than classical matrix multiplication, it is more demanding on the hardware. That is, if one wants to run matrix multiplication near the peak CPU speed, Strassen is more demanding of the memory size and communication bandwidth. This is because the ratio of computational cost to bandwidth cost is lower for Strassen than for classical. From the lower bound in Theorem 5.10, the asymptotic ratio of computational cost to bandwidth cost is $M^{\omega_0/2-1}$ for Strassen-based algorithms, versus $M^{1/2}$ for classical algorithms. This means that it is harder to run Strassen near peak than it is to run classical matrix multiplication near peak. In terms of the machine parameters β and γ introduced in Section 2.1.3, the condition to be able to be computation-bound is $\gamma M^{1/2} \geq c\beta$ for classical matrix multiplication and $\gamma M^{\omega_0/2-1} \geq c'\beta$ for Strassen. Here c and c' are constants that depend on the constants in the communication and computational costs of classical and Strassen-based matrix multiplication.

The above inequalities may guide hardware design as long as classical and Strassen matrix multiplication are considered important computations. They apply both to the distributed case, where M is the local memory size and β is the inverse network bandwidth, and to the sequential/shared-memory case where M is the cache size and β is the inverse memory-cache bandwidth.

11.5.3 Optimizing On-Node Performance

Note that although our implementation performs above the classical peak performance, it performs well below the corresponding Strassen-Winograd peak, defined by the time it takes to perform $c_s n^{\omega_0}/P$ flops at the peak speed of each processor. To some extent, this is because

Strassen is more demanding on the hardware, as noted above. However we have not yet analyzed whether the amount our performance is below Strassen peak can be entirely accounted for based on machine parameters. It is also possible that a high performance shared-memory Strassen implementation might provide substantial speedups for our implementation, as well as for 2D-Strassen and 2.5D-Strassen.

11.5.4 Parallelizing Other Algorithms

11.5.4.1 Another Optimal Classical Algorithm

We can apply our parallelization approach to recursive classical matrix multiplication to obtain a communication-optimal algorithm. This algorithm has the same asymptotic communication costs as the 2.5D algorithm [137]. We observed comparable performance to the 2.5D algorithm on our experimental platform. As with CAPS, this algorithm has not been optimized for contention, whereas the 2.5D algorithm is very well optimized for contention on torus networks.

11.5.4.2 Other Fast Matrix Multiplication Algorithms

Our approach of executing a recursive algorithm in parallel by traversing the recursion tree in DFS (sequential) or BFS (parallel) manners is not limited to Strassen's algorithm. All fast square matrix multiplication algorithms are built out of ways to multiply $n_0 \times n_0$ matrices using $q < n_0^3$ multiplications. Like with Strassen and Strassen-Winograd, they compute q linear combinations of entries of each of A and B, multiply these pairwise, then compute the entries of C as linear combinations of these.[2] CAPS can be easily generalized to any such multiplication, with the following modifications:

- The number of processors P is a power of q.
- The data layout must be such that all n_0^2 blocks of A, B, and C are distributed equally among the P processors with the same layout.
- The BFS and DFS determine whether the q multiplications are performed in parallel or sequentially.

The communication costs are then exactly as above, but with $\omega_0 = \log_{n_0} q$.

It is unclear whether any of the faster matrix multiplication algorithms are useful in practice. One reason is that the fastest algorithms are not explicit. Rather, there are non-constructive proofs that the algorithms exist. To implement these algorithms, they would have to be found, which appears to be a difficult problem. The generalization of CAPS described in this section does apply to all of them, so we have proved the existence of a communication-avoiding non-explicit parallel algorithm corresponding to every fast matrix

[2]By [124], *all* fast matrix multiplication algorithms can be expressed in this bilinear form.

multiplication algorithm. We conjecture that the algorithms are all communication optimal (*i.e.*, they attain a known lower bound), but that is not yet proved since the lower bound proofs in Chapter 5 and Section 6.2.1.2 may not apply to all fast matrix multiplication algorithms. In cases where the lower bounds do apply, they match the performance of the generalization of CAPS, and so the algorithms are communication optimal.

Chapter 12

Conclusion

In this thesis, we have considered the fundamental operations in dense linear algebra. We have seen that, even for a fixed computation, many algorithms exist that exhibit a range of communication costs when analyzed on simple sequential and parallel models. By proving lower bounds on the communication required for particular computations and developing more communication-efficient algorithms, we are able to devise optimal algorithms on state-of-the-art hardware and observe better performance than existing approaches. Extending the algorithmic ideas and techniques to computations outside of dense linear algebra can benefit many more application areas. Even in scientific computing, other "dwarves" include spectral methods (FFTs), sparse linear algebra, and graph algorithms, and communication-avoiding ideas have already proved effective for these areas [36, 114, 136]. We believe similar ideas can and will benefit an even larger range of computations and applications in the future.

We now briefly summarize our contributions and list possible directions for future research within the realm of dense linear algebra. In this work, we prove communication lower bounds for dense and sparse, sequential and parallel algorithms for a general set of classical $\Theta(n^3)$ linear algebra computations (Chapters 3 and 4). We also prove communication lower bounds for Strassen's and Strassen-like fast matrix multiplication algorithms (Chapters 5 and 6) and establish a separate set of lower bounds that apply to parallel algorithms and set limits on an algorithm's ability to perfectly strong scale (Chapter 6). In Chapters 7 and 8, we summarize the state-of-the-art in communication efficiency for both sequential and parallel algorithms for the fundamental computations to which the lower bounds apply. Finally, we present new algorithms for three particular computations: computing a symmetric-indefinite factorization (Chapter 9), reducing a symmetric band matrix to tridiagonal form via orthogonal similarity transformations (Chapter 10), and parallelizing Strassen's matrix multiplication algorithm (Chapter 11).

While there are many open problems described throughout the previous chapters, and developing optimized implementations on the most current hardware is ongoing challenge, we highlight here only a few areas of possible future algorithmic research:

- developing communication-optimal extra-memory algorithms for all of the computations discussed in Chapter 8, which likely includes tightening lower bounds;

- extending algorithmic techniques for dense matrices to sparse direct methods, where computations are more irregular and also communication bound;
- finding and parallelizing other fast, practical matrix multiplication algorithms that are asymptotically superior to Strassen's algorithm but also can be implemented to be faster in practice; and
- developing parallel fast linear algebra computations (*e.g.,* for solving linear systems or least squares problems)–parallelizing those algorithms that rely on fast matrix multiplication subroutines and are already communication optimal in the sequential case.

As mentioned in Chapter 1, the costs of communication relative to computation are growing, and increased parallelism (on and off chip) means that careful consideration of data movement is necessary to achieve satisfactory performance on today's and future machines. Not only is communication becoming more important, it is also becoming more difficult to reason about: processors are growing more complex, with heterogeneous computational units and the ability to vary clock speeds with time, for two examples. Developing efficient algorithms and optimized implementations, within linear algebra and in other domains, requires both strong theoretical foundations and continual adaptation to new architectures. Beyond the particular results in this thesis, the performance analysis and lower bound techniques establish a way of thinking about algorithm design that will be increasingly important as machines evolve.

Bibliography

[1] J. O. Aasen. “On the reduction of a symmetric matrix to tridiagonal form”. In: *BIT Numerical Mathematics* 11.3 (1971). 10.1007/BF01931804, pp. 233–242.

[2] R. Agarwal, S. Balle, F. Gustavson, M. Joshi, and P. Palkar. “A Three-Dimensional Approach to Parallel Matrix Multiplication”. In: *IBM Journal of Research and Development* 39.5 (1995), pp. 575–582.

[3] R. Agarwal, F. Gustavson, and M. Zubair. “A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication”. In: *IBM Journal of Research and Development* 38.6 (1994), pp. 673–681.

[4] A. Aggarwal, A. K. Chandra, and M. Snir. “Communication complexity of PRAMs”. In: *Theoretical Computer Science* 71.1 (1990), pp. 3–28.

[5] A. Aggarwal and J.S. Vitter. “The input/output complexity of sorting and related problems”. In: *Communications of the ACM* 31.9 (1988), pp. 1116–1127.

[6] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan. *PLASMA Users' Guide*. Available from http://icl.cs.utk.edu/plasma/.

[7] N. Ahmed and K. Pingali. “Automatic Generation of Block-Recursive Codes”. In: *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*. Euro-Par '00. London, UK: Springer-Verlag, 2000, pp. 368–378.

[8] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. Also available from http://www.netlib.org/lapack/. Philadelphia, PA, USA: SIAM, 1992.

[9] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. 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] T. Auckenthaler. “Highly Scalable Eigensolvers for Petaflop Applications”. PhD thesis. Fakultät für Informatik, Technische Universität München, 2012.

[11] T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P. Willems. “Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations”. In: *Parallel Computing* 37.12 (2011), pp. 783–794.

[12] T. Auckenthaler, H.-J. Bungartz, T. Huckle, L. Krämer, B. Lang, and P. Willems. “Developing algorithms and software for the parallel solution of the symmetric eigenvalue problem”. In: *Journal of Computational Science* 2.3 (2011), pp. 272–278.

[13] Z. Bai, J. Demmel, and M. Gu. “An inverse free parallel spectral divide and conquer algorithm for nonsymmetric eigenproblems”. In: *Numerische Mathematik* 76.3 (1997), pp. 279–308.

[14] G. Ballard, D. Becker, J. Demmel, J. Dongarra, A. Druinsky, I. Peled, O. Schwartz, S. Toledo, and I. Yamazaki. *Communication-Avoiding Symmetric-Indefinite Factorization.* Tech. rep. UCB/EECS-2013-127. EECS Department, University of California, Berkeley, July 2013.

[15] G. Ballard, D. Becker, J. Demmel, J. Dongarra, A. Druinsky, I. Peled, O. Schwartz, S. Toledo, and I. Yamazaki. “Implementing a Blocked Aasen’s Algorithm with a Dynamic Scheduler on Multicore Architectures”. In: *Proceedings of the 27th IEEE International Parallel Distributed Processing Symposium.* IPDPS ’13. Awarded the Best Paper prize in the Algorithms track. May 2013, pp. 895–907.

[16] G. Ballard, A. Buluç, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, and S. Toledo. “Communication optimal parallel multiplication of sparse random matrices”. In: *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures.* SPAA ’13. Montréal, Québec, Canada: ACM, 2013, pp. 222–231.

[17] G. Ballard, J. Demmel, and I. Dumitriu. *Communication-optimal parallel and sequential eigenvalue and singular value algorithms.* EECS Technical Report EECS-2011-14. UC Berkeley, Feb. 2011.

[18] G. Ballard, J. Demmel, and A. Gearhart. “Brief announcement: communication bounds for heterogeneous architectures”. In: *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures.* SPAA ’11. San Jose, California, USA: ACM, 2011, pp. 257–258.

[19] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. “Brief announcement: strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds”. In: *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures.* SPAA ’12. New York, NY, USA: ACM, 2012, pp. 77–79.

[20] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. “Communication-optimal parallel algorithm for Strassen’s matrix multiplication”. In: *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures.* SPAA ’12. New York, NY, USA: ACM, 2012, pp. 193–204.

[21] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. "Graph Expansion Analysis for Communication Costs of Fast Rectangular Matrix Multiplication". In: *Design and Analysis of Algorithms*. Ed. by G. Even and D. Rawitz. Vol. 7659. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 13–36.

[22] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. *Strong Scaling of Matrix Multiplication Algorithms and Memory-Independent Communication Lower Bounds*. Tech. rep. UCB/EECS-2012-31. EECS Department, University of California, Berkeley, Mar. 2012.

[23] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. "Communication-optimal parallel and sequential Cholesky decomposition". In: *Proceedings of the 22nd Symposium on Parallelism in Algorithms and Architectures*. SPAA '09. Calgary, AB, Canada: ACM, 2009, pp. 245–252.

[24] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. "Communication-optimal Parallel and Sequential Cholesky Decomposition". In: *SIAM Journal on Scientific Computing* 32.6 (2010), pp. 3495–3523.

[25] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. "Graph expansion and communication costs of fast matrix multiplication". In: *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '11. Awarded the Best Paper prize. San Jose, California, USA: ACM, 2011, pp. 1–12.

[26] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. "Graph expansion and communication costs of fast matrix multiplication". In: *Journal of the ACM* 59.6 (Dec. 2012), 32:1–32:23.

[27] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. *Minimizing communication in linear algebra*. EECS Technical Report EECS-2011-15. UC Berkeley, Feb. 2011.

[28] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. "Minimizing Communication in Numerical Linear Algebra". In: *SIAM Journal on Matrix Analysis and Applications* 32.3 (2011). Awarded the 2009-2011 SIAM Linear Algebra Prize, pp. 866–901.

[29] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. *Sequential Communication Bounds for Fast Linear Algebra*. Tech. rep. EECS-2012-36. UC Berkeley, Mar. 2012.

[30] G. Ballard, J. Demmel, and N. Knight. *Avoiding Communication in Successive Band Reduction*. Tech. rep. UCB/EECS-2013-131. EECS Department, University of California, Berkeley, July 2013.

[31] G. Ballard, J. Demmel, and N. Knight. "Communication avoiding successive band reduction". In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '12. New Orleans, Louisiana, USA: ACM, 2012, pp. 35–44.

[32] G. Ballard, J. Demmel, B. Lipshitz, O. Schwartz, and S. Toledo. "Communication efficient Gaussian elimination with partial pivoting using a shape morphing data layout". In: *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '13. Montréal, Québec, Canada: ACM, 2013, pp. 232–240.

[33] M. Benioff and E. Lazowska, eds. *Computational Science: Ensuring America's Competitiveness*. The Networking and Information Technology Research and Development (NITRD) Program. President's Information Technology Advisory Committee, June 2005.

[34] J. Bennett, A. Carbery, M. Christ, and T. Tao. "Finite bounds for Hölder-Brascamp-Lieb multilinear inequalities". In: *Mathematical Research Letters* 17.4 (2010), pp. 647–666.

[35] J. Berntsen. "Communication efficient matrix multiplication on hypercubes". In: *Parallel Computing* 12.3 (1989), pp. 335 –342.

[36] G. Bilardi, M. Scquizzato, and F. Silvestri. "A Lower Bound Technique for Communication on BSP with Application to the FFT". In: *Euro-Par 2012 Parallel Processing*. Ed. by C. Kaklamanis, T. Papatheodorou, and P. Spirakis. Vol. 7484. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 676–687.

[37] D. Bini, M. Capovani, F. Romani, and G. Lotti. "$O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication". In: *Information Processing Letters* 8.5 (1979), pp. 234 –235.

[38] C. Bischof, B. Lang, and X. Sun. "A Framework for Symmetric Band Reduction". In: *ACM Transactions on Mathematical Software* 26.4 (Dec. 2000), pp. 581–601.

[39] C. Bischof, M. Marques, and X. Sun. "Parallel bandreduction and tridiagonalization". In: *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*. Vol. 1. SIAM. 1993, pp. 383–390.

[40] C. Bischof and X. Sun. *A framework for symmetric band reduction and tridiagonalization*. Technical Report MCS-P298-0392. Argonne National Laboratory, 1992.

[41] C. Bischof, X. Sun, and B. Lang. "Parallel tridiagonalization through two-step band reduction". In: *Proceedings of the Scalable High-Performance Computing Conference*. IEEE Computer Society Press, May 1994, pp. 23–27.

[42] C. Bischof and C. Van Loan. "The WY representation for products of Householder matrices". In: *SIAM Journal on Scientific and Statistical Computing* 8.1 (1987).

[43] C. H. Bischof, B. Lang, and X. Sun. "Algorithm 807: The SBR Toolbox—software for successive band reduction". In: *ACM Transactions on Mathematical Software* 26.4 (Dec. 2000), pp. 602–616.

[44] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Also available from http://www.netlib.org/scalapack/. Philadelphia, PA, USA: SIAM, May 1997.

[45] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. "An Updated Set of Basic Linear Algebra Subroutines (BLAS)". In: *ACM Transactions on Mathematical Software* 28.2 (June 2002).

[46] G. Blelloch, R. Chowdhury, P. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. "Provably good multicore cache performance for divide-and-conquer algorithms". In: *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '08. San Francisco, California: SIAM, 2008, pp. 501–510.

[47] J. Bunch and L. Kaufman. "Some stable methods for calculating inertia and solving symmetric linear systems". In: *Mathematics of Computation* 31.137 (Jan. 1977), pp. 163–179.

[48] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. *A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures*. Tech. rep. 191. LAPACK Working Note, Sept. 2007.

[49] L. Cannon. "A cellular computer to implement the Kalman filter algorithm". PhD thesis. Bozeman, MN: Montana State University, 1969.

[50] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn. "Collective communication: theory, practice, and experience". In: *Concurrency and Computation: Practice and Experience* 19.13 (2007), pp. 1749–1783.

[51] M. Christ, J. Demmel, N. Knight, T. Scanlon, and K. Yelick. *Communication Lower Bounds and Optimal Algorithms for Programs That Reference Arrays - Part 1*. Tech. rep. UCB/EECS-2013-61. EECS Department, University of California, Berkeley, May 2013.

[52] P. Colella. *Defining Software Requirements for Scientific Computing*. Presentation. 2004.

[53] Cray. *Cray Application Developer's Environment User's Guide*. Available from http://docs.cray.com/books/S-2396-610/S-2396-610.pdf.

[54] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken. "LogP: a practical model of parallel computation". In: *Communications of the ACM* 39 (Nov. 1996), pp. 78–85.

[55] T. Davis and Y. Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Transactions on Mathematical Software* 38 (2011), 1:1–1:25.

[56] J. Demmel. *An arithmetic complexity lower bound for computing rational functions, with applications to linear algebra*. Tech. rep. UCB/EECS-2013-126. EECS Department, University of California, Berkeley, July 2013.

[57] J. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[58] J. Demmel, I. Dumitriu, and O. Holtz. "Fast linear algebra is stable". In: *Numerische Mathematik* 108.1 (2007). 10.1007/s00211-007-0114-x, pp. 59–91.

[59] J. Demmel, I. Dumitriu, O. Holtz, and R. Kleinberg. "Fast Matrix Multiplication is Stable". In: *Numerische Mathematik* 106.2 (2007), pp. 199–224.

[60] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger. "Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication". In: *Proceedings of the 27th IEEE International Symposium on Parallel and Distributed Processing*. IPDPS '13. 2013, pp. 261–272.

[61] J. Demmel, L. Grigori, M. Gu, and H. Xiang. *Communication Avoiding Rank Revealing QR Factorization with Column Pivoting*. Tech. rep. UCB/EECS-2013-46. EECS Department, University of California, Berkeley, May 2013.

[62] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. "Communication-optimal Parallel and Sequential QR and LU Factorizations". In: *SIAM Journal on Scientific Computing* 34.1 (2012), A206–A239.

[63] J. Demmel, O. Marques, B. Parlett, and C. Vömel. "Performance and accuracy of LAPACK's symmetric tridiagonal eigensolvers". In: *SIAM Journal on Scientific Computing* 30.3 (Mar. 2008), pp. 1508–1526.

[64] F. Desprez and F. Suter. "Impact of mixed-parallelism on parallel implementations of the Strassen and Winograd matrix multiplication algorithms: Research Articles". In: *Concurrency and Computation: Practice and Experience* 16.8 (2004), pp. 771–797.

[65] C. C. Douglas, M. Heroux, G. Slishman, and R. M. Smith. "GEMMW: A Portable Level 3 BLAS Winograd Variant of Strassen's Matrix-Matrix Multiply Algorithm". In: *Journal of Computational Physics* 110.1 (1994), pp. 1–10.

[66] E. Elmroth and F. Gustavson. "New serial and parallel recursive QR factorization algorithms for SMP systems". In: *Applied Parallel Computing. Large Scale Scientific and Industrial Problems*. Ed. by B. Kågström et al. Vol. 1541. Lecture Notes in Computer Science. Springer, 1998, pp. 120–128.

[67] D. Eppstein and Z. Galil. "Annual review of computer science: vol. 3, 1988". In: ed. by J. Traub. Palo Alto, CA, USA: Annual Reviews Inc., 1988. Chap. Parallel algorithmic techniques for combinatorial computation, pp. 233–283.

[68] R. Floyd. "Algorithm 97: Shortest path". In: *Communications of the ACM* 5.6 (June 1962), p. 345.

[69] J. D. Frens and D. S. Wise. “QR factorization with Morton-ordered quadtree matrices for memory re-use and parallelism”. In: *ACM SIGPLAN Notices* 38.10 (2003), pp. 144–154.

[70] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. “Cache-Oblivious Algorithms”. In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. FOCS ’99. Washington, DC, USA: IEEE Computer Society, 1999, p. 285.

[71] R. A. van de Geijn and J. Watts. “SUMMA: scalable universal matrix multiplication algorithm”. In: *Concurrency - Practice and Experience* 9.4 (1997), pp. 255–274.

[72] E. Georganas, J. Gonzalez-Dominguez, E. Solomonik, Yili Zheng, J. Tourino, and K. Yelick. “Communication avoiding and overlapping for numerical linear algebra”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’12. 2012, pp. 1–11.

[73] A. George. “Nested dissection of a regular finite element mesh”. In: *SIAM Journal on Numerical Analysis* 10 (1973), pp. 345–363.

[74] J. R. Gilbert and R. E. Tarjan. “The Analysis of a Nested Dissection Algorithm”. In: *Numerische Mathematik* (1987), pp. 377–404.

[75] 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*. Champaign, IL, USA: University of Illinois Press, 1988, pp. 171–179.

[76] S. L. Graham, M. Snir, and C. A. Patterson, eds. *Getting up to Speed: The Future of Supercomputing*. Report of National Research Council of the National Academies Sciences. Washington, D.C.: The National Academies Press, 2004.

[77] R. Granat, B. Kågström, D. Kressner, and M. Shao. *Parallel library software for the multishift QR algorithm with aggressive early deflation*. Tech. rep. UMINF 12.06. Umeå University, 2012.

[78] B. Grayson, A. Shah, and R. van de Geijn. “A High Performance Parallel Strassen Implementation”. In: *Parallel Processing Letters*. Vol. 6. 1995, pp. 3–12.

[79] L. Grigori, P.-Y. David, J. W. Demmel, and S. Peyronnet. “Brief announcement: Lower bounds on communication for sparse Cholesky factorization of a model problem”. In: *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’10. Thira, Santorini, Greece: ACM, 2010, pp. 79–81.

[80] L. Grigori, J. Demmel, and H. Xiang. “CALU: A Communication Optimal LU Factorization Algorithm”. In: *SIAM Journal on Matrix Analysis and Applications* 32.4 (2011), pp. 1317–1350.

[81] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. “FLAME: Formal Linear Algebra Methods Environment”. In: *ACM Transactions on Mathematical Software* 27.4 (Dec. 2001), pp. 422–455.

[82] B. C. Gunter and R. A. Van De Geijn. "Parallel out-of-core computation and updating of the QR factorization". In: *ACM Transactions on Mathematical Software* 31.1 (2005), pp. 60–78.

[83] F. G. Gustavson. "Recursion leads to automatic variable blocking for dense linear-algebra algorithms". In: *IBM Journal of Research and Development* 41.6 (1997), pp. 737–756.

[84] A. Haidar, H. Ltaief, and J. Dongarra. "Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. 2011, p. 8.

[85] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. 2nd. Philadelphia, PA: SIAM, 2002.

[86] Nicholas J. Higham. "Notes on Accuracy and Stability of Algorithms in Numerical Linear Algebra". In: *The Graduate Student's Guide to Numerical Analysis*. Ed. by Mark Ainsworth, Jeremy Levesley, and Marco Marletta. Springer Berlin Heidelberg, 1999, pp. 47–82.

[87] A. J. Hoffman, M. S. Martin, and D. J. Rose. "Complexity bounds for regular finite difference and finite element grids". In: *SIAM Journal on Numerical Analysis* 10 (1973), pp. 364–369.

[88] J. W. Hong and H. T. Kung. "I/O complexity: The red-blue pebble game". In: *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*. STOC '81. Milwaukee, Wisconsin, United States: ACM, 1981, pp. 326–333.

[89] J. E. Hopcroft and L. R. Kerr. "On Minimizing the Number of Multiplications Necessary for Matrix Multiplication". English. In: *SIAM Journal on Applied Mathematics* 20.1 (1971), pp. 30–36.

[90] S. Hunold, T. Rauber, and G. Rünger. "Combining building blocks for parallel multi-level matrix multiplication". In: *Parallel Computing* 34.6-8 (July 2008), pp. 411–426.

[91] IBM. *Engineering and Scientific Software Library Guide and Reference*. Available from http://publib.boulder.ibm.com/epubs/pdf/a2322683.pdf.

[92] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std. 754-2008* (2008), pp. 1–58.

[93] Intel. *Math Kernel Library*. Available from http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation.

[94] D. Irony and S. Toledo. "The snap-back pivoting method for symmetric indefinite matrices". In: *SIAM Journal on Matrix Analysis and Applications* 28 (2006), pp. 398–424.

[95] D. Irony, S. Toledo, and A. Tiskin. “Communication lower bounds for distributed-memory matrix multiplication”. In: *J. Parallel Distrib. Comput.* 64.9 (2004), pp. 1017–1026.

[96] L. Karlsson and B. Kågström. “Parallel two-stage reduction to Hessenberg form using dynamic scheduling on shared-memory architectures”. In: *Parallel Computing* 37.12 (2011). 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA’10), pp. 771 –782.

[97] L. Kaufman. “Band reduction algorithms revisited”. In: *ACM Transactions on Mathematical Software* 26.4 (2000), pp. 551–567.

[98] L. Kaufman. “Banded eigenvalue solvers on vector machines”. In: *ACM Transactions on Mathematical Software* 10 (1984), pp. 73–86.

[99] Linda Kaufman. “The retraction algorithm for factoring banded symmetric matrices”. In: *Numerical Linear Algebra with Applications* 14 (2007), pp. 237–254.

[100] Amal Khabou, James Demmel, Laura Grigori, and Ming Gu. *LU Factorization with Panel Rank Revealing Pivoting and Its Communication Avoiding Version.* Tech. rep. UCB/EECS-2012-15. To appear in SIMAX. EECS Department, University of California, Berkeley, Jan. 2012.

[101] B. Kumar, C.-H. Huang, R. Johnson, and P. Sadayappan. “A tensor product formulation of Strassen’s matrix multiplication algorithm with memory reduction”. In: *Proceedings of Seventh International Parallel Processing Symposium.* Apr. 1993, pp. 582 –588.

[102] B. Lang. “A parallel algorithm for reducing symmetric banded matrices to tridiagonal form”. In: *SIAM Journal on Scientific Computing* 14.6 (Nov. 1993), pp. 1320–1338.

[103] B. Lang. “Parallele Reduktion Symmetrischer Bandmatrizen auf Tridiagonalgestalt”. PhD thesis. Fakultät für Mathematik, Universität Karlsruhe, 1991.

[104] C. E. Leiserson. *Personal communication with G. Ballard, J. Demmel, O. Holtz, and O. Schwartz.* 2008.

[105] B. Lipshitz, G. Ballard, J. Demmel, and O. Schwartz. “Communication-avoiding parallel Strassen: Implementation and performance”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* SC ’12. Salt Lake City, Utah, 2012, 101:1–101:11.

[106] Benjamin Lipshitz. “Communication-Avoiding Parallel Recursive Algorithms for Matrix Multiplication”. MA thesis. EECS Department, University of California, Berkeley, May 2013.

[107] L. H. Loomis and H. Whitney. “An inequality related to the isoperimetric inequality”. In: *Bulletin of the AMS* 55 (1949), pp. 961–962.

[108] Q. Luo and J. Drake. “A scalable parallel Strassen’s matrix multiplication algorithm for distributed-memory computers”. In: *Proceedings of the 1995 ACM Symposium on Applied Computing*. SAC ’95. Nashville, Tennessee, United States: ACM, 1995, pp. 221–226.

[109] P. Luszczek, H. Ltaief, and J. Dongarra. “Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures”. In: *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*. IPDPS ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 944–955.

[110] R. van de Geijn M. Schatz J. Poulson. *Scalable Universal Matrix Multiplication Algorithms: 2D and 3D Variations on a Theme*. Tech. rep. 2013.

[111] R. S. Martin and J. H. Wilkinson. “Reduction of the Symmetric Eigenproblem $Ax = \lambda Bx$ and Related Problems to Standard Form”. In: *Numerische Mathematik* 11 (1968), pp. 99–110.

[112] W. McColl and A. Tiskin. “Memory-Efficient Matrix Multiplication in the BSP Model”. In: *Algorithmica* 24.3-4 (1999), pp. 287–297.

[113] S. Mohanty and S. Gopalan. “I/O efficient QR and QZ algorithms”. In: *High Performance Computing (HiPC), 2012 19th International Conference on*. 2012, pp. 1–9.

[114] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. “Minimizing communication in sparse matrix solvers”. In: *Proceedings of the International Conference on High Performance Computing Networking, Storage and Analysis*. SC ’09. Portland, Oregon, 2009, 36:1–36:12.

[115] K. Murata and K. Horikoshi. “A new method for the tridiagonalization of the symmetric band matrix”. In: *Information Processing in Japan* 15 (1975), pp. 108–112.

[116] Y. Nakatsukasa and N. Higham. *Stable and efficient spectral divide and conquer algorithms for the symmetric eigenvalue decomposition and the SVD*. Tech. rep. MIMS EPrint 2012.52. The University of Manchester, 2012.

[117] NVIDIA. *CUDA Toolkit Documentation: CUBLAS*. Available from http://docs.nvidia.com/cuda/cublas/index.html.

[118] J.-S. Park, M. Penner, and V. Prasanna. “Optimizing graph algorithms for improved cache performance”. In: *Parallel and Distributed Systems, IEEE Transactions on* 15.9 (2004), pp. 769–782.

[119] B Parlett and J Reid. “On the solution of a system of linear equations whose matrix is symmetric but not definite”. In: *BIT Numerical Mathematics* 10 (1970), pp. 386–397.

[120] J. Poulson, R. van de Geijn, and J. Benninghof. *(Parallel) Algorithms for Two-sided Triangular Solves and Matrix Multiplication*. Tech. rep. Available from http://www.cs.utexas.edu/users/flame/pubs/ElementalGenEig.pdf. University of Texas, 2012.

[121] J. Poulson, B. Marker, R. van de Geijn, J. Hammond, and N. Romero. "Elemental: A New Framework for Distributed Memory Dense Matrix Computations". In: *ACM Transactions on Mathematical Software* 39 (2013), 13:1–13:24.

[122] C. Puglisi. "Modification of the Householder method based on compact WY representation". In: *SIAM Journal on Scientific and Statistical Computing* 13.3 (1992), pp. 723–726.

[123] S. Rajamanickam. "Efficient Algorithms for Sparse Singular Value Decomposition". PhD thesis. University of Florida, 2009.

[124] R. Raz. "On the complexity of matrix product". In: *SIAM Journal on Computing* 32.5 (2003), pp. 1356–1369.

[125] Miroslav Rozloznik, Gil Shklarski, and Sivan Toledo. "Partitioned triangular tridiagonalization". In: *ACM Transactions on Mathematical Software* 37 (4 2011), 38:1–38:16.

[126] H. Rutishauser. "On Jacobi rotation patterns". In: *Proceedings of Symposia in Applied Mathematics*. Vol. 15. AMS, 1963, pp. 219–239.

[127] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Boston: PWS Publishing Co., 1996.

[128] A. Sankar, D. Spielman, and S.-H. Teng. "Smoothed Analysis of the Condition Numbers and Growth Factors of Matrices". In: *SIAM Journal on Matrix Analysis and Applications* 28 (2006), pp. 446–476.

[129] J. E. Savage. "Extending the Hong-Kung Model to Memory Hierarchies". In: *Computing and Combinatorics*. Vol. 959. Springer Verlag, 1995, pp. 270–281.

[130] J. E. Savage and M. Zubair. "A unified model for multicore architectures". In: *Proceedings of the 1st International Forum on Next-Generation Multicore/Manycore Technologies*. IFMT '08. Cairo, Egypt: ACM, 2008, 9:1–9:12.

[131] R. Schreiber and C. Van Loan. "A storage-efficient WY representation for products of Householder transformations". In: *SIAM Journal on Scientific and Statistical Computing* 10.1 (1989), pp. 53–57.

[132] H. Schwarz. "Algorithm 183: reduction of a symmetric bandmatrix to triple diagonal form". In: *Communications of the ACM* 6.6 (June 1963), pp. 315–316.

[133] H. Schwarz. "Tridiagonalization of a symmetric band matrix". In: *Numerische Mathematik* 12 (4 1968), pp. 231–241.

[134] Mark P. Sears, Ken Stanley, and Greg Henry. "Application of a High Performance Parallel Eigensolver to Electronic Structure Calculations". In: *Proceedings of the IEEE/ACM Conference on Supercomputing*. Nov. 1998.

[135] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide*. 2nd. Springer-Verlag, 1976.

[136] E. Solomonik, A. Buluç, and J. Demmel. “Minimizing communication in all-pairs shortest-paths”. In: *Proceedings of the 27th IEEE International Parallel Distributed Processing Symposium.* IPDPS ’13. May 2013, pp. 548–559.

[137] E. Solomonik and J. Demmel. “Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms”. In: *Euro-Par 2011 Parallel Processing.* Ed. by Emmanuel Jeannot, Raymond Namyst, and Jean Roman. Vol. 6853. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 90–109.

[138] F. Song, J. Dongarra, and S. Moore. “Experiments with Strassen’s Algorithm: From Sequential to Parallel”. In: *Proceedings of Parallel and Distributed Computing and Systems (PDCS).* Dallax, Texas: ACTA, Nov. 2006.

[139] V. Strassen. “Gaussian elimination is not optimal”. In: *Numerische Mathematik* 13 (4 1969). 10.1007/BF02165411, pp. 354–356.

[140] A. Tiskin. “Bulk-Synchronous Parallel Gaussian Elimination”. In: *Journal of Mathematical Sciences* 108 (6 2002). 10.1023/A:1013588221172, pp. 977–991.

[141] A. Tiskin. “Communication-efficient parallel generic pairwise elimination”. In: *Future Generation Computer Systems* 23.2 (2007), pp. 179 –188.

[142] S. Toledo. “Locality of Reference in LU Decomposition with Partial Pivoting”. In: *SIAM Journal on Matrix Analysis and Applications* 18.4 (1997), pp. 1065–1081.

[143] S. Tomov, R. Nath, P. Du, and J. Dongarra. *MAGMA Users’ Guide.* Available from http://icl.cs.utk.edu/magma/index.html.

[144] L. Trefethen and R. Schreiber. “Average-Case Stability of Gaussian Elimination”. In: *SIAM Journal on Matrix Analysis and Applications* 11 (1990), pp. 335–360.

[145] L. Valiant. “A bridging model for multi-core computing”. In: *Journal of Computer and System Sciences* 77.1 (2011), pp. 154 –166.

[146] L. Valiant. “A bridging model for parallel computation”. In: *Communications of the ACM* 33.8 (1990), pp. 103–111.

[147] F. Van Zee, R. van de Geijn, and G. Quintana-Orti. *Restructuring the QR algorithm for performance.* Tech. rep. Available from http://www.cs.utexas.edu/users/flame/pubs/RestructuredQRTOMS.pdf. University of Texas, 2013.

[148] S. Warshall. “A Theorem on Boolean Matrices”. In: *Journal of the ACM* 9.1 (1962), pp. 11–12.

[149] R. C. Whaley, A. Petitet, and J. Dongarra. “Automated Empirical Optimization of Software and the ATLAS Project”. In: *Parallel Computing* 27.1–2 (2001), pp. 3–35.

[150] J.H. Wilkinson. “Householder’s method for symmetric matrices”. In: *Numerische Mathematik* 4.1 (1962), pp. 354–361.

[151] V. Williams. “Multiplying matrices faster than Coppersmith-Winograd”. In: *Proceedings of the 44th Annual Symposium on Theory of Computing*. STOC ’12. New York, New York, USA: ACM, 2012, pp. 887–898.

[152] S. Winograd. “On the multiplication of 2×2 matrices”. In: *Linear Algebra and its Applications* 4.4 (1971), pp. 381–388.

[153] S. Winograd. “Some remarks on fast multiplication of polynomials”. In: *Complexity of Sequential and Parallel Numerical Algorithms*. Ed. by J. Traub. New York: Academic Press, 1973, pp. 181–196.

[154] D. Wise. “Ahnentafel Indexing into Morton-Ordered Arrays, or Matrix Locality for Free”. In: *Euro-Par 2000 Parallel Processing*. Ed. by A. Bode, T. Ludwig, W. Karl, and R. Wismöller. Vol. 1900. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pp. 774–783.

[155] C.-Q. Yang and B.P. Miller. “Critical path analysis for the execution of parallel and distributed programs”. In: *Proceedings of the 8th International Conference on Distributed Computing Systems*. June 1988, pp. 366–373.

[156] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson. “An experimental comparison of cache-oblivious and cache-conscious programs”. In: *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’07. San Diego, California, USA: ACM, 2007, pp. 93–104.