Exploiting Data Sparsity in Parallel Matrix Powers Computations

Nicholas Knight
Erin Carson
James Demmel

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-47
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-47.html

May 3, 2013

Acknowledgement

We acknowledge funding from Microsoft (award #024263) and Intel (award #024894), and matching funding by UC Discovery (award #DIG07-10227), with additional support from ParLab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung, and support from MathWorks. We also acknowledge the support of the US DOE (grants DE-SC0003959, DE-SC0004938, DE-SC0005136, DE-SC0008700, DE-AC02-05CH11231, DE-FC02-06ER25753, and DE-FC02-07ER25799) and DOD (DARPA award #HR0011-12-2-0016 and NDSEG fellowship 32 CFR 168a).

Exploiting Data Sparsity in Parallel Matrix Powers Computations

Nicholas Knight, Erin Carson, James Demmel
University of California, Berkeley
{knight,ecc2z,demmel}@cs.berkeley.edu

Abstract

The increasingly high relative cost of moving data on modern parallel machines has caused a paradigm shift in the design of high-performance algorithms: to achieve efficiency, one must focus on strategies which minimize data movement, rather than minimize arithmetic operations. We call this a *communication-avoiding* approach to algorithm design.

In this work, we derive a new parallel communication-avoiding matrix powers algorithm for matrices of the form $A = D+USV^H$, where D is sparse and USV^H has low rank but may be dense. Matrices of this form arise in many practical applications, including power-law graph analysis, circuit simulation, and algorithms involving hierarchical ($\mathcal{H}$) matrices, such as multigrid methods, fast multipole methods, numerical partial differential equation solvers, and preconditioned iterative methods.

If A has this form, our algorithm enables a communication-avoiding approach. We demonstrate that, with respect to the cost of computing k sparse matrix-vector multiplications, our algorithm asymptotically reduces the parallel latency by a factor of $O(k)$ for small additional bandwidth and computation costs. Using problems from real-world applications, our performance model predicts that this reduction in communication allows for up to 24× speedups on petascale machines.

1 Introduction

The runtime of an algorithm can be modeled as a function of *computation* cost, proportional to the number of arithmetic operations, and *communication* cost, proportional to the amount of data movement. Traditionally, to increase the efficiency of an algorithm, one sought to minimize the arithmetic complexity. On modern computer systems, however, the time to move one word of data is much greater than the time to complete one floating point operation. This can cause the performance of algorithms with low arithmetic intensity (see [11]) to be *communication bound*. Technology trends indicate that the performance gap between communication and computation will only widen in future systems. This has resulted in a paradigm shift in the design of high-performance algorithms: to achieve efficiency, one must focus on strategies which minimize data movement, rather than minimize arithmetic operations. We call this a *communication-avoiding* approach to algorithm design.

We consider simplified machine models, where a sequential machine moves words of data between a *slow memory* of unbounded capacity and a *fast memory* of size M words, and a parallel machine consists of p processors, who move data between their *local memories* each of size M words. In both cases, data has a fixed width and is moved in *messages* of up to M contiguous words. In the parallel case, we assume the processors communicate in a point-to-point fashion over a completely connected network (so no contention) and each processor can send or receive at most one message at a time. We characterize an algorithm by its *latency cost*, the number of messages sent, its *bandwidth cost*, the number of words moved, and its *arithmetic (flop) cost*, the number of arithmetic operations performed. We characterize a sequential or parallel machine by its latency α, reciprocal bandwidth β, and arithmetic (flop) rate γ, in addition to M and p, and will estimate the runtime T of an algorithm (along the critical path) as

$$T = (\#\text{messages} \cdot \alpha) + (\#\text{words moved} \cdot \beta) + (\#\text{flops} \cdot \gamma). \tag{1}$$

For simplicity, we will not discuss overlapping communication and computation, although this potential factor of 2 savings may be important in practice.

Computing k repeated sparse matrix-vector[1] multiplications (SpMVs), or, a *matrix powers* computation, with $A \in \mathbb{C}^{n \times n}$ and $x \in \mathbb{C}^{n \times q}$, where typically $q \ll n$, can be written

$$K_{k+1}(A, \{p_j\}_{j=0}^{k}, x) \coloneqq [x^{(0)}, \ldots, x^{(k)}] \coloneqq [p_0(A)x, p_1(A)x, \ldots, p_k(A)x], \tag{2}$$

where p_j is a degree-j polynomial. For $i = 1$ to q, the span of the i^{th} columns of $x^{(0)}, x^{(1)}, \ldots$ is often called a *Krylov (sub)space*. Matrix powers computations constitute a core kernel in a variety of applications, including steepest descent algorithms, the power method to compute PageRank, and Krylov subspace methods for linear systems and eigenvalue problems. Due to its low arithmetic intensity, SpMV performance is communication bound on modern architectures.

In [4], the authors derive parallel communication-avoiding matrix powers algorithms to compute (2) that achieve an $O(k)$ reduction in parallel latency cost versus computing k repeated SpMVs, for a set number of iterations [4, 9]. These algorithms assume that A is *well partitioned*, i.e., we can partition A such that for each processor, computing powers up to A^k involves communication only between $O(1)$ nearest neighbors.

Although such advances show promising speedups for a wide variety of problems, the requirement that A is well partitioned in the communication-avoiding matrix powers formulation of [4] excludes matrices whose partitions have poor surface-to-volume ratio. In this work, we consider matrices of the form $A = D + USV^H$, where D is well partitioned and USV^H may not be well partitioned but has low rank. (Recall $x^H = \overline{x}^T$ denotes the Hermitian transpose of x, and that if an n-by-n matrix has rank r, it can be stored and applied to a vector with $O(nr)$ words and flops, instead of the usual $O(n^2)$.) There are many practical situations where such structures arise, including analysis of power-law (or scale-free) graphs representing the Web and social networks, and circuit simulation. Hierarchical ($\mathcal{H}$) matrices (see, e.g., [1]), which appear in numerical partial differential equation solvers, multigrid methods, fast multipole methods, and preconditioned iterative methods, also have this form.

In this paper, we derive a new parallel communication-avoiding matrix powers algorithm for matrices of the form $A = D + USV^H$, where we only require that D (not A) be well-partitioned. If this splitting is possible, our algorithm enables a communication-avoiding approach; we demonstrate that, with respect to the cost of computing k SpMVs (the standard algorithm), our algorithm asymptotically reduces the parallel latency by a factor of $O(k)$ for small additional bandwidth and computational costs. Using detailed complexity analysis, our performance model predicts that this reduction in communication allows for up to 24× speedups over the standard algorithm on petascale machines.

2 Background and Related Work

We discuss work in related areas, namely parallel communication-avoiding matrix powers algorithms and the serial blocking covers technique. We touch on applications that can benefit from our approach.

2.1 Efficient Matrix Powers Algorithms

The communication-avoiding matrix powers algorithms derived in [5] fuse together a sequence of k SpMV operations into one kernel invocation. Depending on the nonzero structure of A (more precisely, of the polynomials $\{p_j(A)\}_{j=1}^{k}$), this enables communication avoidance in both serial and parallel implementations.

The serial matrix powers kernel reorganizes the k SpMVs to maximize reuse of A and the $k+1$ vectors, ideally reading A and x once, and writing the k output vectors only once. When reading A is the dominant communication cost versus reading/writing the vectors (a common situation), this implies an k-fold decrease in both latency and bandwidth cost.

In parallel, the matrix powers kernel reorganizes the computation in a similar way but with a slightly different goal, since in a parallel SpMV operation only vector entries must be communicated. The parallel matrix powers optimization avoids interprocessor synchronization by storing some redundant elements of A and x on different processors and performing redundant computation to compute the k matrix powers without further synchronization. Provided the additional communication cost to distribute x is a lower-order term (equivalently, A^k is well partitioned), this gives k-fold savings in latency.

[1] When $q > 1$, an 'SpMV' is really a sparse matrix-dense matrix multiplication (SpMM)

We note that this discussion applies even if A is not given explicitly.

Serial and parallel variants of the matrix powers kernel, for both structured and general sparse matrices, are described in [8], which summarizes most of [4] and elaborates on the implementation in [9]. Within [8], we refer the reader to the complexity analysis in Tables 2.3-2.4, the performance modeling in §2.6, and the performance results in §2.10.3 and §2.11.3, which demonstrate that this optimization leads to speedups in practice. For example, for a 2D 9-point stencil on a $n^{1/2}$-by-$n^{1/2}$ mesh with p processors, assuming $k \ll (n/p)^{1/2}$, the number of arithmetic operations grows by a factor $1 + 2k(p/n)^{1/2}$, the number of messages decreases by a factor of k, and the number of words moved grows by a factor of $1+k(p/n)^{1/2}$ [8]. Therefore since the additional arithmetic operations and additional words moved are lower order terms, we expect the parallel matrix powers algorithm of [8] to give a $\Theta(k)$ speedup on latency-bound problems.

2.2 The Blocking Covers Technique

Many earlier research efforts sought to reorganize out-of-core algorithms to minimize data movement within the memory hierarchy on a single processor. In [6], Hong and Kung prove a lower bound on I/O complexity for a matrix powers computation on a regular mesh: if the computational graph has a *τ-neighborhood cover*, then the number of I/Os can be reduced by a factor of τ over the standard method. This is the same restriction required in the matrix powers algorithms in [4], namely, that A be well partitioned.

A shortcoming of Hong and Kung's technique is that certain graphs with low diameter (such as multigrid graphs) cannot be covered efficiently with small, high-diameter subgraphs. In [7], this restriction is overcome by relaxing the constraint that dependencies in the computational graph be respected, using a *blocking covers* technique. A blocking cover of the computational graph has the property that the subgraphs forming the cover have large diameters (equivalent to our definition of well partitioned) once a small number of vertices (the *blockers*) have been removed. The authors show that, by introducing a variable for each blocker for each of the τ iterations and maintaining linear dependences among the removed vertices, each subgraph can compute τ matrix powers on its data without communicating with other subgraphs. We extend the blocking covers technique of [7] to the parallel case, and further generalize this technique to handle a larger class of data-sparse matrix representations.

2.3 Motivating Applications

Many scientific applications require solving linear systems $Ax = b$. Iterative methods are commonly used when the coefficient matrix is large and sparse. The most general and flexible class of iterative methods are preconditioned Krylov subspace methods. In each iteration m, the approximate solution is chosen from the expanding Krylov subspace, $\mathcal{K}_{m+1}(M^{-1}A, x) = \text{span}(x, (M^{-1}A)x, \ldots, (M^{-1}A)^m x)$, or some variation (depending on the preconditioner M^{-1} used). This includes the power method, e.g., in the PageRank algorithm.

Significant reductions in communication can be achieved by rearranging the iterations to compute a k element basis in one step (a matrix powers computation), essentially unrolling the inner loop k times. Such methods are called k-step, or communication-avoiding, Krylov subspace methods. There is a wealth of literature related to k-step methods. Space constraints prohibit an in-depth discussion; we direct the reader to the thorough overview given in [5, §1.5-1.6].

As mentioned above, there are many applications where iterative methods are used for matrices with the property of being mostly sparse, but with a few low-rank components that may be dense. This includes matrices from Web graphs, social networks, and circuit simulations. Hierarchical semiseparable (HSS) matrices [3], commonly used in PDE solvers, also have this property. Our technique is general and can be implemented to avoid communication in all these applications.

3 Derivation

Recall that, given matrices $A \in \mathbb{C}^{n\times n}$ and $x \in \mathbb{C}^{n\times q}$, and $k \in \mathbb{N}$, our task is to compute the matrix in (2). In this work, we will assume that the polynomials p_j in (2) satisfy a three-term recurrence[2]

$$\begin{aligned} &p_0(z) \coloneqq 1, \quad p_1(z) \coloneqq (z-\alpha_0)p_0(z)/\gamma_0, \quad \text{and} \\ &p_j(z) \coloneqq ((z-\alpha_{j-1})p_{j-1}(z) - \beta_{j-2}p_{j-2}(z))/\gamma_{j-1} \quad \text{for } j > 1. \end{aligned} \tag{3}$$

It is convenient to represent the polynomials $\{p_j\}_{j=0}^k$ by their coefficients, stored in a tridiagonal matrix

$$T_k \coloneqq \text{tridiag}\begin{pmatrix} * & \beta_0 & \cdots & \beta_{k-2} \\ \alpha_0 & \alpha_1 & \cdots & \alpha_{k-1} \\ \gamma_0 & \gamma_1 & \cdots & \gamma_{k-1} \end{pmatrix} \in \mathbb{C}^{(k+1)\times k}. \tag{4}$$

3.1 Parallel Matrix Powers Algorithms

For reference, we review two of the parallel matrix powers algorithms from [4, 5, 8], referred to by the authors as PA0 and PA1. PA0 refers to the naïve algorithm for computing (2) via k SpMV operations, and PA1 is the communication-avoiding variant. For simplicity, we use versions of PA0 and PA1 which do not exploit overlapping communication and computation. While our approach can be extended to exploit overlapping communication and computation, the potential savings are limited by a factor of 2 and thus will not affect the asymptotic complexity of the algorithms given here.

Let the notation $\text{nz}(A) = \{(i,j) : A_{ij} \text{ treated as nonzero}\}$ represent the edges in the directed graph of A, and let the notation $A_{\mathcal{I}}$ indicate the submatrix of A consisting of rows $i \in \mathcal{I}$. To simplify the discussion (without affecting correctness), we will ignore cancellation, meaning we assume $\text{nz}(p_j(A)) \subseteq \text{nz}(p_{j+1}(A))$ and every entry of $x^{(j)}$ is treated as nonzero for all $j \geq 0$.

We construct a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ representing the dependencies in computing $x^{(j)} \coloneqq p_j(A)x$ for every $0 \leq j \leq k$. First, denoting row i of $x^{(j)}$ by $x_i^{(j)}$, we define the $n(k+1)$ vertices $\mathcal{V} \coloneqq \{x_i^{(j)} : 1 \leq i \leq n, 0 \leq j \leq k\}$. The edge set $\mathcal{E}$ consists of k copies of $\text{nz}(A)$, between each adjacent pair of the $k+1$ *levels* $\mathcal{V}^{(j)} \coloneqq \{x_i^{(j)} : 1 \leq i \leq n\}$, unioned with the edges due to the polynomial recurrence, i.e.,

$$\mathcal{E} \coloneqq \left\{ \left(x_{i_1}^{(j+1)}, x_{i_2}^{(j)}\right) : \substack{0\leq j<k, \\ (i_1,i_2)\in \text{nz}(A)} \right\} \cup \left\{ \left(x_i^{(j+d')}, x_i^{(j)}\right) : \substack{1\leq d'\leq d, \\ 0\leq j\leq k-d', \\ 1\leq i\leq n} \right\}$$

where d is the number of nonzero superdiagonals of T_k (counting the main diagonal). We ignore possible sparsity *within* these diagonals; this simplifies the discussion without affecting correctness.

Now we partition $\mathcal{V}$ 'rowwise,' that is, each $x_i^{(j)}$ is assigned a processor *affinity* $m \in \{0, \ldots, p-1\}$, for $0 \leq j \leq k$. Let $\mathcal{V}_m$ and $\mathcal{V}_m^{(j)}$ restrict $\mathcal{V}$ and $\mathcal{V}^{(j)}$ to their elements with affinity m. Let $\mathcal{R}(\mathcal{S})$ denote the *reachability set* of $\mathcal{S} \subseteq \mathcal{V}$, i.e., the set $\mathcal{S}$ and vertices reachable from $\mathcal{S}$ via (directed) paths in $\mathcal{G}$; then as with $\mathcal{V}$, we define the subsets $\mathcal{R}^{(j)}$, $\mathcal{R}_m$, and $\mathcal{R}_m^{(j)}$ of $\mathcal{R}$.

At the end of the computation, processor m has computed/stored (a superset of) the entries $\mathcal{V}_m$. Thus, for PA0, processor m must own $A_{\{i:x_i^{(j)}\in\mathcal{V}_m\}}$ and $\mathcal{V}_m^{(0)}$, and for PA1, processor m must own $A_{\{i:x_i^{(1)}\in\mathcal{R}(\mathcal{V}_m)\}}$ and $\mathcal{R}^{(0)}(\mathcal{V}_m)$.

With this notation, we present the parallel matrix powers algorithms PA0 (Alg. 1) and PA1 (Alg. 2), as pseudocode for processor m. The advantage of PA1 over PA0 is that it may send fewer messages between processors: whereas PA0 requires k rounds of messages, PA1 requires only one. If the number of other processors $\ell \neq m$ that each processor m must communicate with is the same for both algorithms, PA1 obtains a k-fold latency savings. In general, however, PA1 incurs greater bandwidth, arithmetic, and storage costs, as processors may perform redundant computations to avoid communication (see Table 1). Whether this tradeoff is worthwhile depends on many machine and algorithm parameters; we refer to the detailed analysis and modeling in [4, 5, 8].

[2]We see no obstruction to generalizing to longer recurrences; we focus on three-term recurrences, commonly used in applications.

PA0	Flops	$\sum_{\substack{x_i^{(j)} \in \mathcal{V}_m \\ 1 \le j \le k}} q(2\lvert\mathrm{nz}(A_i)\rvert - 1)$
	Words	$\sum_{j=1}^{k} \sum_{\ell \neq m} q\big(\lvert\mathcal{R}_m^{(j-1)}(\mathcal{V}_\ell^{(j)})\rvert + \lvert\mathcal{R}_\ell^{(j-1)}(\mathcal{V}_m^{(j)})\rvert\big)$
	Msgs.	$\sum_{j=1}^{k} \big(\lvert\{\ell \neq m : \lvert\mathcal{R}_m^{(j-1)}(\mathcal{V}_\ell^{(j)})\rvert > 0\}\rvert + \lvert\{\ell \neq m : \lvert\mathcal{R}_\ell^{(j-1)}(\mathcal{V}_m^{(j)})\rvert > 0\}\rvert\big)$
	Mem.	$\mathrm{nz}(A_{\{i : x_i^{(j)} \in \mathcal{V}_m\}}) + q\lvert\mathcal{V}_m\rvert$
PA1	Flops	$\sum_{\substack{x_i^{(j)} \in \mathcal{R}(\mathcal{V}_m) \\ 1 \le j \le k}} q(2\lvert\mathrm{nz}(A_i)\rvert - 1)$
	Words	$\sum_{\ell \neq m} q\big(\lvert\mathcal{R}_m^{(0)}(\mathcal{V}_\ell^{(k)})\rvert + \lvert\mathcal{R}_\ell^{(0)}(\mathcal{V}_m^{(k)})\rvert\big)$
	Msgs.	$\lvert\{\ell \neq m : \lvert\mathcal{R}_m^{(0)}(\mathcal{V}_\ell^{(k)})\rvert > 0\}\rvert + \lvert\{\ell \neq m : \lvert\mathcal{R}_\ell^{(0)}(\mathcal{V}_m^{(k)})\rvert > 0\}\rvert$
	Mem.	$\mathrm{nz}(A_{\{i : x_i^{(j)} \in \mathcal{R}(\mathcal{V}_m)\}}) + q\lvert\mathcal{R}(\mathcal{V}_m)\rvert$

Table 1: Complexity of PA0 and PA1 for processor m, in terms of vertices of the computational graph $\mathcal{G}$.

Algorithm 1 PA0. Code for processor m.

for $j = 1, \ldots, k$ **do**
  for all processors $\ell \neq m$ **do**
    Send $x_i^{(j-1)} \in \mathcal{R}_m^{(j-1)}(\mathcal{V}_\ell^{(j)})$ to processor ℓ.
    Receive $x_i^{(j-1)} \in \mathcal{R}_\ell^{(j-1)}(\mathcal{V}_m^{(j)})$ from processor ℓ.
  end for
  Compute $x_i^{(j)} \in \mathcal{V}_m^{(j)}$ via recurrence (3).
end for

Algorithm 2 PA1. Code for processor m.

for all processors $\ell \neq m$ **do**
  Send $x_i^{(0)} \in \mathcal{R}_m^{(0)}(\mathcal{V}_\ell^{(k)})$ to processor ℓ.
  Receive $x_i^{(0)} \in \mathcal{R}_\ell^{(0)}(\mathcal{V}_m^{(k)})$ from processor ℓ.
end for
for $j = 1, \ldots, k$ **do**
  Compute $x_i^{(j)} \in \mathcal{R}^{(j)}(\mathcal{V}_m)$ via recurrence (3).
end for

3.2 Parallel Blocking Covers Algorithm

Consider computing (2) with a dense matrix A. As before, PA0 must communicate at every step, but now the cost of PA1 may be much worse: when $k > 1$, every processor needs all n rows of A and $x^{(0)}$, and so there is no parallelism in computing all but the last SpMV operation. (Note that when $k = 1$, PA1 degenerates to PA0.) If A can be split in the form $D + USV^H$, where D is well-partitioned and USV^H has low rank, we can use a generalization of the blocking covers approach [7] to recover parallelism; we call this algorithm PA1-BC.

Split the matrix $A =: D + USV^H$, and write

$$x^{(j)} = ((D - \alpha_k)x^{(j-1)} - \beta_{j-2}x^{(j-2)} + USV^H x^{(j-1)})/\gamma_{j-1}. \tag{5}$$

Now we manipulate the three-term recurrence (3) to obtain the following identity, which we will then apply to PA1 to obtain the PA1-BC (the blocking covers algorithm). Let $p_j^i(z)$ represent a degree-j polynomial related to $p_j(z)$ by reindexing the coefficients $(\alpha_j, \beta_j, \gamma_j) \coloneqq (\alpha_{i+j}, \beta_{i+j}, \gamma_{i+j})$ in (3).

Lemma 3.1. *Given the additive splitting $z = z_1 + z_2$, (3) can be rewritten as*

$$\begin{aligned} &p_0(z) = p_0(z_1), \quad p_1(z) = p_1(z_1) + z_2 p_0(z)/\gamma_0, \quad \text{and for } j > 1, \\ &p_j(z) = p_j(z_1) + \sum_{i=1}^{j} p_{i-1}^{j-i+1}(z_1) z_2 p_{j-i}(z)/\gamma_{j-i}. \end{aligned} \tag{6}$$

Proof. The result is readily established for $j = 0$ and $j = 1$; supposing it holds up through $j = t$,

$$\begin{aligned} p_{t+1}(z) &= ((z - \alpha_t)p_t(z) - \beta_{t-1}p_{t-1}(z))/\gamma_t \\ &= p_{t+1}(z_1) + z_2 p_t(z)/\gamma_t + \frac{1}{\gamma_t}\sum_{i=1}^{t}(z_1 - \alpha_t)p_{i-1}^{t-i+1}(z_1) z_2 p_{t-i}(z)/\gamma_{t-i} \\ &\quad - \frac{\beta_{t-1}}{\gamma_t}\sum_{i=1}^{t-1} p_{i-1}^{t-i}(z_1) z_2 p_{t-i-1}(z)/\gamma_{t-i-1} \end{aligned}$$

Observe the following identities (for any $j \geq 0$ and z): $p_0^j(z) = 1$, and

$$(z - \alpha_{i+j-1})p_{i-1}^j(z) = \begin{cases} \gamma_{i+j-1} p_i^j(z) & i = 1 \\ \gamma_{i+j-1} p_i^j(z) + \beta_{i+j-2} p_{i-2}^j(z) & i > 1 \end{cases}$$

Substituting these identities, we obtain

$$\begin{aligned} p_{t+1}(z) &= p_{t+1}(z_1) + p_0^{t+1}(z_1) z_2 p_t(z)/\gamma_t + \sum_{i=1}^{t} p_i^{t-i+1}(z_1) z_2 p_{t-i}(z)/\gamma_{t-i} \\ &= p_{t+1}(z_1) + \sum_{i=1}^{t+1} p_{i-1}^{t-i+2}(z_1) z_2 p_{t-i+1}(z)/\gamma_{t-i+1} \end{aligned}$$

This completes the induction. □

Now substitute $z \coloneqq A = D + USV^H =: z_1 + z_2$ in (6), premultiply by SV^H, and postmultiply by x, to obtain

$$SV^H p_j(A)x = S\Big(V^H p_j(D)x + \sum_{i=1}^{j}(V^H p_{i-1}^{j-i+1}(D)U)(SV^H p_{j-i}(A)x/\gamma_{j-i}\Big). \tag{7}$$

To simplify (7), we define

$$W_i \coloneqq V^H p_i(D)U \quad \text{for } 0 \leq i \leq k-2, \tag{8}$$

$$y_i \coloneqq V^H p_i(D)x \quad \text{for } 0 \leq i \leq k-1, \tag{9}$$

$$b_j \coloneqq SV^H x^{(j)} \quad \text{for } 0 \leq j \leq k-1, \tag{10}$$

and rewrite the polynomials p_i^j in terms of the original polynomials $p_i = p_i^0$, via the following result.

Lemma 3.2. *There exist coefficient vectors $w_i^j \in \mathbb{C}^{i+1}$ satisfying*

$$[W_0, \ldots, W_i](w_i^j \otimes I_{r,r}) \coloneqq V^H p_i^j(D)U \tag{11}$$

for $0 \le i \le k-2, 1 \le j \le k-i-1$, and they can be computed by the recurrence

$$\begin{aligned} w_0^j &\coloneqq 1, \quad w_1^j \coloneqq (T_1 - \alpha_{i+j-1} I_{2,1}) w_0^j / \gamma_{j-1}, \quad \text{and for } i > 1, \\ w_i^j &\coloneqq ((T_i - \alpha_{i+j-1} I_{i+1,i}) w_{i-1}^j - \beta_{i+j-2} I_{i+1,i-1} w_{i-2}^j) / \gamma_{i+j-1}, \end{aligned} \tag{12}$$

where $I_{r,c}$ denotes the leading r-by-c submatrix of an identity matrix.

Proof. For any j, we verify the claim for $i = 0$ and $i = 1$; supposing it holds up to some $i = t$. Write

$$V^H((D - \alpha_{t+j}) p_t^j(D) - \beta_{t+j-1} p_{t-1}^j(D))U / \gamma_{t+j},$$

then substitute

$$p_i^j(D)U = [p_0(D)U, \ldots, p_i(D)U](w_i^j \otimes I_{r,r})$$

for $i = t-1$ and $i = t$. The conclusion follows from rewriting (3) as

$$V^H D[p_0(D)U, \ldots, p_{i-1}(D)U] = V^H[p_0(D)U, \ldots, p_i(D)U](T_i \otimes I_{r,r})$$

(for any i), and substituting. □

Substituting (8), (9), (10), and (11) into (7), we obtain

$$b_j = S\Big(y_j + \sum\nolimits_{i=1}^{j} [W_0, \ldots, W_{i-1}] \cdot (w_{i-1}^{j-i+1} \otimes I_{r,r}) b_{j-i} / \gamma_{j-i}\Big). \tag{13}$$

Ultimately we must evaluate (5), which we rewrite as

$$x^{(j)} = ((D - \alpha_{j-1}) x^{(j-1)} - \beta_{j-2} x^{(j-2)} + U b_{j-1}) / \gamma_{j-1}. \tag{14}$$

Computing $[x^{(0)}, \ldots, x^{(k)}]$ by this recurrence can be accomplished by applying PA1 to the following recurrence for polynomials $p_j(z, c)$, where the sequence of indeterminates $c \coloneqq (c_0, \ldots, c_{j-1}, \ldots)$ represents the terms $(Ub_0, \ldots, Ub_{j-1}, \ldots)$:

$$\begin{aligned} p_0(z, c) &\coloneqq 1, \; p_1(z, c) \coloneqq ((z - \alpha_0) p_0(z, c) + c_0)/\gamma_0, \quad \text{and for } j > 1, \\ p_j(z, c) &\coloneqq ((z - \alpha_{j-1}) p_{j-1}(z, c) - \beta_{j-2} p_{j-2}(z, c) + c_{j-1})/\gamma_{j-1}. \end{aligned} \tag{15}$$

Given the notation established, we construct PA1-BC (Alg. 3).

We redefine $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in terms of the graph of D (as opposed to that of A, which may be complete). Processor m must own the following rows of D, U, V, and $x^{(0)}$:

$$D_{\{i : x_i^{(1)} \in \mathcal{R}(\mathcal{V}_m)\}}, \; U_{\{i : x_i^{(1)} \in \mathcal{R}(\mathcal{V}_m)\}}, \; V_{\{i : x_i^{(j)} \in \mathcal{V}_m\}}, \text{ and } \mathcal{R}^{(0)}(\mathcal{V}_m) = x^{(0)}_{\{i : x_i^{(0)} \in \mathcal{R}(\mathcal{V}_m)\}},$$

in order to compute the entries $x_i^{(j)} \in \mathcal{V}_m$.

Algorithm 3 PA1-BC. Code for processor m.

1: Compute local rows of $K_{k-1}(D, U, T_{k-2})$ with PA1, premultiply by local columns of V^H.
2: Compute $[W_0, \ldots, W_{k-2}]$ by an Allreduce.
3: Compute w_i^j for $0 \le i \le k-2$ and $1 \le j \le k-i-1$, via (12).
4: Compute local rows of $K_k(D, x^{(0)}, T_{k-1})$ with PA1, premultiply by local columns of V^H.
5: Compute $[y_0, \ldots, y_{k-1}]$ by an Allreduce.
6: Compute $[b_0, \ldots, b_{k-1}]$ by (13).
7: Compute local rows of $[x^{(0)}, \ldots, x^{(k)}]$ with PA1, modified for (14).

In exact arithmetic, PA1-BC returns the same output as PA0. However, by exploiting the splitting $A = D + USV^H$, PA1-BC may overcome the problems of PA1, and allow us to avoid communication when A itself is not well partitioned.

One can think of the sequential blocking covers algorithm in [7] as a special case of a sequential execution of PA1-BC. Given a set of blocker vertices, indexed $\mathcal{I} \subseteq \{1, \ldots, n\}$, the algorithm in [7] removes the outgoing edges from those vertices, i.e., removes rows $\mathcal{I}$ from A, to obtain a well-partitioned matrix D. In our notation, this means setting $U \coloneqq [e_i : i \in \mathcal{I}]$ and $SV^H \coloneqq A_{\mathcal{I}}$, where e_i is the i^{th} identity column.

PA0	Flops	$(8s^2+8s+1)kq(n/p)$
	Words	$4skq(n/p+s)$
	Msgs.	$8k$
	Mem.	$((2s+1)^2+q(k+1))(n/p)+4sq(n/p)^{1/2}+4s^2q$
PA1	Flops	$(8s^2+8s+1)\big(kq(n/p)+2sk^2q(n/p)^{1/2}+(4/3)s^2k^3q\big)$
	Words	$4skq(n/p+sk)$
	Msgs.	8
	Mem.	$((2s+1)^2+q(k+1))(n/p)+((2s+1)^2+1)\big(4skq(n/p)^{1/2}+4s^2k^2q\big)$

Table 2: Complexity comparison of PA0 and PA1 for computing $[Dx,\ldots,D^kx]$ on p processors, when D is a $(2s+1)^2$-point stencil on an $n^{1/2}$-by-$n^{1/2}$ mesh. These results are modified from [4, Table 1] to allow for x to have $q>1$ columns. When computing the 3-term recurrence (3) resp. (15), the flop costs for both algorithms increase by additive factors of $(5k-7)q(n/p)$ resp. $(6k-8)q(n/p)$.

PA0	Flops	$\mathrm{F}_{\mathrm{PA0(3)}}(k,q)+kqr(4(n/p)+p)$
	Words	$\mathrm{W}_{\mathrm{PA0}}(k,q)+2kqrp$
	Msgs.	$\mathrm{S}_{\mathrm{PA0}}(k,q)+k\lg(p)$
	Mem.	$\mathrm{M}_{\mathrm{PA0}}(k,q)+2r(n/p)$
PA1-BC (offline)	Flops	$\mathrm{F}_{\mathrm{PA1(3)}}(k-2,r)+kr^2(2(n/p)+p)$
	Words	$\mathrm{W}_{\mathrm{PA1}}(k-2,r)+2kr^2p$
	Msgs.	$\mathrm{S}_{\mathrm{PA1}}(k-2,r)+\lg(p)$
	Mem.	$\mathrm{M}_{\mathrm{PA1}}(k-2,r)+r(n/p)$
PA1-BC (online)	Flops	$\mathrm{F}_{\mathrm{PA1(3)}}(k-1,q)+\mathrm{F}_{\mathrm{PA1(3c)}}(k,q)+kqr\big(4(n/p)+p+O(sk(n/p)^{1/2})\big)+k^2(k+q)r^2$
	Words	$\mathrm{W}_{\mathrm{PA1}}(k-1,q)+\mathrm{W}_{\mathrm{PA1}}(k,q)+2kqrp$
	Msgs.	$\mathrm{S}_{\mathrm{PA1}}(k-1,q)+\mathrm{S}_{\mathrm{PA1}}(k,q)+\lg(p)$
	Mem.	$\mathrm{M}_{\mathrm{PA1}}(k-1,q)+\mathrm{M}_{\mathrm{PA1}}(k,q)+r(n/p)+O(skr(n/p)^{1/2})$

Table 3: Complexity comparison for 'stencil plus rank-r' example, showing leading order constant factors. 'Offline' refers to Lines 1–3 and 'Online' refers to Lines 4–7 of PA1-BC. The functions F, W, S, M denote flops, words, messages, and memory according to Table 2; the '(3)' resp. '(3c)' suffixes in the subscripts of F indicate the 3-term recurrences (3) resp. (15) (see caption of Table 2).

3.3 Complexity Analysis for a Model Problem

To demonstrate the potential performance benefits of PA1-BC over PA1 and PA0, we consider the following example. Let $A=D+UV^H$ be a dense $n\times n$ matrix, where the graph of D is a $(2s+1)^2$-point stencil on a $n^{1/2}$-by-$n^{1/2}$ mesh, and U and V are dense n-by-r matrices. We assume p and n are perfect squares and $p^{1/2}$ divides $n^{1/2}$, and we partition the vertices $\{1,\ldots,n\}$ so that each processor owns a $(n/p)^{1/2}$-by-$(n/p)^{1/2}$ square of the domain. Following [4], we assume $k<(n/p)^{1/2}$, so that PA1 (applied to D) need only communicate among neighboring processors. We give the complexity for PA0 and PA1 in Table 2, modified from [4, Table 1] to allow x to have $q>1$ columns, and to use the three-term recurrences (3) and (15).

We assume our collective communications attain the lower bounds in [2, Table 1]. We consider Lines 1-3 of PA1-BC as 'offline' operations, and Lines 4-7 as 'online' computations, since typically the matrix A and the polynomials p_j are fixed across many calls to PA1-BC (or PA0/1), while the input matrix x changes on each invocation. We compare PA0 and PA1-BC in Table 3.

We use the following arithmetic counts: $\mathrm{apply}(S,n_1,n_2)=0$ is the cost of applying S to an n_1-by-n_2 matrix (in this example S is the identity), $\mathrm{add}(n_1,n_2)=\mathrm{scal}(n_1,n_2)=n_1n_2$ is the cost of adding two n_1-by-n_2 matrices or multiplying one by a scalar, and $\mathrm{mm}(n_1,n_2,n_3)=2n_1n_2n_3-n_1n_3$ is the cost of multiplying matrices of size n_1-by-n_2 and n_2-by-n_3.

PA0 We modify PA0 slightly to exploit the splitting $Ax=Dx+U(V^Hx)$ to save computation and communication. We assign each processor m a block row (indices $\{i:x_i^{(j)}\in\mathcal{V}_m\}$) of D, U, V, and $x^{(0)}$ — a nonoverlapping partition of each, as opposed to PA1, whose partitions may overlap when $k>1$. We first

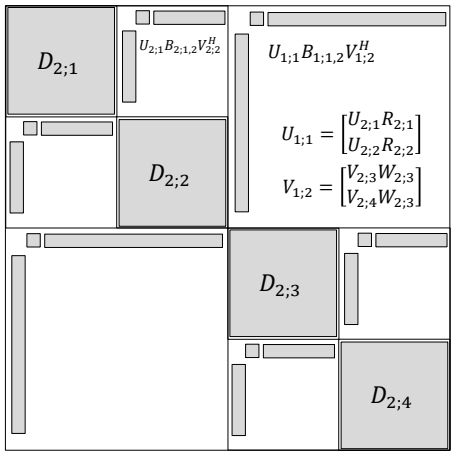

Figure 1: An example of a hierarchical semiseparable matrix, with $\ell = 2$.

compute $y = V^H x$ via a local matrix multiplication (costing $\mathrm{mm}(r, n/p, q)$ flops) followed by an Allreduce of y ($(p-1)rq$ flops, $2(p-1)rq$ words, and $\lg(p)$ messages). Then we compute Uy by a local matrix multiplication (costing $\mathrm{mm}(n/p, r, q)$ flops). Then we communicate rows of $x^{(j)}$ in order to compute the local rows of $Dx^{(j)}$; the costs of this step follow from Table 2.

PA1-BC PA1-BC makes 3 calls to PA1, in Lines 1, 4, and 7. These costs are shown in Table 2 (note that the third PA1 call uses the PA1(3c) variant).

Now we analyze the applications of V^H (Lines 1 and 4) and subsequent Allreduce collectives (Lines 2 and 5). Applying the local columns of V^H costs $\mathrm{mm}(r, n/p, (k-1)r)$ (resp. $\mathrm{mm}(r, n/p, kq)$) flops, and the subsequent Allreduce costs $(p-1)(k-1)r^2$ flops and $2(p-1)(k-1)r^2$ words (resp. $(p-1)krq$ flops and $2(p-1)krq$ words), and $\lg(p)$ messages (both).

Computing w_i^j in Line 3 costs

$$(3k-2) + \sum_{i=2}^{k-2} \sum_{j=1}^{k-i-1} 8i + 5 = 4/3k^3 + O(1)$$

flops (performed locally), and computing b_0 through b_{k-1} in Line 6 costs

$$\sum_{j=0}^{k-1} \left(\mathrm{apply}(S, r, q) + j \cdot \mathrm{add}(r, q) + \sum_{i=1}^{j} \left(\mathrm{mm}(r, r, q) + \mathrm{scal}(r, q) + i \cdot \mathrm{scal}(r, r) + (i-1)\mathrm{add}(r, r) \right) \right) < k^2(k+q)r^2$$

local flops. Note that if T_k is Toeplitz (e.g., $\{p_j\}$ are monomials), then we do not need to compute w_i^j, as each is an identity column; furthermore, the upper bound $k^2(k+q)r^2$ above reduces to $k^2(O(1)+q)r^2$ flops.

Lastly, computing $U[b_0, \ldots, b_{k-1}]$ in Line 7 requires a local computation of

$$\mathrm{mm}(n/p + 4sk(n/p)^{1/2} + 4s^2k^2, r, kq) < 2kqr(n/p) + 8sk^2qr(n/p)^{1/2} + O(s^2k^3qr)$$

flops.

4 Extension to Hierarchical Matrices

Hierarchical ($\mathcal{H}$-) matrices (mentioned above) are amenable to the splitting $D + USV^H$. Particularly interesting are a special class of $\mathcal{H}$-matrices called hierarchical semiseparable (HSS) matrices. We briefly review the HSS representation, using the notation in [3] (see also Fig. 1). For any $L \in \{0, \ldots, \lfloor \lg n \rfloor\}$, we

can write A hierarchically by recursively defining its diagonal blocks as $A =: D_{0;i}$ and for $1 \le \ell \le L$ and $1 \le i \le 2^{\ell-1}$,

$$D_{\ell-1;i} =: \begin{bmatrix} D_{\ell;2i-1} & U_{\ell;2i-1}B_{\ell;2i-1,2i}V_{\ell;2i}^H \\ U_{\ell;2i}B_{\ell;2i,2i-1}V_{\ell;2i-1}^H & D_{\ell;2i} \end{bmatrix},$$

whose off-diagonal blocks are defined recursively by

$$U_{\ell-1;i} =: \begin{bmatrix} U_{\ell;2i-1}R_{\ell;2i-1} \\ U_{\ell;2i}R_{\ell;2i} \end{bmatrix} \quad \text{and} \quad V_{\ell-1;i} =: \begin{bmatrix} V_{\ell;2i-1}W_{\ell;2i-1} \\ V_{\ell;2i}W_{\ell;2i} \end{bmatrix}$$

for $2 \le \ell \le L$ and $1 \le i \le 2^{\ell-1}$, and of course are empty at level 0: $U_{0;1}, V_{0;1} := []$. The subscript ℓ indicates the level in a perfect binary tree of depth L, and i indexes the 2^ℓ vertices at depth ℓ.

The action of A on a matrix x, i.e., $v := Ax$, satisfies $v_{0;1} = D_{0;1}x_{0;1}$, and for $1 \le \ell \le L$ and $1 \le i \le 2^\ell$, satisfies $v_{\ell;i} = D_{\ell;i}x_{\ell;i} + U_{\ell;i}f_{\ell;i}$, with $f_{1;1} = B_{1;1;2}g_{1;2}$, $f_{1;2} = B_{1;2;1}g_{1;1}$, and for $1 \le \ell \le L-1$ and $1 \le i \le 2^\ell$,

$$f_{\ell+1;2i-1} = \begin{bmatrix} R_{\ell+1;2i-1}^T \\ B_{\ell+1;2i-1,2i}^T \end{bmatrix}^T \begin{bmatrix} f_{\ell;i} \\ g_{\ell+1;2i} \end{bmatrix} \quad \text{and} \quad f_{\ell+1;2i} = \begin{bmatrix} R_{\ell+1;2i}^T \\ B_{\ell+1;2i,2i-1}^T \end{bmatrix}^T \begin{bmatrix} f_{\ell;i} \\ g_{\ell+1;2i-1} \end{bmatrix},$$

where for $1 \le \ell \le L-1$ and $1 \le i \le 2^\ell$, $g_{\ell;i} = \begin{bmatrix} W_{\ell+1;2i-1} \\ W_{\ell+1;2i} \end{bmatrix}^H \begin{bmatrix} g_{\ell+1;2i-1} \\ g_{\ell+1;2i} \end{bmatrix}$, and $g_{L;i} = V_{L;i}^H x_{L;i}$ for $1 \le i \le 2^L$. For any level ℓ we assemble the block diagonal matrices

$$U_\ell := \bigoplus_{i=1}^{2^\ell} U_{\ell;i}, \qquad V := \bigoplus_{i=1}^{2^\ell} V_{\ell;i}, \qquad D_\ell := \bigoplus_{i=1}^{2^\ell} D_{\ell;i}, \tag{16}$$

denoted here as direct sums of their diagonal blocks. We also define matrices S_ℓ, representing the recurrences for $f_{\ell;i}$ and $g_{\ell;i}$, satisfying

$$v = Ax =: D_\ell x + U_\ell S_\ell V_\ell^H x. \tag{17}$$

We will now discuss parallelizing $v = Ax$, to generalize PA0 and PA1 to HSS matrices. We make the following assumptions for the subsequent sections. The matrix A is n-by-n, and its HSS representation has a perfect binary tree structure to some level $L > 2$. We have $p \ge 4$ processors, and for simplicity assume that p is a power of 2. For each processor $m \in \{0, 1, \ldots, p-1\}$, let L_m denote the smallest level $\ell \ge 1$ such that $p/2^\ell$ divides m. We also define the intermediate level $1 < L_p := \lg(p) \le L$ of the HSS tree; each $L_m \ge L_p$, where equality is attained when m is odd.

4.1 PA0 for HSS Matrices

Given the notation and assumptions in the preceding section, we show how to modify PA0 when A is HSS, exploiting the $v = Ax$ recurrences for each $1 \le j \le k$ — the result is called PA0-HSS. For clarity, we write the parallel upsweep/downsweep subroutine separately, in Alg. 5.

Our PA0-HSS algorithm is based on the serial algorithm for HSS matrix-vector multiplication from [3], summarized in the recurrences in the preceding section. We are unaware of previous work demonstrating a parallelization of these recurrences, although it is obvious, given the perfect binary tree structure. First, on the upsweep, each processor locally computes $V_{L_p}^H x$ (its subtree, rooted at level $L_p = \lg(p)$) and then performs L_p steps of parallel reduction, until there are two processors active, and then a downsweep until level L_p, at which point each processor is active, owns $S_{L_p}V_{L_p}^H x$, and recurses into its local subtree to finally compute its rows of $v = D_L x + U_L S_L V_L x$. More precisely, we assign processor m the computations

$$f_{\ell;i} \text{ and } g_{\ell;i} \text{ for } \left\{\ell, i : {}_{2^\ell m/p+1 \le i \le 2^\ell(m+1)/p}^{L \ge \ell \ge L_p}\right\} \text{ and for } \left\{\ell, i : {}_{i=2^\ell m/p+1}^{L_p - 1 \ge \ell \ge L_m}\right\}$$

(so only processors 0 and $p/2$ are active at the top level ($\ell = 1$)), and the matrices D_L, U_L, and V_L are distributed contiguously block rowwise, so processor m stores blocks $D_{L_p;m+1}$, $U_{L_p;m+1}$, and $V_{L_p;m+1}$. The $R_{\ell;i}$, $W_{\ell;i}$, and $B_{\ell;i}$ matrices are distributed so that they are available for the computations in the upsweep/downsweep (Alg. 5); we will omit further details for brevity, but will analyze the memory requirements when we compare with PA1-HSS, below.

Algorithm 4 PA0-HSS. Code for processor m.

for $j = 1, \ldots, k$ **do**
  Perform upsweep and downsweep (Alg. 5) to compute local rows of $Ax^{(j-1)}$.
  Compute $x_i^{(j)} \in \mathcal{V}_m^{(j)}$ locally.
end for

Algorithm 5 Upsweep and downsweep for PA0-HSS. Code for processor m.

UPSWEEP:
for $\ell = L, L-1, \ldots, L_p$ **do**
  for $i = 2^\ell m/p + 1, \ldots, 2^\ell(m+1)/p$ **do**
    Compute $g_{\ell;i}$
  end for
end for
for $\ell = L_p - 1, \ldots, 1, 0$ **do**
  if $p/2^\ell$ divides m **then**
    $i = 2^\ell m/p + 1$.
    Send $g_{\ell+1;2i-1}$ to proc. $m + p/2^{\ell+1}$.
    Receive $g_{\ell+1;2i}$ from proc. $m + p/2^{\ell+1}$.
    if $\ell > 0$ **then**
      Compute $g_{\ell;i}$.
    end if
  else if $p/2^{\ell+1}$ divides m **then**
    $i = 2^\ell m/p + 1/2$.
    Send $g_{\ell+1;2i}$ to proc. $m - p/2^{\ell+1}$.
    Receive $g_{\ell+1;2i-1}$ from proc. $m - p/2^{\ell+1}$.
  end if
end for
DOWNSWEEP:
for $\ell = 0, \ldots, L_p - 1$ **do**
  if $p/2^\ell$ divides m **then**
    $i = 2^\ell m/p + 1$.
    Compute $f_{\ell+1;2i-1}$.
    if $\ell < L_p - 1$ **then**
      Send $f_{\ell+1;2i-1}$ to proc. $m + p/2^{\ell+2}$.
    end if
  else if $p/2^{\ell+1}$ divides m **then**
    $i = 2^\ell m/p + 1/2$.
    if $\ell > 0$ **then**
      Receive $f_{\ell;i}$ from proc. $m - p/2^{\ell+1}$.
    end if
    Compute $f_{\ell+1;2i}$.
    if $\ell < L_p - 1$ **then**
      Send $f_{\ell+1;2i}$ to proc. $m + p/2^{\ell+2}$.
    end if
  end if
end for
for $\ell = L_p, \ldots, L - 1$ **do**
  for $i = 2^\ell m/p + 1, \ldots, 2^\ell(m+1)/p$ **do**
    Compute $f_{\ell+1;2i-1}$ and $f_{\ell+1;2i}$.
  end for
end for
Compute $v_{L;i}$ for i=$mn/(pr)$+1, $\ldots$, $(m$+1$)n/(pr)$.

4.2 PA1 for HSS Matrices

The block-diagonal structure of D_ℓ, U_ℓ, and V_ℓ in (16) suggests an efficient parallel implementation of PA1-BC, which we present as PA1-HSS (Alg. 6). As opposed to PA0-HSS, the only parallel communication in PA1-HSS occurs in two Allgather operations, in Lines 1 and 5. The computation cost increases, however, since each processor performs the *entire* upsweep/downsweep between levels 1 and L_p locally. Recall in PA0-HSS, there was parallelism, albeit less than full, during these levels (Lines 7–39 of Alg. 5). Our further loss of parallelism shows up in our complexity analysis (see Table 4) as a factor of p, compared to a factor of $\lg(p)$ in PA0-HSS; we also illustrate this tradeoff in our performance modeling (see §5).

We assume the same data layout as PA0-HSS: each processor stores a diagonal block of D_{L_p}, U_{L_p}, and V_{L_p} (but only stores the smaller blocks of level L). We assume each processor is able to apply S_{L_p}. We rewrite (14) for the local rows, and exploit the block diagonal structure of D_{L_p} and U_{L_p}, to write

$$x^{(j)}_{L_p,m+1} = \Big((D_{L_p,m+1} - \alpha_{j-1})x^{(j-1)}_{L_p,m+1} - \beta_{j-2}x^{(j-2)}_{L_p,m+1} + U_{L_p,m+1}(b_{j-1})_{\{mr+1,\ldots,(m+1)r\}}\Big)/\gamma_{j-1}. \quad (18)$$

We will not exploit the fact that each processor ultimately needs only a subset of the rows of b_j computed in Line 6. Each processor locally computes all rows of $b_j = S_{L_p}V^H_{L_p}x^{(j)} = S_{L_p} \cdot z$, where z is the maximal parenthesized term in (13), using the HSS recurrences:

$$V^H_{L_p}x^{(j)} = z =: \begin{bmatrix} g^T_{L_p,1} & \cdots & g^T_{L_p,p} \end{bmatrix}^T \mapsto \begin{bmatrix} f^T_{L_p,1} & \cdots & f^T_{L_p,p} \end{bmatrix}^T := b_j = S_{L_p}V^H_{L_p}x^{(j)}.$$

The rest of PA1-HSS is similar to PA1-BC, except that the Allreduce operations have now been replaced by Allgather operations, to exploit the block structures of W_i and y_i.

Algorithm 6 PA1-HSS (Blocking Covers). Code for processor m.

1: Compute $K_{k-1}(D_{L_p;m+1}, U_{L_p;m+1}, T_{k-2})$, premultiply by $V^H_{L_p;m+1}$.
2: Compute $[W_0, \ldots, W_{k-2}]$ by an Allgather.
3: Compute w^j_i for $0 \le i \le k-2$, and $1 \le j \le k-i-1$, via (12).
4: Compute $K_k(D_{L_p;m+1}, x^{(0)}_{L_p;m+1}, T_{k-1})$, premultiply by $V^H_{L_p;m+1}$.
5: Compute $[y_0, \ldots, y_{k-1}]$ by an Allgather.
6: Compute $[b_0, \ldots, b_{k-1}]$ by (13), where $S = S_{L_p}$ is applied as described above.
7: Compute local rows of $[x^{(0)}, \ldots, x^{(k)}]$ according to (18).

4.3 Complexity Analysis for a Model Problem

We compare the asymptotic complexity of PA0-HSS and PA1-HSS. We assume A is n-by-n and dense, and represented by the dense r-by-r matrices $R_{\ell;i}$ and $W_{\ell;i}$ (for levels 2 through L), $B_{\ell;i,i\pm1}$ (for levels 1 through L) and $D_{L;i}$, $U_{L;i}$, and $V_{L;i}$ (i.e., at the leaf level). To simplify the presentation, we assume n and r are powers of 2 and $L = \lg(n/r)$. A matrix satisfying these assumptions is said to have *HSS rank* r; here we assume A is already represented as such a set of r-by-r matrices. We summarize the results in Table 4.

PA0-HSS Computing the local portion of the upsweep and downsweep (levels L through L_p, i.e., Lines 2–6,40–45 of Alg. 5) costs

$$(2^L/p)\mathrm{mm}(r,r,q) + \sum_{\ell=L_p}^{L-1} 2(2^\ell/p)\mathrm{mm}(r,2r,q) + (2^L/p)\mathrm{mm}(r,2r,q)$$

flops and no communication. For the intermediate lines, we follow processor 0, who is active on every level. Processor 0 computes one $g_{\ell;i}$ for $\ell = L_p, L_p-1, \ldots, 1$, performing L_p+1 sends and receives of size rq), and then computes one $f_{\ell;i}$ for $\ell = 1, \ldots, L_p$, sending messages (of size rq) $L_p - 1$ times. Thus, $2L_p\mathrm{mm}(r,2r,q)$ flops. The remaining cost is computing the three-term recurrence (locally), which is an additional $(5k-7)q(n/p)$ flops.

PA0-HSS requires enough memory to store the local blocks of D_L, U_L, and V_L, a total of $3rn/p$ words. Furthermore, each processor must store the $R_{\ell;i}$, $W_{\ell;i}$ and $B_{\ell;i,i\pm1}$ matrices for its local upsweep/downsweep, a total of $\sum_{\ell=L_p}^{L}(2^\ell/p)2r^2$ words, and some subset of these matrices for the parallel portion of the upsweep/downsweep. Processor 0 must be able to store $2L_p \cdot 2r^2$ more, an upper bound for the other processors. Lastly, each processor must be able to store their $(k+1)qn/p$ entries of $x^{(0)}, \ldots, x^{(k)}$.

PA1-HSS As opposed to PA1-BC, Lines 1, 4, and 7 of PA1-HSS will not require communication, due to the block diagonal structure of D, U, and V. We assume all three lines are implemented to exploit the upsweep/downsweep recurrences for (locally) applying the (HSS rank r) matrix $D_{L_p;m+1}$ to an n/p-by-r matrix (Line 1) or to an n/p-by-q matrix (Lines 4 and 7). (Note that in Line 1 we could further expand $U_{L_p;m+1}$ into a block diagonal matrix (with r-by-r blocks), using the $R_{\ell,i}$ matrices; however, this seems to only increase the arithmetic cost.) The arithmetic cost for (locally) multiplying $D_{L_p;m+1}$ (with HSS rank r) by a dense n/p-by-q matrix is:

$$\sum_{\ell=L_p+1}^{L} \begin{cases} (2^\ell/p)\mathrm{mm}(r,r,q) & \ell = L \\ (2^\ell/p)\mathrm{mm}(r,2r,q) & \ell < L \end{cases} + \sum_{\ell=L_p+1}^{L} \begin{cases} (2^\ell/p)\mathrm{mm}(r,2r,q) & \ell > L_p+1 \\ (2^\ell/p)\mathrm{mm}(r,r,q) & \ell = L_p+1 \end{cases} + (2^L/p)\mathrm{mm}(r,2r,q)$$
$$< 18qr(n/p) + 20qr^2$$

flops. We apply $D_{L_p;m+1}$ $k-2$, $k-1$, and k times, in lines Line 1, Line 4, and Line 7, resp. Evaluating the three-term recurrences (in the same three lines) increases the cost by $(5k-17)r(n/p)$, $(5k-12)q(n/p)$, and $(6k-8)q(n/p)$ flops, resp. Then applying $V_{L_p;m+1}^H$ in Lines 1 and 4 costs $\mathrm{mm}(r, n/p, r)$ and $\mathrm{mm}(r, n/p, q)$ flops, resp. (note that it would nearly quadruple these costs to apply $V_{L_p;m+1}^H$ using the HSS upsweep recurrences, rather than as a full matrix). The Allreduce operations in PA1-BC have become Allgather operations in Lines 2 and 5, due to the structure and parallel layout of the matrices $W_i = V^H p_i(D)U$ and $y_i = V^H p_i(D)x$. Since we parallelize at HSS level L_p (rather than the leaf level L), each processor distributes $k-1$ r-by-r (resp. k r-by-q) blocks. This costs

$$0 \text{ flops}, \quad (k-1)r^2(p-1) \text{ resp. } kqr(p-1)) \text{ words}, \quad \lg(p) \text{ msgs}$$

Line 3 (performed locally) has the same $O(k^3)$ cost as in PA1-BC, a lower order term. The analysis of Line 6 (also performed locally) is similar to that of PA1-BC, except now r is replaced by pr, except for the terms involving W_i, which is block diagonal with p r-by-r diagonal blocks. In total, each processor performs

$$\sum_{j=0}^{k-1} \Big(\mathrm{apply}(S, pr, q) + j \cdot \mathrm{add}(pr, q) + \sum_{i=1}^{j} \Big(\mathrm{mm}(pr, r, q) + \mathrm{scal}(r, q) + i \cdot \mathrm{scal}(pr, r) + (i-1)\mathrm{add}(pr, r) \Big) \Big)$$

flops, where each application of S to a pr-by-q matrix costs

$$\sum_{\ell=1}^{L_p} 2^\ell \mathrm{mm}(r, 2r, q) + \sum_{\ell=1}^{L_p} \begin{cases} 2^\ell \mathrm{mm}(r,2r,q) & \ell > 1 \\ (2^\ell/p)\mathrm{mm}(r,r,q) & \ell = 1 \end{cases} \quad = \quad 16pqr^2 + O(qr)$$

flops.

The memory requirements for PA1-HSS are the same as PA0-HSS, except that each processor needs all the $R_{\ell;i}$, $W_{\ell;i}$, and $B_{\ell;i}$ matrices for levels 1 through L_p, a cost of $8pr^2$ words.

5 Performance Modelling

We model speedups of our new algorithms for the two example problems discussed in the text: a 2D stencil plus rank-r component, and an HSS matrix. Complexity counts used can be found in Tables 3 and 4, resp. We use two machine models used in [8] – 'Peta,' an 8100 processor petascale machine, and 'Grid,' 125 terascale machines connected via the Internet. The Peta machine has a flop rate of $\gamma = 2 \cdot 10^{-11}$

PA0-HSS	Flops	$kq(14r + O(1))n/p + 8kqr^2 \lg(p)$
	Words	$kq(3r + O(1)) \lg(p)$
	Msgs.	$3k \lg(p)$
	Mem.	$(7r + (k+1)q)n/p + 4r^2 \lg(p)$
PA1-HSS (offline)	Flops	$18kr^2n/p$
	Words	kr^2p
	Msgs.	$\lg(p)$
	Mem.	$(7r + (k+1)q)n/p + 16r^2p)$
PA1-HSS (online)	Flops	$kq(36r + O(1))n/p + (k^2/3)(k + 10q)r^2p$
	Words	$kqrp$
	Msgs.	$\lg(p)$
	Mem.	$(7r + (k+1)q)n/p + 16r^2p$

Table 4: Complexity comparison for parallel HSS algorithms PA0-HSS and PA1-HSS, showing leading order constant factors. 'Offline' refers to Lines 1–3 and 'Online' refers to Lines 4–7 of PA1-HSS.

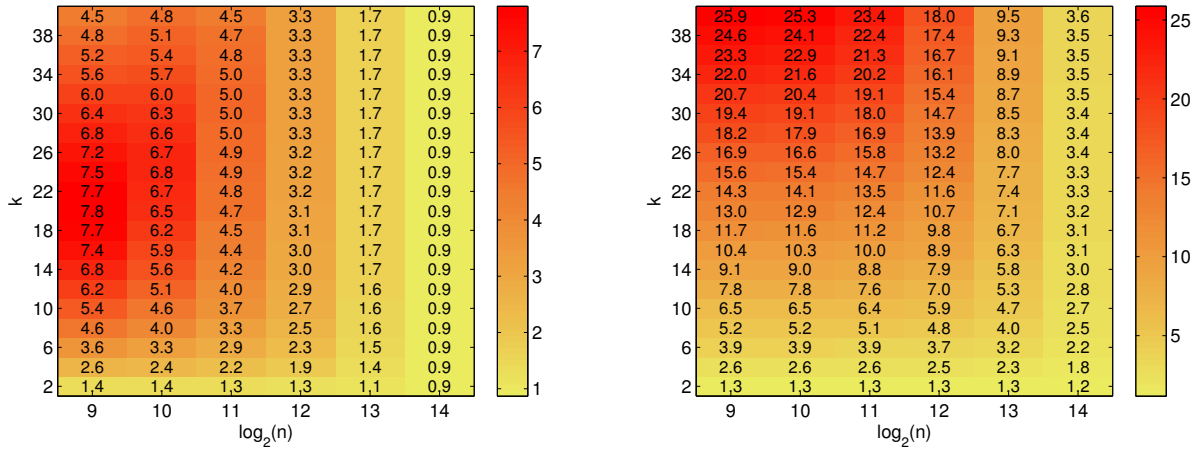

Figure 2: Predicted speedups for model problem: 2D 9-point stencil + rank-1 dense component. Parameters obtained from performance modeling in [8]. *Left*: Speedups for PA1 over PA0 for various k values on Peta. *Right*: Speedups for PA1 over PA0 for various k values on Grid.

s/flop, latency cost $\alpha = 10^{-5}$ s/message, and bandwidth cost $\beta = 2 \cdot 10^{-9}$ s/word. The Grid machine has flop rate $\gamma = 10^{-12}$ s/flop, latency cost $\alpha = 10^{-1}$ s/message, and bandwidth cost $\beta = 25 \cdot 10^{-9}$ s/word.

For the stencil plus rank-r test, we ran the performance model for $q = r = s = 1$, i.e., a 2D 9-point stencil plus rank-1 matrix (times a vector). We picked n and k based on those chosen in [8]; note that the dimension of A is n^2. Fig. 2 shows the best speedup of PA1-BC over PA0 over all p values such that $p \leq \min(p_{\max}, (n/k)^2)$. The p values which resulted in the maximum speedup for this test problem are shown in Fig. 3. On Peta, the largest speedup was 7.8×. Speedups were generally higher on Grid, with a maximum speedup of 25.9× for this range of k. We expect higher speedups on Grid since PA0 is extremely latency bound on this machine. For both models, predicted speedups decrease with increasing n and k due to growing additional flop and bandwidth terms.

Speedups of PA1-HSS over k invocations of PA0-HSS, for both Peta and Grid, are shown in Fig. 4. We use the parameter triplets (n_i, p_i, r_i) where $p = (4, 16, 64, 256, 1024, 4096)$, $n = (2.5, 5, 10, 20, 40, 80) \cdot 10^3$, and $r = (5, 5, 5, 5, 6, 7)$, based on the parameters for parallel HSS performance tests in [10]. Note that for Grid we only use the first 3 parameter triplets since $p_{\max} = 125$. On Grid, PA0-HSS is extremely latency bound, so our $3k$ reduction in latency results in a $3k$× faster algorithm (the extra flop and bandwidth costs are insignificant for these values of k). This is the best we can expect. As shown in Fig. 5, for very large k, the extra terms begin to dominate, and speedups eventually begin to decrease with k. On Peta, we see $O(k)$ speedups for smaller p and k, but as these quantities increase, the expected speedup drops.

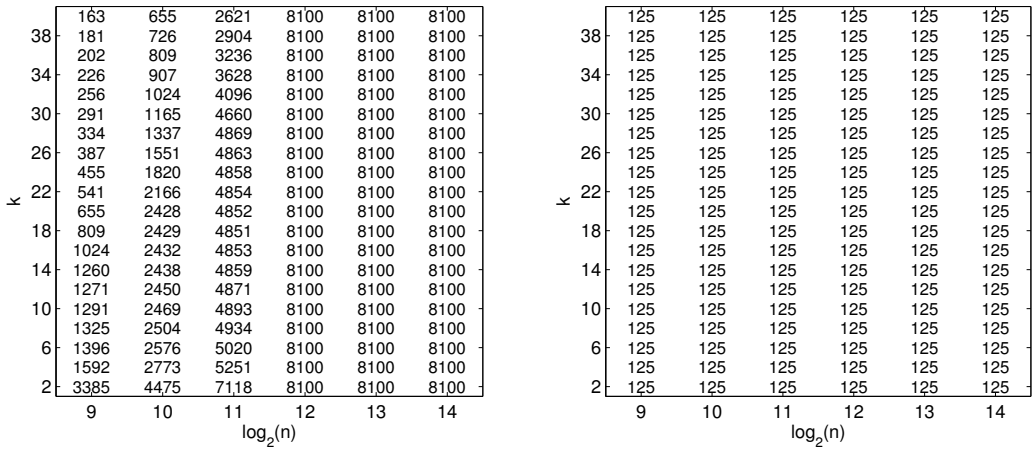

Figure 3: Best p values, used to obtain speedups in Fig. 4. *Left*: Peta machine model, where $p_{\max} = 8100$. *Right*: Grid machine model, where $p_{\max} = 125$.

This is due to the extra factor of $p/\lg(p)$ in the bandwidth cost and the factor of $k^2 p/\lg(p)$ in the flop cost of PA1-HSS. Since the relative latency cost is lower on Peta, the effect of the extra terms becomes apparent for larger k and p.

6 Future Work and Conclusions

In this work, we derive a new parallel communication-avoiding algorithm for the matrix powers computation with $A = D + USV^H$, where D is well partitioned and USV^H has low rank but may not be well partitioned. This is a significant improvement over the algorithms in [4, 8], which require A to be well partitioned. Our approach allows us to exploit the low-rank structure to asymptotically reduce the parallel latency cost: on latency-bound problems, our model indicates an $O(k)$ speedup. We demonstrate the generality of our parallel blocking covers technique by applying it to matrices with hierarchical structure. Detailed performance modeling suggests 24× speedups on petascale machines, and up to $3k$ speedups on extremely latency-bound machines, despite growth in arithmetic and bandwidth costs. Future work includes a high-performance parallel implementation of our matrix powers kernel variants to verify the speedups predicted by performance modeling. We also plan to explore the application of blocking covers to other parallel algorithms.

References

[1] M. Bebendorf. *Hierarchical Matrices*, volume 63. Springer Berlin Heidelberg, 2008.

[2] E. Chan, M. Heimlich, A. Purkayastha, and R. Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[3] S. Chandrasekaran, P. Dewilde, M. Gu, W. Lyons, and T. Pals. A fast solver for HSS representations via sparse matrices. *SIAM J. Matrix Anal. Appl.*, 29(1):67–81, 2006.

[4] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in computing Krylov subspaces. Technical Report UCB/EECS-2007-123, EECS Dept., U.C. Berkeley, Oct 2007.

[5] M. Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, EECS Dept., U.C. Berkeley, 2010.

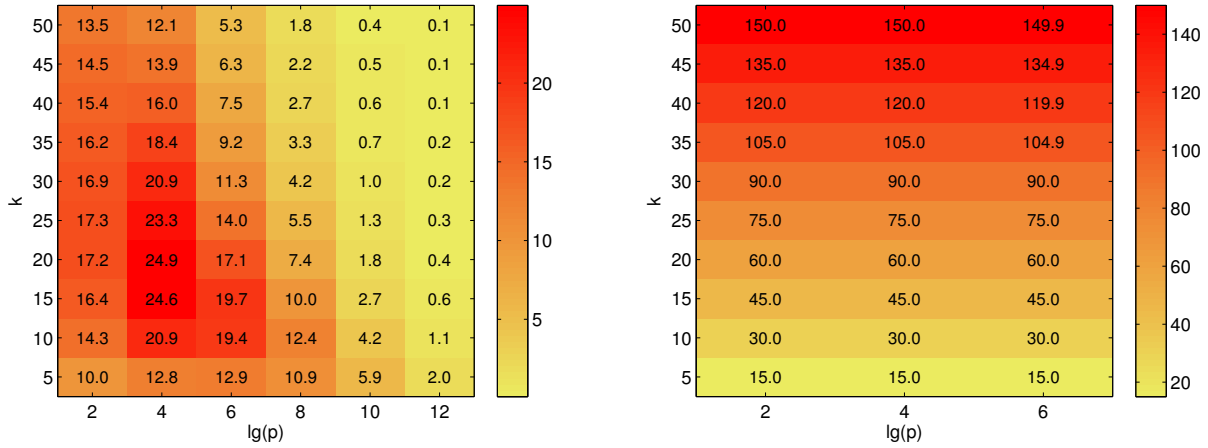

Figure 4: *Left*: Speedups for PA1-HSS over classical algorithm for various k values on Peta. We use the parameters used in parallel runtime tests for 2D problems in [10], where $p = \{4, 16, 64, 256, 1024, 4096\}$, $n = \{2.5, 5, 10, 20, 40, 80\} \times 10^3$, $r = \{5, 5, 5, 5, 6, 7\}$. *Right*: Speedups for PA1-HSS over classical algorithm for various k values on Grid. We use parameters $p = \{4, 16, 64\}$, $n = \{2.5, 5, 10\} \times 10^3$, $r = \{5, 5, 5\}$.

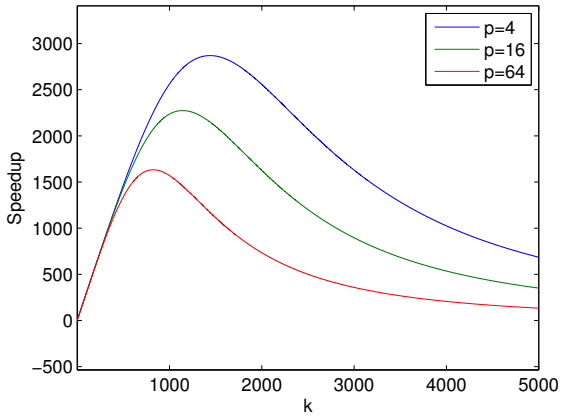

Figure 5: Speedup versus k for $p = \{4, 16, 64\}$, for HSS problem on Grid machine model.

[6] J. Hong and H. Kung. I/O complexity: the red-blue pebble game. In *Proc. 13th Symp. Theory Comp.*, pages 326–333. ACM, 1981.

[7] C. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. *J. Comput. Syst. Sci. Int.*, 54(2):332–344, 1997.

[8] M. Mohiyuddin. *Tuning Hardware and Software for Multiprocessors*. PhD thesis, EECS Dept., U.C. Berkeley, May 2012.

[9] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. Minimizing communication in sparse matrix solvers. In *Proc. ACM/IEEE Conference on Supercomputing*, 2009.

[10] S. Wang, X.S. Li, J. Xia, Y. Situ, and M.V. de Hoop. Efficient scalable algorithms for hierarchically semiseparable matrices. *SIAM J. Sci. Comput.*, 2012. (under review).

[11] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.