Avoiding Communication in Computing Krylov Subspaces

James Demmel
Mark Frederick Hoemmen
Marghoob Mohiyuddin
Katherine A. Yelick

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-123
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-123.html

October 9, 2007

Acknowledgement

The authors wish to acknowledge the contribution from Intel Corporation, Hewlett-Packard Corporation, IBM Corporation, and the National Science Foundation grant EIA-0303575 in making hardware and software available for the CITRIS Cluster which was used in producing these research results. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Avoiding Communication in Computing Krylov Subspaces

James Demmel,[*]Mark Hoemmen,[†]Marghoob Mohiyuddin,[‡]and Katherine Yelick,[§]

August 30, 2007

Abstract

Our goal is to minimize the communication costs of Krylov Subspace Methods (KSMs) to solve either $Ax = b$ or $Ax = \lambda x$, when A is a large sparse matrix. By communication costs we mean both bandwidth and latency costs, either between processors on a parallel computer, or between levels of a memory hierarchy on a sequential computer. As the cost of communication is growing exponentially relative to computation on modern computers, reducing communication is becoming ever more important. It is possible to reduce communication costs to near their theoretical minima: On a parallel computer this means latency costs are *independent* of the dimension k of the Krylov subspace, as opposed to growing proportionally to k for a conventional implementation. On a sequential computer, this mean latency *and* bandwidth costs are independent of k. Achieving this speedup requires a new algorithmic formulation of KSMs.

In this paper we present just part of this new formulation, namely computing a basis of the Krylov subspace spanned by $[x, Ax, A^2x, ...A^kx]$; other papers will present the rest of the KSM formulation. We present theory, performance models, and computational results: Our parallel performance models of 2D and 3D model problems predict speedups over a conventional algorithm of up to 15x on a model Petaflop machine, and up to 22x on a model of the Grid. Our sequential performance models of the same model problems predict speedups over a conventional algorithm of up to 10x on an out-of-core implementation, and up to 2.5x on Intel Clovertown, where we use our ideas to reduce off-chip latency and bandwidth to DRAM. Finally, we measured a speedup of over 3x on an actual out-of-core implementation.

We also consider the related kernel $[Ax, MAx, AMAx, ..., A(MA)^{k-1}x, (MA)^kx]$, which arises in preconditioned KSMs. Under certain mathematical conditions on A and the preconditioner M, we show how to avoid latency and bandwidth for this kernel as well.

[*]Computer Science Division and Mathematics Dept., University of California, Berkeley, CA 94720 (demmel@cs.berkeley.edu).

[†]Computer Science Division, University of California, Berkeley, CA 94720 (mhoemmen@eecs.berkeley.edu).

[‡]EECS Department, University of California, Berkeley, CA 94720 (marghoob@eecs.berkeley.edu).

[§]Computer Science Division, University of California, Berkeley, CA 94720 (yelick@cs.berkeley.edu).

Contents

1 Introduction

Our goal is to minimize the communication costs of Krylov Subspace Methods (KSMs) to solve either $Ax = b$ or $Ax = \lambda x$, when A is a large sparse matrix. Unpreconditioned KSMs seek an optimal solution vector (in a sense that depends on the method) in the Krylov subspace spanned by $[x, Ax, A^2x, ...A^kx]$ (we consider the preconditioned case below). Communication costs include both latency and bandwidth costs: On a parallel computer, the communication cost of a message of n words sent from one processor to another is modeled as the latency plus n divided by the bandwidth. On a sequential computer with a memory hierarchy, the communication cost of moving n (consecutive) words between slow and fast memory is similarly modeled.

Our motivation for minimizing communication costs is that current technology trends show exponentially increasing gaps between computation, bandwidth and latency costs. A recent study [26] of high performance computing shows floating point speeds increasing historically at 59%/year, but interprocessor bandwidth improving only 26%/year, and interprocessor latency improving only 15%/year. Indeed, on certain very large, distributed computing platforms (like the Grid) latencies are already speed-of-light limited and on the order of milliseconds, as opposed to fractions of nanoseconds for floating point operations. Similarly, memory (DRAM) bandwidth is improving only at 23%/year, and memory latency at 5.5%/year. For out-of-core algorithms, with disk bandwidth and latency limited by the rotational speed of disks, the gaps are even larger. Another study [20] observes that latency improves much more slowly than bandwidth across many technologies.

Conventional implementations of KSMs alternate multiplication of the sparse matrix A times a single vector with other vector operations like multiplying vectors by scalars, adding vectors, and dot products. Therefore both the computation and communication cost of k such steps grows (at least) proportionally to k. Our goal is to reorganize KSMs so that the communication cost is (nearly) as small as theoretically possible. On a parallel computer, this means with latency costs that are independent of k. In fact if the sparse matrix has a suitable (and common) sparsity structure described below, we will see that the latency cost of the kernel $[x, Ax, ..., A^kx]$ is just $O(1)$. On a sequential computer, this means that latency and bandwidth costs are independent of k. Said another way, both the matrix A and vectors $[x, Ax, ..., A^kx]$ will need to be moved between fast and slow memory just $1 + o(1)$ times (1 time is obviously the minimum), not k times. Since bandwidth is always the bottleneck in the sequential case, our approach is always an improvement.

Our algorithms apply to general matrices, but they are easiest to understand and analyze for sparse matrices with a suitable (and common) sparsity structure, namely a mesh of a 1D, 2D, or 3D region. More generally, the benefits are largest when it is possible to partition A into row blocks

with a *low surface-to-volume ratio*, i.e. where the components of $y = Ax$ corresponding to each row block depend on few components of x outside the row block. See section 1.1 for details.

In the parallel case, we use the usual mapping where row blocks j of A, x and $y = Ax$ are all assigned to processor j. As stated above, the latency cost of our version of a KSM will be *independent* of k, requiring just 1 message between each processor and its "neighbors", i.e. those processors owning components of x needed to compute the local components of $y = Ax$. The bandwidth and computation costs will be nearly minimal, increasing only by lower order terms (depending on the surface-to-volume ratio) compared to a conventional implementation. For example, suppose that the sparsity pattern of our matrix is that of a 27 point stencil operating on an n-by-n-by-n mesh, and that we assign $\frac{n}{p^{1/3}}$-by-$\frac{n}{p^{1/3}}$-by-$\frac{n}{p^{1/3}}$ "cubes" of mesh points to each of p processors. Then using our approach drops the number of messages per processor from $26k$ to 26, while only increasing the number of words communicated per processor from $\frac{6kn^2}{p^{2/3}}$ to $\frac{6kn^2}{p^{2/3}} \cdot (1 + \frac{2kp^{1/3}}{n})$ and arithmetic operations per processor from $\frac{53kn^3}{p}$ to $\frac{53kn^3}{p} \cdot (1 + \frac{1.5kp^{1/3}}{n})$. In both cases, the factor $\frac{p^{1/3}}{n}$ is proportional to the surface-to-volume ratio, which will be small for problems of interest. (See Table 1 for omitted lower order terms and other details).

In the sequential case, our algorithm will mimic the parallel algorithm, processing the matrix block by block. As stated above, all the data (matrix and vectors) will only have to move from slow to fast memory $1 + o(1)$ times in order to implement k steps of a KSM, not move k times. In other words the number of slow memory accesses (the latency cost) and the bandwidth cost will exceed their minimal values by this $1 + o(1)$ factor. The $o(1)$ term will be proportional to the surface-to-volume ratio. See Table 2 for details.

We contrast our approach of *avoiding* communication with the complementary approach of *overlapping* communication and computation. The latter approach can at best halve the running time, whereas avoiding communication can achieve up to k-fold speedups when communication is dominant. Furthermore, we can use overlapping to accelerate our algorithms as well.

We present both detailed performance models and measurements of an implementation. We model matrices with the sparsity patterns of both a 2D and 3D stencil on a variety of parallel and sequential computers. The two parallel computers modeled are a Petaflop machine consisting of 8100 50 GFlop/s processors connected over a fast network, and a Grid consisting of 125 1 TFlop/s processors connected over the internet. The speedup over a conventional algorithm depends on whether it is a 2D or 3D problem, the width of the stencil (eg 5 point, 9 point etc.), the problem size, and how much computation and communication can be overlapped. We summarize the maximum speedups modeled below for matrices whose graphs are 9 point 2D stencils and 27 point 3D stencils (but stored as general sparse matrices). For Peta, the best speedups were for smaller n in the range studied, because communication was more dominant; maximum speedups fell as n increased and the problem became computation bound. Also for Peta, nonoverlapping computation made latency more important, and so our approach to avoiding latency yielded larger speedups. On the Grid, for the lower n in the range modeled, it was fastest to use just one processor because communication was so expensive. But as n grew, it eventually became effective to use parallelism, and close to this transition point our approach yielded large speedups. Details may be found in Section 5.1:

Machine	Matrix	Range of n	Overlap communication & computation?	Max Modeled Speedup
Peta	2D	2^{10} to 2^{22}	Yes	6.9
			No	15.1
	3D	2^{9} to 2^{14}	Yes	1.02
			No	3.56
Grid	2D	2^{10} to 2^{22}	Yes	22.22
			No	15.63
	3D	2^{9} to 2^{14}	Yes	4.41
			No	7.79

The two "sequential" machines modeled are (1) a uniprocessor with DRAM as fast memory and a single disk as slow memory (called OOC for Out-Of-Core), and (2) the Intel Clovertown multicore chip with on-chip cache as fast memory and DRAM as slow memory. The Clovertown is a parallel machine but here we address how to avoid off-chip latency and bandwidth to DRAM. We only modeled the non-overlapping case, with modeled speedups as shown in the table below. In contrast to the last table, we show the range of speedups attained over all problem sizes n, since bandwidth is always the bottleneck, so significant speedups were attained for all problems sizes. Here, "% Peak" is the ratio of the (modeled) running time of the algorithm on a zero latency / infinite bandwidth machine to the (modeled) true time. The closer this is to 100%, the more completely the algorithm masks the cost of slow memory access. On OOC, we see that we get high speedups, though we are not near peak. On Clovertown our speedups are more modest, but still good, and we are closer to peak. Details may be found in Section 5.2:

Machine	Matrix	Range of n	(Range of) Modeled Speedup	(Range of) % Peak
OOC	2D	2^{14} to 2^{25}	10.2	17%
	3D	2^{8} to 2^{17}	[7.39,9.51]	[14%, 18%]
Clovertown	2D	2^{8} to 2^{19}	[2.45,2.58]	[62%, 65%]
	3D	2^{8} to 2^{12}	[1.34,1.36]	38%

Section 6 describes an actual out-of-core implementation, which achieves a speedup of 3.2x, and is 16% as fast as it would be if run on a machine with zero disk latency and infinite disk bandwidth, up from 5%. This is both a good speedup, and shows that we are within 16% of peak. We also describe our performance model, which uses measured machine parameters and agree with measured performance closely.

In this paper we will only discuss how to achieve these speedups for computing $[Ax, A^2x, ...A^kx]$. The overall KSM will be described in separate papers in preparation. We will actually need to compute the slightly different basis $[p_1(A)x, p_2(A)x, ..., p_k(A)x]$ of the desired Krylov subspace, where $p_j(A)$ is a polynomial in A of degree j, i.e. a linear combination of $x, Ax, ..., A^jx$. This basis, which is important for numerical stability, can easily be implemented using the same techniques described here, but for simplicity we only describe $[Ax, A^2x, ...A^kx]$ in this paper.

Preconditioning is an important technique in KSMs, so we will also describe how to minimize communication cost in the preconditioned basis $[Ax, MAx, AMAx, ..., A(MA)^{k-1}x, (MA)^kx]$, where M is the preconditioner. Our ability to minimize communication depends not just on the sparsity structure of A but on M. It turns out that for a large class of preconditioners, including but not limited to H-matrices [5], it is also possible to drastically reduce communication costs.

The rest of this paper is organized as follows. Section 2 describes our new parallel algorithms, the simplest one (PA1) and then a more complicated one that reduces the surface-to-volume overhead by a factor of 2 (PA2). PA1 is well known, but we believe PA2 is new. Section 3 describes our new sequential algorithms SA1 and SA2, both of which are based on PA1. Both sequential algorithms assume A is too large to fit in fast memory, but differ based on whether they assume there is room to keep all the vectors $[x, Ax, ...A^k x]$ in fast memory (SA1) or not (SA2). SA1 is well known, but we believe SA2 is new.

In both sections 2 and 3 we first describe our algorithms for an extremely simple sparse matrix, a tridiagonal matrix. This matrix is too sparse for our methods to be of much advantage, but is good for explaining how they work. Then we describe our algorithms for a more interesting model problem, the 2D analogue of a tridiagonal matrix, namely a matrix whose sparsity pattern is that of a 2D 5 point stencil (see section 1.1 for details). Then we describe 3D meshes, summarize the operation counts in a table for more general meshes, describe now the algorithm work for general sparse matrices, and finally describe how the algorithm simplifies when the sparse matrix is really a *stencil operator* (i.e. the matrix entries are the same for each mesh point, and so do not need to be stored for each mesh point or communicated). In the sequential case SA2 we also show how the problem of achieving the absolute minimal communication complexity for a general matrix can be reduced to solving a certain instance of the Travelling Salesman Problem (TSP), and describe our cheap approximation scheme for this NP-complete problem.

Section 4 uses the performance models for 2D and 3D meshes to describe how to optimally choose k for asymptotically large problems. Section 5 presents the detailed performance models of the machines Peta, Grid, OOC and Clovertown described above. Section 6 describes the performance of our actual out-of-core implementation.

Section 7 shows how all these results may be extended to the case of preconditioned KSMs. under suitable conditions on the preconditioner.

Section 8 discusses related work. There is a long history of contributions to this area, and we detail what we believe is new about our work. In particular, we believe algorithms PA2 and SA2, our extensions to preconditioned kernels, and our implementations and performance modeling are new. We have chosen to make a comprehensive presentation including the known algorithms PA1 and SA1 to make the material more comprehensible.

1.1 Model Problems

Our techniques work for general sparse matrices that have sparsity patterns that partition "nicely" into row blocks (perhaps after reordering) but to analyze performance more precisely we will consider the following model problems. We assume a symmetric pattern (but arbitrary nonsymmetric matrix entries) and describe the pattern in terms of its undirected graph. We use the term *mesh with bandwidth b* to mean a graph whose basic connectivity is nearest neighbors on a d-dimensional mesh along with connections to mesh points that can be reached by traversing b neighest neighbors in the mesh in all possible directions. Another way to describe this graph is a d-dimensional mesh with a $(2b+1)^d$ point *stencil*. (We say a matrix has a *stencil graph* to refer only to the nonzero pattern, not the matrix entries, which are arbitrary. We will also consider how our techniques specialize when applied to *stencil matrices*, where the nonzero matrix entries are the same for each mesh point.)

1. 1D mesh with n unknowns and bandwidth $b = 1$, or a 3 point stencil graph; i.e. a tridiagonal

matrix (we consider tridiagonal matrices in order to illustrate our techniques most clearly, not because they are computationally challenging for computing products)

2. 1D mesh with n unknowns and bandwidth $b > 1$, or a $2b + 1$ point stencil graph; i.e. a band matrix with bandwidth b
3. 2D mesh with n^2 unknowns and a 5 point stencil graph
4. 2D mesh with n^2 unknowns and bandwidth 1, i.e. a 9 point stencil graph
5. 2D mesh with n^2 unknowns and bandwidth b, i.e. a $(2b + 1)^2$ point stencil graph
6. 3D mesh with n^3 unknowns and a 7 point stencil graph
7. 3D mesh with n^3 unknowns and bandwidth 1, i.e. a 27 point stencil graph
8. 3D mesh with n^3 unknowns and bandwidth b, i.e. a $(2b + 1)^3$ point stencil graph

We distribute the 1D meshes in natural order on p processors (i.e. with n/p consecutive unknowns per processor, along with the entries of the corresponding matrix rows), the 2D meshes in nested dissection ordering (i.e. with an $\frac{n}{p^{1/2}}$-by-$\frac{n}{p^{1/2}}$ square of $\frac{n^2}{p}$ unknowns per processor, along with corresponding matrix rows), and the 3D meshes in nested dissection ordering (i.e. with an $\frac{n}{p^{1/3}}$-by-$\frac{n}{p^{1/3}}$-by-$\frac{n}{p^{1/3}}$ cube of $\frac{n^3}{p}$ unknowns per processor, again with corresponding matrix rows). We assume all the above roots (like $p^{1/3}$) and fractions (like $\frac{n}{p^{1/2}}$) are integers, for simplicity.

The *surface* of a mesh is the number of points x_i in a processor's partition where $(Ax)_i$ depends on a point on another processor. For a 2D with a 5 point stencil, as just described, the surface is $4\frac{n}{p^{1/2}}$. For a 3D mesh with a 7 point stencil, the surface is $6\frac{n^2}{p^{2/3}}$. The *volume* of a mesh is the total number of points in processor's partition, namely $\frac{n^2}{p}$ and $\frac{n^3}{p}$ in the 2D and 3D cases. The *surface-to-volume ratio* of a mesh is therefore $4\frac{p^{1/2}}{n}$ in the 2D case and $6\frac{p^{1/3}}{n}$ in the 3D case. The surface-to-volume ratio of a partition of a general sparse matrix is defined analogously. All our algorithm work best when the surface-to-volume ratio is small, as is the case for meshes with large n and sufficiently smaller p. partitions described above, the surface is

2 Parallel Algorithms

We consider the conventional parallel algorithm, as well as our two new approaches:

Conventional Parallel Approach (PA0). The algorithm runs in k phases, where phase j computes $y_j = A^j x$ from $y_{j-1} = A^{j-1}x$ by each processor receiving messages with the needed remotely stored entries of y_{j-1} and computing its local components of y_j.

Parallel Approach 1 (PA1). We begin the computation of all locally computable components of $[Ax, ..., A^k x]$, and simultaneously begin sending all the components of x needed by the neighboring processors to compute the remaining components of $[Ax, ..., A^k x]$). When the locally computable components are complete, we block until the remote components of x arrive. This maximizes the potential overlap of computation and communication, but does not minimize redundant work, as we will see in PA2.

Parallel Approach 2 (PA2). We compute the set of local values of $[Ax, ..., A^k x]$ needed by the neighboring processors, so as to minimize redundant computation. Then we send these values to the neighboring processors, and simultaneously compute the remaining locally computable values. When all the locally computable values are complete, we block until the remote components of $[Ax, ..., A^k x]$ arrive, and complete the work. This minimizes redundant work, but permits slightly less overlap of computation and communication.

The difference between PA1 and PA2 will become clearer when we explain them for the 1D mesh.

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

1. number of floating point operations per processor
2. number of floating point numbers communicated per processor (the "bandwidth cost")
3. number of messages sent per processor (the "latency cost"),
4. total memory required per processor for the matrix, and
5. total memory required per processor for the vectors.

Now we argue informally why either approach PA1 or PA2 approximately minimizes communication. We assume that there is no cancellation in any of the powers A^j or $A^j x$ that would make them sparser than if all their nonzero entries were nonnegative, and that there are no algebraic relations among entries of A and/or x. Thus the complexity only depends on the sparsity pattern, and for simplicity of notation we assume all the nonzero entries of A and x are positive. In particular the set $\mathcal{D}$ of processors owning entries of x on which block row i of $[Ax, A^2x, ..., A^k x]$ depends may be described as the set of processors owning those x_j where block row i of $A + A^2 + \cdots + A^k$ has a nonzero j-th column. In both algorithms PA1 and PA2, the processor owning row block i receives exactly one message from each processor in $\mathcal{D}$, which minimizes latency. Furthermore, PA1 only sends those entries of x in each message on which the answer depends, which minimizes the bandwidth cost. PA2 sends the same amount of data although different values so as to minimize redundant computation.

The rest of this section is organized as follows. Subsection 2.1 describes PA1 and PA2 in more detail for 1D meshes (band matrices). Subsection 2.2 describes PA1 and PA2 in more detail for 2D and 3D meshes. Subsection 2.3 presents a tabular summary of all the operation counts for meshes, and discusses how they specialize to stencil matrices, where each row of the matrix has identical nonzero entries (modulo boundaries). Subsection 2.4 describes PA1 and PA2 on general sparse matrices.

2.1 1D meshes

We begin by considering a tradiagonal matrix (i.e. with bandwidth $b = 1$). In the conventional parallel algorithm each processor executes the following code in order to compute $x^{(j)} = A^j x^{(0)}$ for $j = 1$ to k:

Alg PA0: Conventional Parallel Approach for 1D mesh with $b = 1$
... assume processor q owns $x_{s_q}^{(0)}$,...,$x_{e_q}^{(0)}$

... algorithm ignores boundaries $q = 1$ and $q = p$
for $j = 1$ to k
start sending $x_{s_q}^{(j-1)}$ to processor $q-1$
start sending $x_{e_q}^{(j-1)}$ to processor $q+1$
start receiving $x_{s_q-1}^{(j-1)}$ from processor $q-1$
start receiving $x_{e_q+1}^{(j-1)}$ from processor $q+1$
compute $x_{s_q+1}^{(j)} = (Ax^{(j-1)})_{s_q+1}$, ... , $x_{e_q-1}^{(j)} = (Ax^{(j-1)})_{e_q-1}$
wait for messages to arrive
compute $x_{s_q}^{(j)} = (Ax^{(j-1)})_{s_q}$ and $x_{e_q}^{(j)} = (Ax^{(j-1)})_{e_q}$
endfor

The computational cost is clearly $2k$ messages, $2k$ words sent, and $5k\frac{n}{p}$ flops (3 multiplies and 2 additions per vector component computed). The memory required per processor is $3\frac{n}{p}$ matrix entries and $(k+1)\frac{n}{p}+2$ vector entries (for the local components of $[x, Ax, .., A^k x]$ and for the values on neighboring processors).

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

Figure 2 shows how to compute these remaining circles using PA1. The dependencies are again shown by diagonal and vertical lines below each circle, but now dependencies on data formally owned by the right processor are shown in red. All these values in turn depend on the $k = 8$ leftmost value of $x^{(0)}$ owned by the right processor, shown as black circles containing red asterisks in the bottom row. By sending these values from the right processor to the left processor, the left processor can compute all the circles whose dependencies are shown in Figure 2. The black circles indicate computations ideally done only by the left processor, and the red circles show redundant computations, i.e. ones also done by the right processor.

The following algorithm summarizes this discussion:

Alg PA1: New Parallel Approach 1 for 1D mesh with $b = 1$
... assume processor q owns $x_{s_q}^{(0)}$,...,$x_{e_q}^{(0)}$; ignore boundaries as in PA0
start sending $x_{s_q}^{(0)}$ through $x_{s_q+k-1}^{(0)}$ to processor $q-1$
start sending $x_{e_q-k+1}^{(0)}$ through $x_{e_q}^{(0)}$ to processor $q+1$
start receiving $x_{s_q-k}^{(0)}$ through $x_{s_q-1}^{(0)}$ from processor $q-1$
start receiving $x_{e_q+1}^{(0)}$ through $x_{e_q+k}^{(0)}$ from processor $q+1$
compute locally dependent components of $A^j x^{(0)}$ as shown in Figure 1
wait for messages to arrive
compute remaining red and black components of $A^j x^{(0)}$ as shown in Figure 2

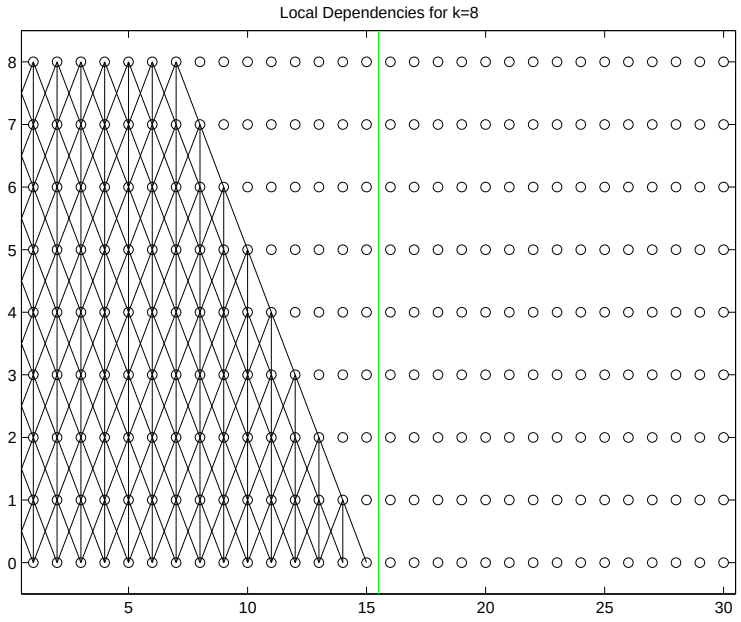


Figure 1: Locally Computable components of $[Ax, ..., A^8x]$ for tridiagonal matrix

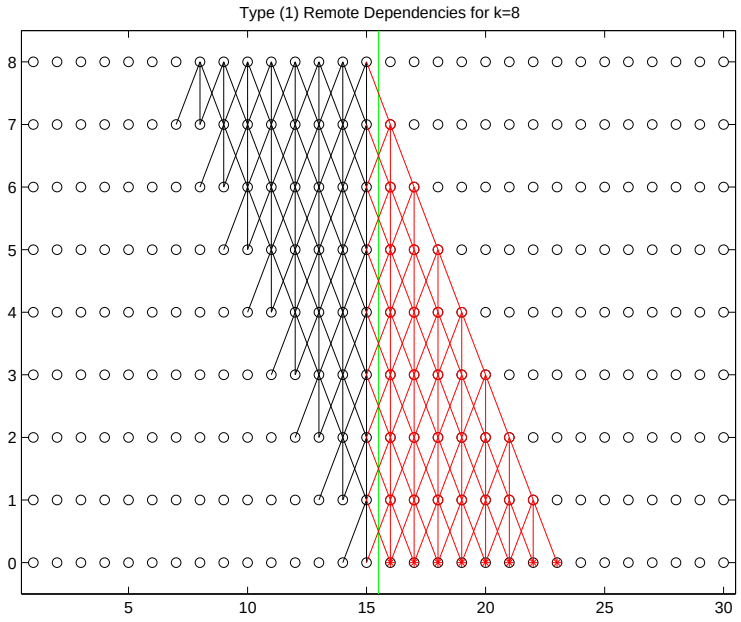


Figure 2: Remote dependencies in PA1 for $[Ax, ..., A^8x]$ for tridiagonal matrix

The memory required by PA1 is $(k+4)\frac{n}{p}$ as in PA0 plus $2k$ more words for vector entries plus $6(k-1)$ more words for matrix entries, altogether $8k-6$ more than PA0. PA1's other costs are 2 messages (versus $2k$ for the conventional method), $2k$ words sent (same as the conventional method), and $5k\frac{n}{p}+5k(k-1)$ flops, or roughly $5k^2$ more flops than the conventional method. This can also be described as an increase in flops by a factor $1+k/(\frac{n}{p})$.

Note that we are assuming that $k < \frac{n}{p}$, so that only data from neighboring processors is needed, rather than more distant processors. Indeed, we expect that $k \ll \frac{n}{p}$ in practice, which will mean that the number of extra flops (not to mention extra memory) will be negligible. We continue to make this assumption later without repeating it, and use it to simplify some expressions in the Summary Table 1 in Section 2.3.

Now consider PA2, illustrated in Figure 3. We note that the blue circles owned by the right processor and attached to blue lines can be computed locally by the right processor. The 8 circles containing red asterisks can then be sent to the left processor to compute the remaining circles connected to black and/or red lines. This saves the redundant work represented by the blue circles, but leaves the redundant work to compute the red circles, about half the redundant work of PA1.

The following algorithm summarizes this discussion:

Alg PA2: New Parallel Approach 2 for 1D mesh with $b = 1$
... assume processor q owns $x^{(0)}_{s_q}$,...,$x^{(0)}_{e_q}$; ignore boundaries as in PA0
compute blue circles corresponding to own blue triangles in Figure 3
start sending appropriate blue circles with red asterisks to processor $q-1$
start sending appropriate blue circles with red asterisks to processor $q+1$
start receiving appropriate blue circles with red asterisks from processor $q-1$
start receiving appropriate blue circles with red asterisks from processor $q+1$
compute locally dependent components of $A^j x^{(0)}$ as shown in Figure 1
wait for messages to arrive
compute remaining red and black components of $A^j x^{(0)}$ as shown in Figure 3

The memory required by PA2 is $(k+4)\frac{n}{p}$ as in PA0 plus $2k$ more words for vector entries plus $6\lfloor\frac{k}{2}\rfloor$ more words for matrix entries, altogether roughly $5k$ more words than PA0. The other costs of PA2 are 2 messages (versus $2k$ for the conventional method), $2k$ words sent (same as the conventional method), and $5k\frac{n}{p}+10\lfloor\frac{k}{2}\rfloor(\lfloor\frac{k}{2}\rfloor+odd(k))$ flops, where $odd(k)=1$ if k is odd and $odd(k)=0$ if k is even. In other words, PA2 takes roughly $\frac{5}{2}k^2$ more flops than the conventional method, half as many extra flops as PA1.

We will not always draw the corresponding detailed pictures or algorithms for the other meshes, but the same kinds of analyses apply. Nor will we compute the exact expressions for the number of extra flops, but rather approximate the number of mesh points in the black red and blue regions (pyramids, triangles, tetrahedra, and higher dimensional polyhedra) that arise by computing the leading terms of the volumes of these geometric objects. The next sections will sketch these results, and the Section 2.3 will summarize all the results in a table.

Now we briefly address 1D meshes with bandwidth $b > 1$, i.e. band matrices. The work per mesh point is $2b+1$ multiplications and $2b$ additions, or $4b+1$ flops in all per mesh point, or $(4b+1)nk/p$

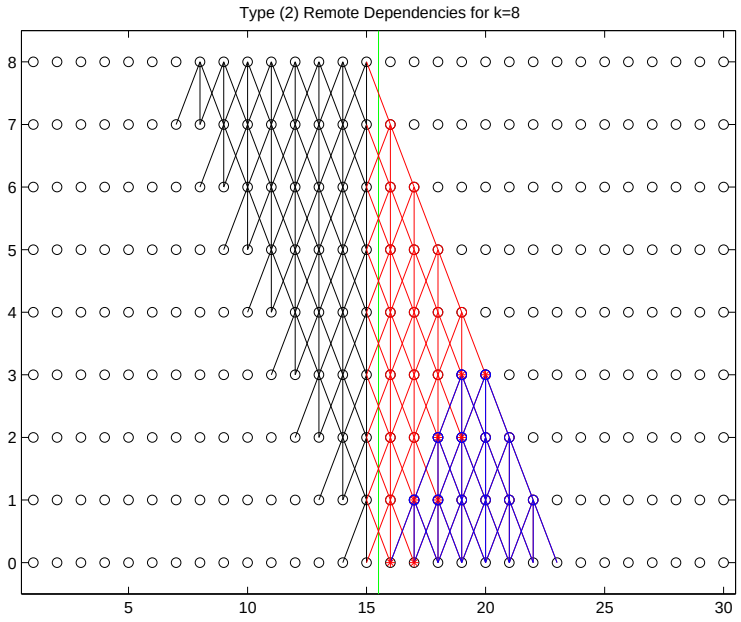


Figure 3: Remote dependencies in PA2 for $[Ax, ..., A^8x]$ for tridiagonal matrix

flops for the conventional method. A total of $2kb$ words are communicated with processors to the left and right, in $2k$ messages. The memory required per processor is $(k+1)\frac{n}{p} + 2b$ words for the vectors and $(2b+1)\frac{n}{p}$ words for the matrix entries, or $(2b+k+2)\frac{n}{p} + 2b$ words in all.

Now consider PA1. To compute the extra flops, we must count the number of mesh points in the region corresponding to the red triangle in Figure 2, namely $bk(k-1)/2$. To get the number of extra flops, this is multiplied by $4b+1$. The number of messages is again 2, containing $2kb$ words in all. The number of words of memory required is $(k+1)\frac{n}{p} + 2kb$ for the vectors and $(2b+1)(\frac{n}{p} + 2kb)$ for the matrix entries, or $(2b+k+2)\frac{n}{p} + kb(4b+4)$ words in all.

Now consider PA2. The region corresponding to the blue region in Figure 3 has again about half the number of mesh points as the region corresponding to the red region in Figure 2, roughly $bk(k-1)/4$. To get the number of extra flops, this is again multipled by $4b+1$, and by 2, for the right and left boundaries. The number of words of memory required for vectors is the same as PA1, and slightly smaller for matrix entries, $(2b+1)(\frac{n}{p} + kb)$.

2.2 2D and 3D meshes

2.2.1 2D mesh with a 5 point stencil graph

We consider multiplying by a matrix whose graph is the 5-point stencil, i.e. with North, South, East, West (NSEW) connections on an n-by-n grid of unknowns partitioned on $p^{1/2}$-by-$p^{1/2}$ grid of processors. We assume $p^{1/2}|n$, so that each processor owns a $\frac{n}{p^{1/2}}$-by-$\frac{n}{p^{1/2}}$ square of grid points (vector components), and their corresponding matrix rows, and that $k < \frac{n}{p^{1/2}}$. We expect that $k \ll \frac{n}{p^{1/2}}$ in practice, which will justify omitting certain lower order terms in k in Table 1.

Figure 4 shows the remote domain of dependence for a single processor (demarcated by green lines as before) owning a 10-by-10 grid of unknowns (the black circles). When $k = 3$, the results of $[Ax, .., A^k x]$ will depend on the remote values shown by black circles containing red asterisks (the same notation as Figure 2). Unlike Figure 2, Figure 4 does not show circles for components of $A^j x$ for $j > 0$, but rather a projected view. Figure 5 shows a 3D view analogous to Figure 2.

The number of messages decreases from $4k$ for the conventional algorithm to 8 instead of to 4 for PA1, because communication is required with the corner neighbors (NW, SW, NE and SE), as well as side neighbors (N, S, E and W). The volume of communication also grows slightly to include the triangles of size $k-1$ owned by the 4 corner neighbors. The number of flops grows roughly by the factor $1 + 2k/(\frac{n}{p^{1/2}})$. When $k \ll \frac{n}{p^{1/2}}$, this increase is quite small. It is a little harder to visualize the regions of redundant computations than in the 1D mesh case: In the side neighbors, the red circles denoting redundant computations form a prism with triangular cross section and volume (and number of contained points) proportional to $k\frac{n}{p^{1/2}}$, and in the corner neighbors the red circles form a pyramid with volume (and number of contained points) proportional to k^3. The memory requirement for the conventional algorithm is $(k+1)\frac{n^2}{p} + 4\frac{n}{p^{1/2}}$ vector entries and $5\frac{n^2}{p}$ matrix entries; for PA1 it increases by an additional $4k\frac{n}{p^{1/2}} + 2k^2$ vector entries and 5 times as many matrix entries. The Summary Table has details.

Figure 6 shows the dependencies for PA2 applies to the 2D mesh with a 5 point stencil graph. As in Figure 3, there are black circles that are the desired (or initial) values, red circles representing redundant work, blue circles denoting work that was redundant in PA1 but saved by PA2, and circles containing red asterisks that are to be communicated. In the side processors, the regions of redundant computations again form prisms with triangular cross sections, with half the cross

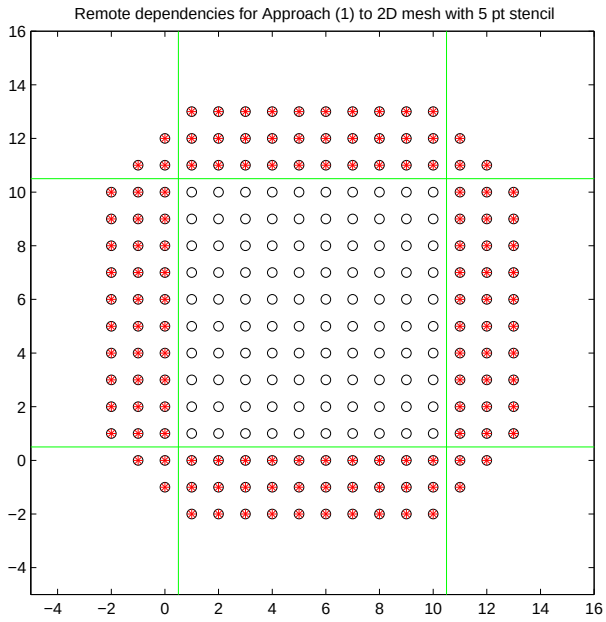


Figure 4: Remote dependencies in PA1 for $[Ax, ..., A^3x]$ for 2D mesh with 5 point stencil, projected view

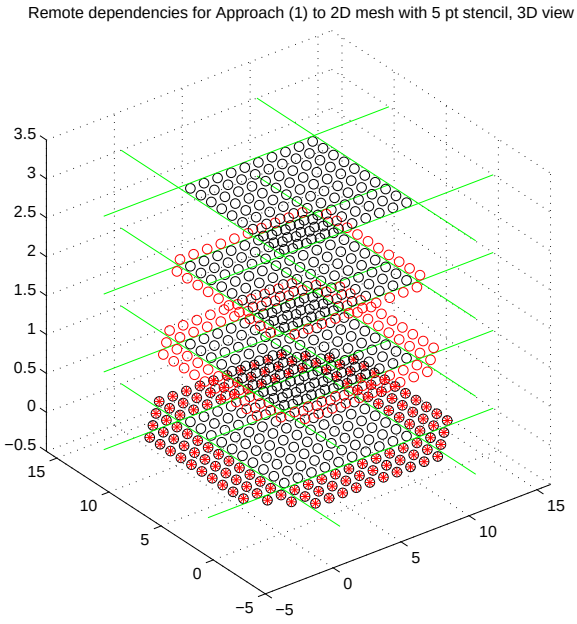


Figure 5: Remote dependencies in PA1 for $[Ax, ..., A^3x]$ for 2D mesh with 5 point stencil, 3D view

sectional area of PA1. Thus, like the 1D case, this means about half the redundant work is saved. It is somewhat more difficult to see what is happening in the corner processors. The pyramid of redundant operations from PA1 has a smaller tetrahedron of locally computable components subtracted from it; geometrical symmetry considerations indicate that this saves about 1/3 of the redundant operations in the corners. The number of extra words of memory required for matrix entries decreases by nearly a factor of 2.

2.2.2 2D mesh with 9 point stencil graph

We consider multiplying by a matrix whose graph is the 9-point stencil, i.e. with N, S, E, W, NE, SE, SW, and NW connections on an n-by-n grid of unknowns partitioned on $p^{1/2}$-by-$p^{1/2}$ grid of processors. We assume $p^{1/2}|n$.

Figures 7, 8, and 9 are analogous to Figures 4, 5, and 6, respectively. A similar counting exercise leads to the entries in the Summary Table.

2.2.3 2D mesh with $(2b+1)^2$ point stencil graph

We consider multiplying by a matrix A whose graph is a $(2b+1)^2$ point stencil, i.e. where each vertex has connections to other vertices within b to the left, right, up or down. The 9 point stencil graph of the last section is the special case $b = 1$. This can be thought of as a generalization of a band matrix to 2D; when exhibited in natural order the matrix has $2b+1$ bands, each of which is $2b+1$ entries wide.

The conventional algorithm for multiplying Ax requires 8 messages. The 4 to the side processors each receive $b\frac{n}{p^{1/2}}$ vector entries, and the 4 to the corner processors each receive b^2 vector entries. The number of flops is $2(2b+1)^2 - 1$ per vector entry for a total of $(8b^2+8b+1)\frac{n^2}{p}$.

Similar counting exercises lead to the other entries in the Summary Table.

2.2.4 3D meshes, with 7 point, 27 point and $(2b+1)^3$ point stencils graphs

We first consider multiplying by a matrix whose graph is the 7-point stencil on an n-by-n-by-n grid of unknowns partitioned on a $p^{1/3}$-by-$p^{1/3}$-by-$p^{1/3}$ grid of processors. We assume $p^{1/3}|n$.

The Summary Table entries are estimated as follows. There are 26 neighbors of a processor, so 26 messages need to be exchanged. The conventional algorithm will exchange $\frac{n^2}{p^{2/3}}$ boundary values with each of its 6 "face neighbors" at each step, for a total of $6k\frac{n^2}{p^{2/3}}$ words sent. The conventional algorithm will also do 13 flops to compute each of the $\frac{n^3}{p}$ components it owns at each step, for a total of $13k\frac{n^2}{p}$ flops.

The other entries of the Summary Table are based on surface-to-volume analogies to the earlier cases.

2.3 Summary of Parallel Complexity of Computing $[Ax, ..., A^k x]$ on Meshes

In the Summary table for Parallel Algorithms, "Mess" is the number of messages sent per processor, "Words" is the total size of these messages, "Flops" is the number of floating point operations, "MMem" is the amount of memory needed per processor for the matrix entries, and "VMem" is the amount of memory needed per processor for the vector entries.

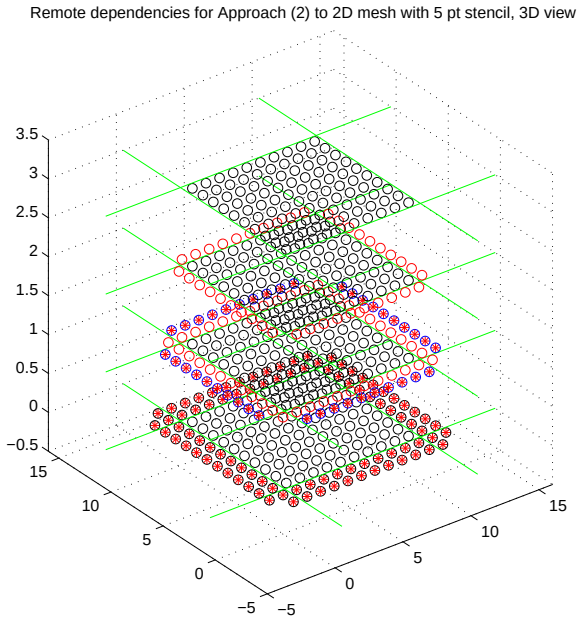


Figure 6: Remote dependencies in PA2 for $[Ax, ..., A^3x]$ for 2D mesh with 5 point stencil, 3D view

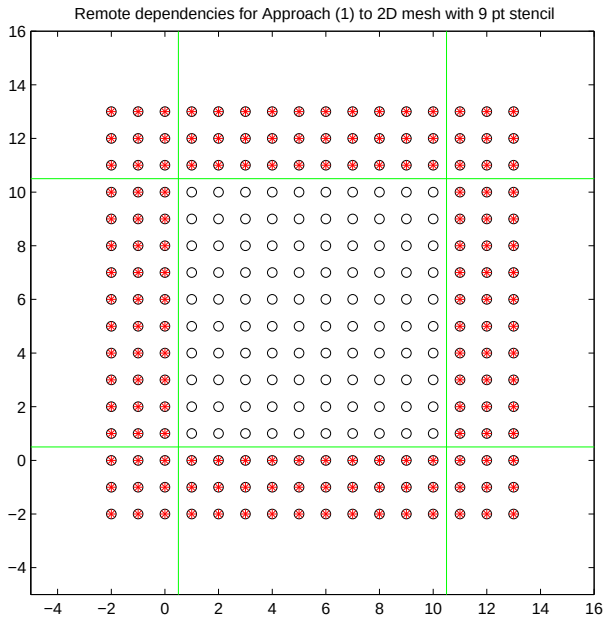


Figure 7: Remote dependencies in PA1 for $[Ax, ..., A^3x]$ for 2D mesh with 9 point stencil, projected view

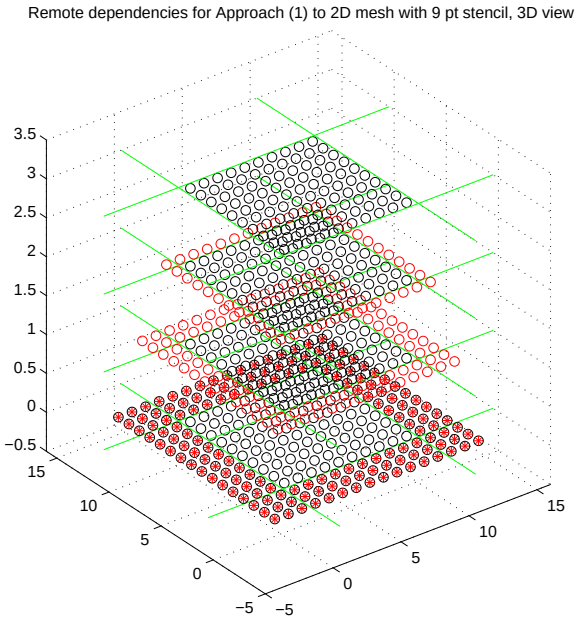


Figure 8: Remote dependencies in PA1 for $[Ax, ..., A^3x]$ for 2D mesh with 9 point stencil, 3D view

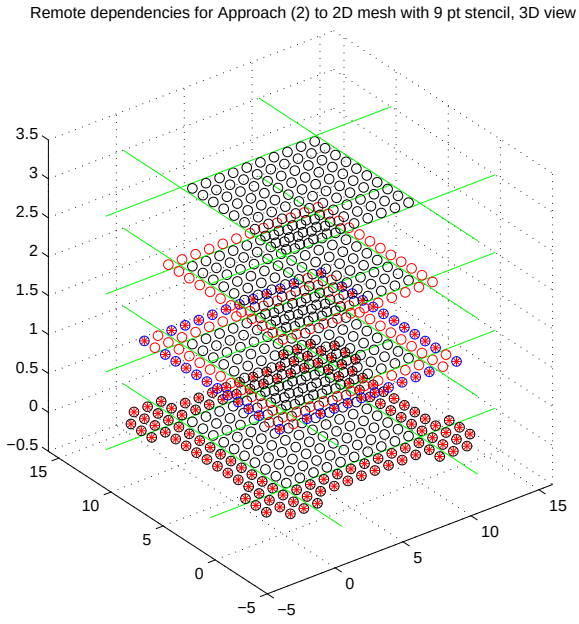


Figure 9: Remote dependencies in PA2 for $[Ax, ..., A^3x]$ for 2D mesh with 9 point stencil, 3D view

Lower order terms are sometimes omitted for clarity.

We consider the special case of stencil matrices, where in addition to the graph being a stencil, each row of the matrix has the same nonzero entries (modulo boundaries). In this case at most MMem $\leq (2b+1)^d$ words are needed to store the matrix entries for a d-dimensional mesh, not $(2b+1)^d \frac{n^d}{p}$, but otherwise the table entries do not change.

Table 1: Summary Table for Parallel Algorithms (some lower order terms omitted)				
Problem	Costs	Conventional Approach	Parallel Approach 1	Parallel Approach 2
1D mesh $b=1$	Mess	$2k$	2	2
	Words	$2k$	$2k$	$2k$
	Flops	$5k\frac{n}{p}$	$5(k\frac{n}{p}+k^2)$	$5(k\frac{n}{p}+\frac{k^2}{2})$
	MMem	$3\frac{n}{p}$	$3\frac{n}{p}+6k$	$3\frac{n}{p}+3k$
	VMem	$(k+1)\frac{n}{p}$	$(k+1)\frac{n}{p}+2k$	$(k+1)\frac{n}{p}+2k$
1D mesh $b\geq 1$	Mess	$2k$	2	2
	Words	$2bk$	$2bk$	$2bk$
	Flops	$(4b+1)k\frac{n}{p}$	$(4b+1)(k\frac{n}{p}+bk^2)$	$(4b+1)(k\frac{n}{p}+\frac{bk^2}{2})$
	MMem	$(2b+1)\frac{n}{p}$	$(2b+1)\frac{n}{p}+bk(4b+2)$	$(2b+1)\frac{n}{p}+bk(2b+1)$
	VMem	$(k+1)\frac{n}{p}+2b$	$(k+1)\frac{n}{p}+2bk$	$(k+1)\frac{n}{p}+2bk$
2D mesh 5 pt	Mess	$4k$	8	8
	Words	$4k\frac{n}{p^{1/2}}$	$4k\frac{n}{p^{1/2}}+2k^2$	$4k\frac{n}{p^{1/2}}+2k^2$
	Flops	$9k\frac{n^2}{p}$	$9k\frac{n^2}{p}+18k^2\frac{n}{p^{1/2}}+6k^3$	$9k\frac{n^2}{p}+9k^2\frac{n}{p^{1/2}}+4k^3$
	MMem	$5\frac{n^2}{p}$	$5\frac{n^2}{p}+20\frac{kn}{p^{1/2}}+10k^2$	$5\frac{n^2}{p}+10\frac{kn}{p^{1/2}}+5k^2$
	VMem	$(k+1)\frac{n^2}{p}+4\frac{n}{p^{1/2}}$	$(k+1)\frac{n^2}{p}+4\frac{kn}{p^{1/2}}+2k^2$	$(k+1)\frac{n^2}{p}+4\frac{kn}{p^{1/2}}+2k^2$
2D mesh 9 pt	Mess	$8k$	8	8
	Words	$4k(\frac{n}{p^{1/2}}+1)$	$4k(\frac{n}{p^{1/2}}+k)$	$4k(\frac{n}{p^{1/2}}+1.5k)$
	Flops	$17k\frac{n^2}{p}$	$17k\frac{n^2}{p}+34k^2\frac{n}{p^{1/2}}+\frac{68}{3}k^3$	$17k\frac{n^2}{p}+17k^2\frac{n}{p^{1/2}}+17k^3$
	MMem	$9\frac{n^2}{p}$	$9\frac{n^2}{p}+36\frac{kn}{p^{1/2}}+36k^2$	$9\frac{n^2}{p}+18\frac{kn}{p^{1/2}}+9k^2$
	VMem	$(k+1)\frac{n^2}{p}+4\frac{n}{p^{1/2}}$	$(k+1)\frac{n^2}{p}+4\frac{kn}{p^{1/2}}+4k^2$	$(k+1)\frac{n^2}{p}+4\frac{kn}{p^{1/2}}+6k^2$
2D mesh $(2b+1)^2$ pt stencil	Mess	$8k$	8	8
	Words	$4bk(\frac{n}{p^{1/2}}+b)$	$4bk(\frac{n}{p^{1/2}}+bk)$	$4bk(\frac{n}{p^{1/2}}+1.5bk)$
	Flops	$(8b^2+8b+1)k\frac{n^2}{p}$	$(8b^2+8b+1)\cdot$ $(k\frac{n^2}{p}+2bk^2\frac{n}{p^{1/2}}+\frac{4}{3}b^2k^3)$	$(8b^2+8b+1)\cdot$ $(k\frac{n^2}{p}+bk^2\frac{n}{p^{1/2}}+b^2k^3)$
	MMem	$(2b+1)^2)\frac{n^2}{p}$	$(2b+1)^2(\frac{n^2}{p}+4bk\frac{n}{p^{1/2}}+4b^2k^2)$	$(2b+1)^2(\frac{n^2}{p}+2bk\frac{n}{p^{1/2}}+b^2k^2)$
	VMem	$(k+1)\frac{n^2}{p}+4b\frac{n}{p^{1/2}}+4b^2$	$(k+1)\frac{n^2}{p}+4bk\frac{n}{p^{1/2}}+4b^2k^2$	$(k+1)\frac{n^2}{p}+4bk\frac{n}{p^{1/2}}+6b^2k^2$
3D mesh 7 pt stencil	Mess	$6k$	26	26
	Words	$6k\frac{n^2}{p^{2/3}}$	$6k\frac{n^2}{p^{2/3}}+6k^2\frac{n}{p^{1/3}}+O(k^3)$	$6k\frac{n^2}{p^{2/3}}+6k^2\frac{n}{p^{1/3}}+O(k^3)$
	Flops	$13k\frac{n^3}{p}$	$13k\frac{n^3}{p}+39k^2\frac{n^2}{p^{2/3}}+O(k^3\frac{n}{p^{1/3}})$	$13k\frac{n^3}{p}+\frac{39}{2}k^2\frac{n^2}{p^{2/3}}+O(k^3\frac{n}{p^{1/3}})$
	MMem	$7\frac{n^3}{p}$	$7\frac{n^3}{p}+42\frac{n^2}{p^{2/3}}+O(k^2\frac{n}{p^{1/3}})$	$7\frac{n^3}{p}+21k\frac{n^2}{p^{2/3}}+O(k^2\frac{n}{p^{1/3}})$
	VMem	$(k+1)\frac{n^3}{p}+6\frac{n^2}{p^{2/3}}$	$(k+1)\frac{n^3}{p}+6k\frac{n^2}{p^{2/3}}+O(k^2\frac{n}{p^{1/3}})$	$(k+1)\frac{n^3}{p}+6k\frac{n^2}{p^{2/3}}+O(k^2\frac{n}{p^{1/3}})$
3D mesh 27 pt stencil	Mess	$26k$	26	26
	Words	$6k\frac{n^2}{p^{2/3}}+12k\frac{n}{p^{1/3}}+O(k)$	$6k\frac{n^2}{p^{2/3}}+12k^2\frac{n}{p^{1/3}}+O(k^3)$	$6k\frac{n^2}{p^{2/3}}+12k^2\frac{n}{p^{1/3}}+O(k^3)$
	Flops	$53k\frac{n^3}{p}$	$53k\frac{n^3}{p}+159k^2\frac{n^2}{p^{2/3}}+O(k^3\frac{n}{p^{1/3}})$	$53k\frac{n^3}{p}+\frac{159}{2}k^2\frac{n^2}{p^{2/3}}+O(k^3\frac{n}{p^{1/3}})$
	MMem	$27\frac{n^3}{p}$	$27\frac{n^3}{p}+162k\frac{n^2}{p^{2/3}}+O(k^2\frac{n}{p^{1/3}})$	$27\frac{n^3}{p}+81k\frac{n^2}{p^{2/3}}+O(k^2\frac{n}{p^{1/3}})$
	VMem	$(k+1)\frac{n^3}{p}+6\frac{n^2}{p^{2/3}}+O(\frac{n}{p^{1/3}})$	$(k+1)\frac{n^3}{p}+6k\frac{n^2}{p^{2/3}}+O(k^2\frac{n}{p^{1/3}})$	$(k+1)\frac{n^3}{p}+6k\frac{n^2}{p^{2/3}}+O(k^2\frac{n}{p^{1/3}})$
3D mesh $(2b+1)^3$ pt stencil	Mess	$26k$	26	26
	Words	$6bk\frac{n^2}{p^{2/3}}+12b^2k\frac{n}{p^{1/3}}+O(b^3k)$	$6bk\frac{n^2}{p^{2/3}}+12b^2k^2\frac{n}{p^{1/3}}+O(b^3k^3)$	$6bk\frac{n^2}{p^{2/3}}+12b^2k^2\frac{n}{p^{1/3}}+O(b^3k^3)$
	Flops	$(2(2b+1)^3-1)k\frac{n^3}{p}$	$(2(2b+1)^3-1)\cdot$ $(k\frac{n^3}{p}+3bk^2\frac{n^2}{p^{2/3}}+O(b^2k^3\frac{n}{p^{1/3}}))$	$(2(2b+1)^3-1)\cdot$ $(k\frac{n^3}{p}+\frac{3}{2}bk^2\frac{n^2}{p^{2/3}}+O(b^2k^3\frac{n}{p^{1/3}}))$
	MMem	$(2b+1)^3\frac{n^3}{p}$	$(2b+1)^3(\frac{n^3}{p}+6bk\frac{n^2}{p^{2/3}}+O(b^2k^2\frac{n}{p^{1/3}}))$	$(2b+1)^3(\frac{n^3}{p}+3bk\frac{n^2}{p^{2/3}}+O(b^2k^2\frac{n}{p^{1/3}}))$
	VMem	$(k+1)\frac{n^3}{p}+6b\frac{n^2}{p^{2/3}}+O(b^2\frac{n}{p^{1/3}})$	$(k+1)\frac{n^3}{p}+6bk\frac{n^2}{p^{2/3}}+O(b^2k^2\frac{n}{p^{1/3}})$	$(k+1)\frac{n^3}{p}+6bk\frac{n^2}{p^{2/3}}+O(b^2k^2\frac{n}{p^{1/3}})$

2.4 General Graphs

Here we show how to extend the approaches PA1 and PA2 to general sparse matrices. To do so we need some graph theoretic notation. It is natural to associate a directed graph with a square sparse matrix A, with one vertex for every row/column, and an edge from vertex i to vertex j if $A_{ij} \neq 0$, meaning that component i of $y = Ax$ depends on component j of x. We will build an analogous graph, essentially consisting of k copies of this basic graph: Let $x_j^{(i)}$ be the j-th component of

$x^{(i)} = A^i \cdot x^{(0)}$. We associate a vertex with each $x_j^{(i)}$ for $i = 0, ..., k$ and $j = 1, ..., n$ (and use the same notation to name the vertex), and an edge from $x_j^{(i+1)}$ to $x_m^{(i)}$ when $A_{jm} \neq 0$, and call this graph of $n(k+1)$ vertices G. (We will not need to construct all of G in practice, but using G makes it easy to describe our algorithms, in a fashion analogous to Figures 1 through 3.) We say that i is the *level* of vertex $x_j^{(i)}$. Each vertex will also have an *affinity* q, corresponding to the processor number where it is stored; we assume all vertices $x_j^{(0)}, x_j^{(1)}, ..., x_j^{(k)}$ have the same affinity, depending only on j.

We write G_q to mean the subset of vertices of G with affinity q, $G^{(i)}$ to mean the subest of vertices of G with level i, and $G_q^{(i)}$ to mean the subset with affinity q and level i.

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

Next we need to identify the locally computable components, that processor q can compute given only the values in $G_q^{(0)}$. We denote the set of locally computable components by $L_q \equiv \{x \in G_q : R(x) \subset G_q\}$. As before $L_q^{(i)}$ will denote the vertices in L_q at level i.

Finally, for PA2 we need to identify the minimal subset $B_{q,r}$ of vertices (i.e. their values) that processor r needs to send processor q so that processor q can finish computing all its vertices G_q (eg the 8 circles containing red asterisks in Figure 3): We say that $x \in B_{q,r}$ if and only if $x \in L_r$, and there is a path from some $y \in G_q$ to x such that x is the first vertex of the path in L_r.

Given all this notation, we can finally state versions of PA0, PA1 and PA2 for general graphs and partitions among processors:

PA0 - Conventional parallel algorithm for a general graph (code for processor q)
for $i = 1$ to k
for all other processors $r \neq q$, send (values of) all $x_j^{(i-1)}$ in $R_q^{(i-1)}(G_r^{(i)})$ to processor r
for all other processors $r \neq q$, receive all $x_j^{(i-1)}$ in $R_r^{(i-1)}(G_q^{(i)})$ from processor r
compute all $x_j^{(i)}$ in $L_q^{(i)}$
wait for receives to finish
compute remaining $x_j^{(i)}$ in $G_q^{(i)} - L_q^{(i)}$
end for

PA1 for a general graph (code for processor q)
for all other processors $r \neq q$, send all $x_j^{(0)}$ in $R_q^{(0)}(G_r)$ to processor r
for all other processors $r \neq q$, receive all $x_j^{(0)}$ in $R_r^{(0)}(G_q)$ from processor r
compute all $x_j^{(i)}$ in L_q (in order of increasing i)
... ex: circled vertices in Figure 1
wait for receives to finish
compute remaining $x_j^{(i)}$ in $R(G_q) - L_q$ (in order of increasing i)
... ex: circled vertices in Figure 2

We illustrate the algorithm PA1 on the matrix A whose graph is in Figure 10(a). The vertices represent rows and columns of A and the edges represent nonzeros; for simplicity we use a symmetric

matrix so the edges can be undirected. The dotted orange lines separate vertices owned by different processors, and we will let q denote the processor owning the 9 gray vertices in the center of the figure. In other words the gray vertices are $G_q^{(0)}$. For all the neighboring processors $r \neq q$, the red vertices are $R_r^{(0)}(G_q)$ for $k = 1$, the red and green vertices together are $R_r^{(0)}(G_q)$ for $k = 2$, and the red, green and blue vertices together are $R_r^{(0)}(G_q)$ for $k = 3$.

PA2 for a general graph (code for processor q)
- Phase I: compute $x_j^{(i)}$ in $\cup_{r \neq q}(R(G_r) \cap L_q)$
- ... ex: blue circled vertices in Figure 3
- for all other processors $r \neq q$ send $x_j^{(i)}$ in $B_{r,q}$ to processor r
- ... ex: blue circled vertices containing red asterisks in Figure 3
- for all other vertices $r \neq q$ receive $x_j^{(i)}$ in $B_{q,r}$ from processor r
- ... ex: blue circled vertices containing red asterisks in Figure 3
- Phase 2: compute $x_j^{(i)}$ in $L_q - \cup_{r \neq q}(R(G_r) \cap L_q)$
- ... ex: locally computable vertices of right processor minus blue circled vertices in Figure 3
- wait for receives to finish
- Phase 3: compute remaining $x_j^{(i)}$ in $R(G_q) - L_q - \cup_{r \neq q}(R(G_q) \cap L_r)$
- ... ex: black circled vertices in Figure 3 that are connected by lines

The Phases in PA2 will be referred to in section 5.

We illustrate algorithm PA2 on the same matrix (as the one used for PA1), shown in Figures 10(b) and (c), just for the case $k = 3$. In Figure 10(b), the red vertices are $B_{q,r}^{(0)}$ (the members of $B_{q,r}$ at level 0), and in Figure 10(c) the green vertices are $B_{q,r}^{(1)}$.

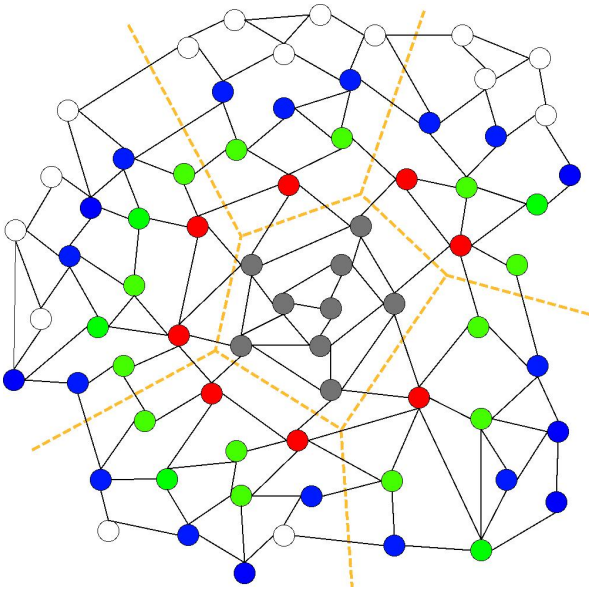
2.5 Stencils

Now we discuss how computing $[x, Ax, ..., A^k x]$ simplifies when A is truly a stencil operator, i.e. not only is the sparsity pattern described by a mesh as before, but the values of the matrix are the same in every row (modulo mesh boundaries). This means that we do not need to store any matrix entries. Otherwise, the PA1 and PA2 algorithms described so far are identical, with identical communication and computation costs. (If some nonzeros in matrix row are identical, it is also be possible to reduce the number of multiplications.)

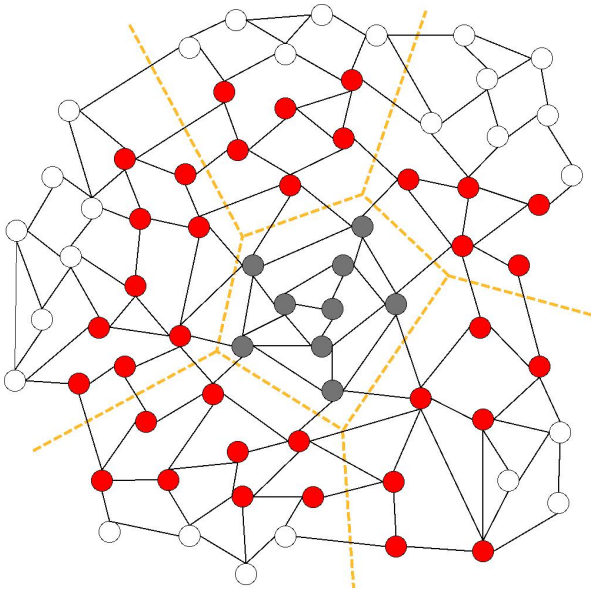
If in addition only the final vector $A^k x$ is needed but not $Ax, ..., A^{k-1}x$, then further memory savings are possible, but not computation or communication savings. This can complicate indexing; see section 8 for related work.

3 Sequential Algorithms

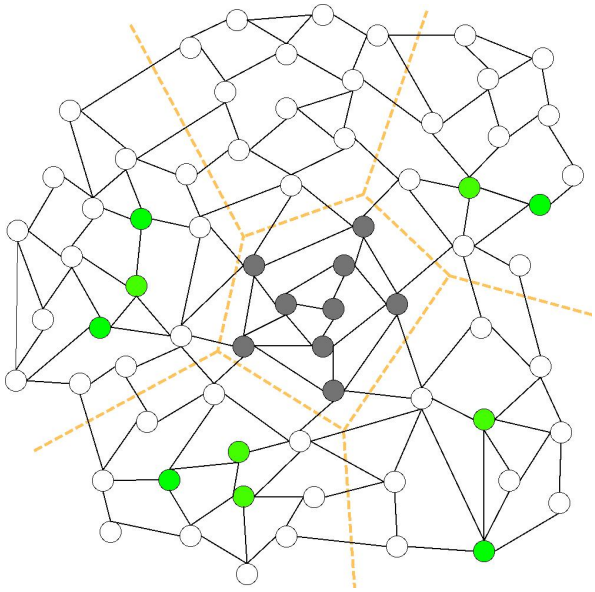
Now consider the sequential algorithm with fast and slow memories. Let us motivate the two situations we are trying to optimize. It is typical for a large N-by-N sparse matrix to take many times as much memory as vector of length N. For example, a typical density of .1% means that an N-by-N matrix takes roughly $.001N^2$ 8-byte floats and 4-byte indices to store (assuming CSR format), so roughly $.012N^2$ bytes, whereas a vector takes $8N$ bytes, which is smaller for $N > 666$, and typically many times smaller. So the two cases of most interest are (1) the matrix does not fit



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



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



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

Figure 10: Example for PA1 and PA2. The dotted lines define the different blocks. Each block resides on a different processor. The example shows from the perspective of the processor holding the central block.

in fast memory but the $k+1$ vectors $[x, Ax, ..., A^k x]$ do, and (2) neither the matrix nor the vectors fit in fast memory.

We note that for our model problems, the relative sizes of the vectors and matrix depends on k and the bandwidth b, and our algorithms are of most interest when the matrix is larger than the vectors. While this is not the case for tridiagonal matrices and $k > 2$, we will still use this simple case to illustrate how the algorithms work.

Conventional Sequential Approach (SA0). We assume the matrix does not fit in fast memory but the vectors do. This algorithm will keep all the components of $[x, Ax, ..., A^k x]$ in fast memory, and read all the entries of A from slow to fast memory to compute each vector $A^j x$, thereby reading A k times in all.

New Sequential Approach 1 (SA1). We again assume that the matrix does not fit in fast memory but the vectors do. SA1 will emulate PA1 by partitioning the matrix into p block rows, and looping from $i = 1$ to $i = p$, reading from slow memory those parts of the matrix needed to perform the same computations performed by processor i in PA1, and updating the appropriate components of $[Ax, ..., A^k x]$ in fast memory. Since all components of $[Ax, ..., A^k x]$ are in fast memory, no redundant computation is necessary. We choose p as small as possible, to minimize the number of slow memory accesses, as described below.

New Sequential Approach 2 (SA2). Now we assume that neither the matrix nor the vectors fit in memory. SA2 will still emulate PA1 by looping from $i = 1$ to $i = p$, but read from slow memory not just parts of the matrix but also those parts x needed to perform the same computations performed by processor i in PA1, and finally writing back to slow memory the corresponding components of $[Ax, ..., A^k x]$. Depending on the structure of A, redundant computation may or may not be necessary. We again choose p as small as possible.

For SA1, the total number of slow memory accesses is roughly the minimum number needed to read in the whole matrix once from slow memory, while also keeping $[x, Ax, ..., A^k x]$ in fast memory. For a sparse matrix with nnz nonzeros and a fast memory of size M bytes, the number of slow memory accesses is therefore roughly $(12nnz)/(M - 8(k+1)n)$. As long as $M \gg n$ and $nnz \gg n$, this number grows very slowly with k, justifying our claims of the latency cost being independent of k.

For SA2, we also need to read and write $k+1$ vectors to and from slow memory, so the number of slow memory accesses is roughly $(8(k+1)n + 12nnz)/M$. As long as $8(k+1)n \lessapprox 12nnz$, or

$$k \lessapprox \frac{3}{2} \cdot \frac{nnz}{n} = \frac{3}{2} \cdot \text{ the average number of nonzeros per row,}$$

the number of slow memory accesses will again be roughly independent of k as claimed.

Our sequential approach is broadly similar to that of Strout [28] and Vuduc [31] but differs in that we assume a cost of "latency + n/bandwidth" to read or write any contiguous set of n bytes from slow memory, where latency may be dominant. Therefore it is critical for us to organize our data structures so that data to be read from slow memory is entirely contiguous. We will see that this leads to new data layouts where, for example, we interleave matrix and vector entries. (In subsection 3.5 we show that finding the optimal way to reorganize the data may be formulated via the Travelling Salesman Problem (TSP), but we only need to solve it approximately to get a

reasonable solution.) This reorganization is not necessary in the parallel case, because the sending and receiving processors can pack and unpack data structures into contiguous memory segments, something a disk or memory prefetch unit cannot do (yet). This packing/unpacking has the effect of decreasing the effective bandwidth, but not the number of messages.

The rest of this section is organized as follows. Subsection 3.1 describes SA1 and SA2 in more detail for 1D meshes (band matrices). Subsection 3.2 describes SA1 and SA2 in more details for 2D and 3D meshes. Subsection 3.3 presents a tabular summary of all the operation counts for meshes. Subsection 3.4 describes SA1 and SA2 on general sparse matrices. Subsection 3.5 describes how to find the optimal ordering of unknowns, as well as good approximations to this ordering.

3.1 1D Meshes

We will explain both SA1 and SA2 for tridiagonal matrices, even though (as stated above) only SA2 makes sense in practice for such matrices, since SA1 uses too much fast memory. We also present the conventional algorithm, for contrast:

Alg SA0: Conventional Sequential Approach with Fast/Slow Memory for 1D mesh with $b = 1$
... assume matrix stored in slow memory in p equal sized chunks
... where chunk q consists of rows s_q through e_q
... assume $k + 1$ vectors $[x, Ax, ..., A^k x]$ all fit in fast memory
... let initial x be denoted $x^{(0)}$
for $j = 1$ to k
    for $q = 1$ to p
        read rows s_q through e_q of matrix from slow memory
        ... this assumes that the matrix is stored by rows, so that
        ... rows s_q through e_q are located contiguously
        compute $x_{s_q}^{(j)} = (Ax^{(j-1)})_{s_q}$, ... , $x_{e_q}^{(j)} = (Ax^{(j-1)})_{e_q}$
    endfor
endfor

The cost of this algorithm is $5kn$ flops, $3kn$ words read from slow memory, and kp accesses to slow memory. The memory required is $(k + 1)n + 3\frac{n}{p}$. Since the tridiagonal matrix has so few nonzeros, this conventional approach has no memory advantages over computing (and using and then overwriting!) the powers $A^j x$ one at a time.

As stated in the introduction, the presence of all components of $[Ax, ..., A^k x]$ in the same memory makes it unnecessary to perform any redundant computation in the new approach:

Alg SA1: New Sequential Approach 1 with Fast/Slow Memory for 1D mesh with $b = 1$
... assume matrix stored in slow memory in p equal sized chunks
... where chunk q consists of rows s_q through e_q;
... assume $k + 1$ vectors $[x, Ax, ..., A^k x]$ all fit in fast memory;
... ignore boundaries $q = 1$ and $q = p$;
for $q = 1$ to p
    read rows s_q through e_q of A from slow memory
    ... note: for $q > 1$, rows $s_q - k$ through $s_q - 1$ were already read last time,
    ... and must be kept in memory;

... this also assumes that the matrix is stored by rows, so that
... rows s_q through e_q are located contiguously
compute locally dependent components of $A^j x$ as shown in Figure 11
endfor

The cost of this algorithm is $5kn$ flops (no more than the conventional algorithm), $3n$ words read from slow memory (the matrix is read just once), and p accesses to slow memory (in contrast to kp for the conventional algorithm). The memory required is $(k+1)n + 3\frac{n}{p}$ as for SA0, plus $3k$ more for keeping rows $s_q - k$ through $s_q - 1$ of A in fast memory.

By unrolling the loop in SA1 (but at the cost of more fast memory), the latency of the read from slow memory could be overlapped with computation.

Alg SA2: New Sequential Approach 2 with Fast/Slow Memory for 1D mesh with $b = 1$
... assume matrix and vectors $[x, Ax, ..., A^k x]$ stored in slow memory in p equal sized chunks
... where chunk q consists of rows s_q through e_q;
... ignore boundaries $q = 1$ and $q = p$;
for $q = 1$ to p
read rows s_q through e_q of A and of $[x, Ax, ..., A^k x]$ from slow memory
... note: for $q > 1$, rows $s_q - k$ through $s_q - 1$ of A and of $[x, Ax, ..., A^k x]$
... were already read last time, and must be kept in memory
compute locally dependent components of $A^j x$ as shown in Figure 11
write rows $s_q - k$ through $e_q - k = s_{q+1} - k - 1$ of $[Ax, ..., A^k x]$ back to slow memory
endfor

In order to perform the read in SA2 in exactly one slow memory access, we would need to interleave the data structures of A and of $[x, Ax, ..., A^k x]$ so that the first n/p consecutive rows of A and of $[x, Ax, ..., A^k x]$ were stored contiguously together, then the second n/p rows of both, and so on. In order to also perform the write in SA2 in exactly one slow memory access, the n/p rows of x would have to come at the end of the corresponding rows of A and of $[Ax, ..., A^k x]$. We can simplify these data structures at the cost of increasing the number of slow memory accesses from 2 to at most 5, by storing A, x and $[Ax, ..., A^k x]$ separately (but still by rows).

The cost of this algorithm is $5kn$ flops (no more than the conventional algorithm), $3n + (k+1)n$ words read from slow memory (the matrix and all the vectors are read just once), and p accesses to slow memory, assuming the best possible interleaved layout of A and $[x, Ax, ..., A^k x]$ just described (in contrast to kp for the conventional algorithm).

The memory required is $3\frac{n}{p} + 3k$ for the matrix and $(k+1)\frac{n}{p} + k(k+1)$ for the vectors, roughly a factor of p less than the conventional algorithm. If the main memory size is M words, we get the inequality $(k+4)(\frac{n}{p} + k) \leq M$, or $p \geq \frac{n(k+4)}{M-k(k+4)} \approx \frac{n(k+4)}{M}$ as the minimum number of slow memory accesses.

We can avoid all redundant flops since we can "leave behind" the components of $[x, Ax, ..., A^k x]$ in memory. This phenomenon is unfortunately limited to a 1D mesh, and higher dimensional meshes will again have some redundant work, as they did in the parallel case.

By unrolling the loop in Alg. SA2 (but at the cost of more fast memory), the latency of the read from slow memory can again be overlapped with computation.

Now we consider matrices with bandwidth $b > 1$, i.e. band matrices. The conventional algorithm for $b > 1$ differs very little from the $b = 1$ base described above. The costs are $(4b+1)kn$ flops,

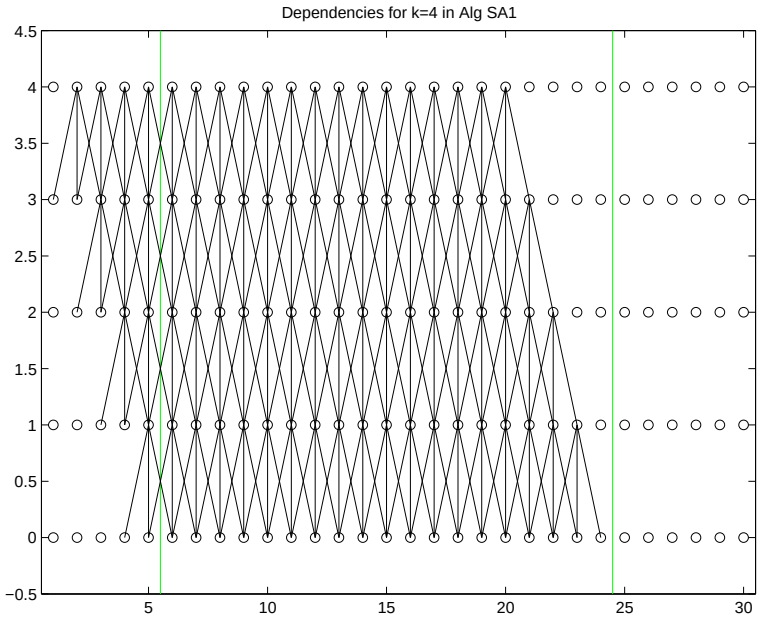


Figure 11: Local dependencies in SA1 for $[Ax, ..., A^4x]$ for tridiagonal matrix

$(2b+1)kn$ words read from slow memory, and kp accesses to slow memory. The fast memory required is $(k+1)n + (2b+1)\frac{n}{p}$.

SA1 for bandwidth $b > 1$ differs from $b = 1$ as follows: Instead of leaving rows $s_q - k$ through $s_q - 1$ of A in memory, we must leave rows $s_q - kb$ through $s_q - 1$. Altogether, the costs are $(4b+1)kn$ flops as before, $(2b+1)n$ words read from slow memory (the matrix is read once), and p accesses to slow memory. The fast memory required is $(k+1)n$ words for the vector entries plus $(2b+1)(\frac{n}{p} + kb)$ words for the matrix entries.

SA2 for bandwidth $b > 1$ also differs from $b = 1$ by needing to keep rows $s_q - kb$ through $s_q - 1$ of A and $[x, Ax, ..., A^k x]$ in memory. The number of flops and slow memory accesses are the same as for SA1. The fast memory required goes down to $(2b+1)(\frac{n}{p} + kb)$ words for the matrix entries and $(k+1)(\frac{n}{p} + kb)$ for the vector entries, or $(2b+k+2)(\frac{n}{p} + kb)$ in all. The number of words read from slow memory increases to $(k+1)n + (2b+1)n$, since the vectors and the matrix need to be read in once.

3.2 2D and 3D Meshes

With the 1D mesh, the natural ordering of the unknowns had several attractive properties: the matrix and vector entries were ordered to minimize the number of accesses to slow memory (i.e. the "boundary vertices" were always contiguous to each partition), and all redundant flops could be avoided because the required data was already in memory, without requiring extra fast memory. Neither of these is possible for 2D or 3D meshes (or for a general graph), but we will nevertheless minimize the amount of redundant work and number of slow memory accesses, as we did in the parallel case. This is more difficult in the sequential than the parallel case, because in the parallel case we could have the sending processor pack up all the desired vectors entries (matrix entries were not communicated) into a single contiguous message to be sent. This could be modeled by using a slightly lower communication bandwidth to account for the copying (which may be done anyway by the communication layer). In contrast, in the sequential case, there is generally no opportunity to reorder data in the slow memory: If data is not contiguous, either more accesses are needed (and so more latency costs are incurred), or else more data than needed is fetched (and so more bandwidth costs are incurred).

To illustrate, consider the grid points associated with a single partition, as shown in left of Figure 12. The unknowns within each numbered region are ordered contiguously, and the regions are ordered as shown. Thus, when the North region requires data, it can read regions 1 through 5 in one access. Similarly, the NE region can read region 5, the E region can read regions 4 through 8, the SE region can read region 8, the S region can read regions 7 through 11, and the SW region can read region 11. However, the W region needs 10, 11, 12, 1 and 2, and so requires 2 slow memory accesses (or else fetching regions 1 through 12 in one access and throwing away the unneeded data). Since the adjacency graph of the regions 1 through 12 has a cycle, one can easily see that no linear order exists requiring only 1 slow memory access for all the required data.

Unlike the 1D case, we will store x and the other vectors $[Ax, ..., A^k x]$ separately, using the order shown in left of Figure 12.

The right of Figure 12 shows a similar, simpler ordering with the same number of slow memory accesses, and with simpler indexing, but which accesses slightly more words than necessary from slow memory (the triangles 2, 4, 7 and 10 are unnecessarily sent to the corner processors). The simpler indexing could result in a faster algorithm.

Figuring out orderings of grid points analogous to these, but for general graphs, that minimize the number of slow memory accesses, may be reduced to an instance of the Travelling Salesman Problem, see section 3.5.

Since the matrix entries are read-only, we can minimize the number of slow memory accesses to the matrix by using extra slow memory (which is cheap) to store some rows of A redundantly, so that the needed rows of A for any region are always stored contiguously. We will henceforth assume this has been done as a preprocessing step, and not count its costs in the Summary Tables. This extra slow memory, like other extra costs, is proportional to the size of the boundary, and so is asymptotically small.

Using the above assumptions and data layouts, the conventional sequential algorithm works analogously to Alg SA0 in Section 3.1: The cost is $9kn^2$ flops, $5kn^2$ words read from slow memory in kp slow memory accesses, and $(k+1)n^2$ fast memory words for the vectors and $5\frac{n^2}{p}$ fast memory words for the matrix.

Alg SA1: New Sequential Approach 1 with Fast/Slow Memory for 2D mesh with 5 point stencil
... assume matrix stored in slow memory in p equal sized chunks
... where chunk q consists of rows s_q through e_q along with
... B "boundary rows" needed for redundant computation of border regions
... assume $k+1$ vectors $[x, Ax, ..., A^k x]$ all fit in fast memory;
... ignore boundaries $q = 1$ and $q = p$;
for $q = 1$ to p
    read rows s_q through $e_q + B$ of A from slow memory
    ... note: rows $e_q + 1$ through $e_q + B$ are redundant copies of "boundary rows"
    compute locally dependent components of $A^j x$ as shown in Figure 5
    ... reusing previously computed red circles
endfor

Alg SA1 costs $9kn^2$ flops, accesses slow memory p times, and reads $5n^2 + 20kn^{1/2} + 10pk^2$ words from slow memory.

Alg SA2: New Sequential Approach 2 with Fast/Slow Memory for 2D mesh with 5 point stencil
... assume matrix stored in slow memory in p equal sized chunks
... where chunk q consists of rows s_q through e_q along with
... B "boundary rows" needed for redundant computation of border regions
... assume x stored in slow memory in corresponding p chunks, where
... each chunk internally ordered as in the left of Figure 12;
... assume k vectors $[Ax, ..., A^k x]$ stored analogously to x
... ignore boundaries $q = 1$ and $q = p$;
for $q = 1$ to p
    read rows s_q through $e_q + B$ of A from slow memory
    ... note: rows $e_q + 1$ through $e_q + B$ are redundant copies of "boundary rows"
    read rows s_q through e_q of x from slow memory
    ... this costs one slow memory access
    read needed rows of x from N, NE, E, SE, S, SW, W and NW regions
    ... this costs 9 slow memory accesses
    compute locally dependent components of $A^j x$ as shown in Figure 5

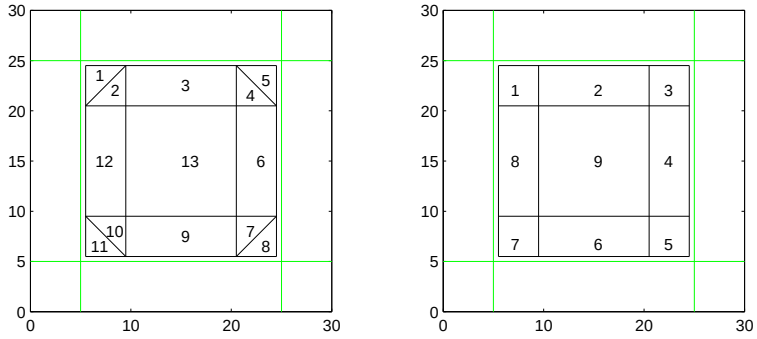


Figure 12: Ordering the unknowns in a 2D Mesh for contiguity

write rows s_q through e_q of $[Ax, ..., A^k x]$ to slow memory
endfor

The costs for Alg SA2 are $12p$ slow memory accesses, $(6+k)n^2 + 24knp^{1/2} + 12pk^2$ words read from or written to slow memory, and $9kn^2 + 18k^2np^{1/2} + 6k^3p$ flops, p times as many as PA1.

A similar counting exercise leads to the other entries in the Summary Table in the next section.

3.3 Summary of Sequential Complexity of Computing $[Ax, ..., A^k x]$ on Meshes

In the Summary Table for Sequential Algorithms, "Acc" is the number of accesses of slow memory (reads or writes), "MWords" is the total number of matrix entries accessed, "VWords" is the total number of vector entries accessed, "Flops" is the number of floating point operations, "MMem" is the fast memory required for matrix entries, and "VMem" is the fast memory required for vector entries.

Lower order terms are sometimes omitted for clarity.

We consider the special case of stencil matrices, where in addition to the graph being a stencil, each row of the matrix has the same nonzero entries (modulo boundaries). In this case no matrix entries need to be fetched from slow memory (MWords=0) and at most MMem $\leq (2b+1)^d$ words are needed to store the matrix entries for a d-dimensional mesh, not $(2b+1)^d \frac{n^d}{p}$. Otherwise the table entries do not change.

Table 2: Summary Table for Sequential Algorithms (some lower order terms omitted)

Problem	Costs	Conventional Approach	Sequential Approach 1	Sequential Approach 2
1D mesh $b = 1$	Acc	kp	p	p
	MWords	$3kn$	$3n$	$3n$
	VWords	0	0	$(k+1)n$
	Flops	$5kn$	$5kn$	$5kn$
	MMem	$3\frac{n}{p}$	$3(\frac{n}{p}+k)$	$3(\frac{n}{p}+k)$
	VMem	$(k+1)n$	$(k+1)n$	$(k+1)(\frac{n}{p}+k)$
1D mesh $b \geq 1$	Acc	kp	p	p
	MWords	$(2b+1)kn$	$(2b+1)n$	$(2b+1)n$
	VWords	0	0	$(k+1)n$
	Flops	$(4b+1)kn$	$(4b+1)kn$	$(4b+1)kn$
	MMem	$(2b+1)\frac{n}{p}$	$(2b+1)(\frac{n}{p}+bk)$	$(2b+1)(\frac{n}{p}+bk)$
	VMem	$(k+1)n$	$(k+1)n$	$(k+1)(\frac{n}{p}+bk)$
2D mesh 5 pt stencil	Acc	kp	p	$12p$
	MWords	$5kn^2$	$5n^2+20knp^{1/2}+10pk^2$	$5n^2+20knp^{1/2}+10pk^2$
	VWords	0	0	$(k+1)n^2+4knp^{1/2}+2pk^2$
	Flops	$9kn^2$	$9kn^2$	$9kn^2+18k^2np^{1/2}+6k^3p$
	MMem	$5\frac{n^2}{p}$	$5\frac{n^2}{p}+20k\frac{n}{p^{1/2}}+10k^2$	$5\frac{n^2}{p}+20k\frac{n}{p^{1/2}}+10k^2$
	VMem	$(k+1)n^2$	$(k+1)n^2$	$(k+1)\frac{n^2}{p}+4k\frac{n}{p^{1/2}}+2k^2$
2D mesh 9 pt stencil	Acc	kp	p	$12p$
	MWords	$9kn^2$	$9n^2+36knp^{1/2}+36pk^2$	$9n^2+36knp^{1/2}+36pk^2$
	VWords	0	0	$(k+1)n^2+4knp^{1/2}+4pk^2$
	Flops	$17kn^2$	$17kn^2$	$17kn^2+34k^2np^{1/2}+\frac{68}{3}k^3p$
	MMem	$9\frac{n^2}{p}$	$9\frac{n^2}{p}+36k\frac{n}{p^{1/2}}+36k^2$	$9\frac{n^2}{p}+36k\frac{n}{p^{1/2}}+36k^2$
	VMem	$(k+1)n^2$	$(k+1)n^2$	$(k+1)\frac{n^2}{p}+4k\frac{n}{p^{1/2}}+4k^2$
2D mesh $(2b+1)^2$ pt stencil	Acc	kp	p	$12p$
	MWords	$(2b+1)^2kn^2$	$(2b+1)^2(n^2+4bknp^{1/2}+4pb^2k^2)$	$(2b+1)^2(n^2+4bknp^{1/2}+4pb^2k^2)$
	VWords	0	0	$(k+1)n^2+4bknp^{1/2}+4pb^2k^2$
	Flops	$(8b^2+8b+1)kn^2$	$(8b^2+8b+1)kn^2$	$(8b^2+8b+1)\cdot(kn^2+2bk^2np^{1/2}+\frac{4}{3}b^2k^3p)$
	MMem	$(2b+1)^2\frac{n^2}{p}$	$(2b+1)^2(\frac{n^2}{p}+4bk\frac{n}{p^{1/2}}+4b^2k^2)$	$(2b+1)^2(\frac{n^2}{p}+4bk\frac{n}{p^{1/2}}+4b^2k^2)$
	VMem	$(k+1)n^2$	$(k+1)n^2$	$(k+1)\frac{n^2}{p}+4bk\frac{n}{p^{1/2}}+4b^2k^2$
3D mesh 7 pt stencil	Acc	kp	p	$O(p)$
	MWords	$7kn^3$	$7n^3+42kn^2p^{1/3}+O(k^2np^{2/3})$	$7n^3+42kn^2p^{1/3}+O(k^2np^{2/3})$
	VWords	0	0	$(k+1)n^3+6kn^2p^{1/3}+O(k^2np^{2/3})$
	Flops	$13kn^3$	$13kn^3$	$13kn^3+39k^2n^2p^{1/3}+O(k^3np^{2/3})$
	MMem	$7\frac{n^3}{p}$	$7\frac{n^3}{p}+42k\frac{n^2}{p^{2/3}}+O(k^2\frac{n}{p^{1/3}})$	$7\frac{n^3}{p}+42k\frac{n^2}{p^{2/3}}+O(k^2\frac{n}{p^{1/3}})$
	VMem	$(k+1)n^3$	$(k+1)n^3$	$(k+1)\frac{n^3}{p}+6k\frac{n^2}{p^{2/3}}+O(k^2\frac{n}{p^{1/3}})$
3D mesh 27 pt stencil	Acc	kp	p	$O(p)$
	MWords	$27kn^3$	$27n^3+162kn^2p^{1/3}+O(k^2np^{2/3})$	$27n^3+162kn^2p^{1/3}+O(k^2np^{2/3})$
	VWords	0	0	$(k+1)n^3+6kn^2p^{1/3}+O(k^2np^{2/3})$
	Flops	$53kn^3$	$53kn^3$	$53kn^3+159k^2n^2p^{1/3}+O(k^3np^{2/3})$
	MMem	$27\frac{n^3}{p}$	$27\frac{n^3}{p}+162k\frac{n^2}{p^{2/3}}+O(k^2\frac{n}{p^{1/3}})$	$27\frac{n^3}{p}+162k\frac{n^2}{p^{2/3}}+O(k^2\frac{n}{p^{1/3}})$
	VMem	$(k+1)n^3$	$(k+1)n^3$	$(k+1)\frac{n^3}{p}+6k\frac{n^2}{p^{2/3}}+O(k^2\frac{n}{p^{1/3}})$
3D mesh $(2b+1)^3$ pt stencil	Acc	kp	p	$O(p)$
	MWords	$(2b+1)^3kn^3$	$(2b+1)^3\cdot(n^3+6bkn^2p^{1/3}+O(b^2k^2np^{2/3}))$	$(2b+1)^3\cdot(n^3+6bkn^2p^{1/3}+O(b^2k^2np^{2/3}))$
	VWords	0	0	$(k+1)n^3+6bkn^2p^{1/3}+O(b^2k^2np^{2/3})$
	Flops	$(2(2b+1)^3-1)kn^3$	$(2(2b+1)^3-1)kn^3$	$(2(2b+1)^3-1)\cdot(kn^3+3bk^2n^2p^{1/3}+O(b^2k^3np^{2/3}))$
	MMem	$(2b+1)^3\frac{n^3}{p}$	$(2b+1)^3\cdot(\frac{n^3}{p}+6bk\frac{n^2}{p^{2/3}}+O(b^2k^2\frac{n}{p^{1/3}}))$	$(2b+1)^3\cdot(\frac{n^3}{p}+6bk\frac{n^2}{p^{2/3}}+O(b^2k^2\frac{n}{p^{1/3}}))$
	VMem	$(k+1)n^3$	$(k+1)n^3$	$(k+1)\frac{n^3}{p}+6bk\frac{n^2}{p^{2/3}}+O(b^2k^2\frac{n}{p^{1/3}})$

3.4 General Graphs

We use notation defined in section 2.4. In the pseudocode below "read" means from slow to fast memory, and "write" means from fast to slow.

SA0 - Conventional sequential algorithm for a general graph
... assume the vectors fit in fast memory but the matrix does not
for $i = 1$ to k
for $q = 1$ to p
read all rows j of A such that some $x_j^{(i)} \in G_q^{(i)}$
compute all $x_j^{(i)}$ in $G_q^{(i)}$
end for
end for

SA1 for a general graph
... emulate PA1, assuming the vectors fit in fast memory but the matrix does not
... no redundant arithmetic required
$S = \phi$ ... S is set of $x_j^{(i)}$ computed so far
for $q = 1$ to p
$S' = \{x_j^{(i)} : R^{(0)}(x_j^{(i)}) \subset G_q^{(0)} \cup S^{(0)}\} - S$
... S' = set of $x_j^{(i)}$ that depend only on current and previous $x_m^{(0)}$
read all rows j of A such that some $x_j^{(i)} \in S'$
... may store some rows of A redundantly to reduce # reads to 1
compute all $x_j^{(i)}$ in S' (in order of increasing i)
$S = S \cup S'$
endfor

SA2 for a general graph
... emulate PA1, assuming neither vectors nor matrix fit in fast memory
... will be redundant arithmetic as in PA1
$S = \phi$ ... S is set of $x_j^{(i)}$ computed so far
for $q = 1$ to p
read $G_q^{(0)}$
read all rows j of A such that some $x_j^{(i)} \in R(G_q)$
... may store some rows of A redundantly to reduce # reads to 1
compute all $x_i^{(j)}$ in L_q
for all $r \neq q$, read $R_r^{(0)}(G_q)$
... possible to optimize order in which $x_j^{(0)}$ stored to minimize # slow memory accesses
... see next section for details
compute remaining $x_j^{(i)}$ in $R(G_q) - L_q$
write $G_q^{(1:k)}$
end for

3.5 Optimizing the Order of Unknowns in SA2

As illustrated in Figure 12, the order in which vector components are stored within each block influences the number of slow memory accesses needed to read the data needed from neighboring blocks, namely the data $R_r^{(0)}(G_q)$ that block q needs from block r, for all $r \neq q$. The left part of Figure 12 shows the best order for a 2D mesh with a 5 point stencil. (The components within each block can be numbered in any order, but all the components in block i must be numbered before those in block $i+1$, and so on.) In this case, where each block has 8 neighboring blocks, 8 (simultaneous) accesses is clearly a lower bound. If we insist on reading only the needed data, then the best we can do is 9 accesses, as discussed in section 3.2.

Alternatively, when regions 1, 2, 10, 11 and 12 are needed, all regions 1 through 12 could be fetched and regions 3 through 9 discarded. But in this section we consider solutions that fetch only the required data, and so always minimize the bandwidth costs.

Furthermore, the order in which we access blocks is important. For example, for the 2D mesh, if we access blocks from left to right in each processor row, then the needed data from the previous block is still in fast memory and does not need to be accessed. So we also need to choose the order in which to access blocks. We call this *block ordering*, as opposed to the *component ordering* within an individual block, as discussed in the first paragraph.

In this section we ask how to determine the optimal block ordering and component ordering for SA2 for a general graph, by reducing the question to an instance of the Travelling Salesman Problem (TSP).

For the rest of this subsection, we let n denote the dimension of the matrix A. We will start by assuming a block ordering is given, so that the blocks of the vector x are $B_1, \ldots, B_p$, where each B_i is a disjoint subset of $\{1, ..., n\}$, and show how to choose the optimal component ordering within each block. Later we will show how to choose the optimal block ordering. Let $|B_i|$ denote the number of elements in block B_i. Let $N_{i,j}$ denote the set of elements of block B_i needed when computing block B_j. Let $E_{i,j}$ denote the set of elements of block B_i needed to be fetched from slow memory when computing for block B_j. The set $E_{i,j}$ is fixed by the ordering of the blocks–if block B_j comes immediately after block B_l, then $E_{i,j} = N_{i,j} - N_{i,l}$, i.e., we only fetch elements which are not already in fast memory. So, there will always be an implicit ordering of the blocks when we talk about $E_{i,j}$. We call block B_i a neighbor of block B_j if $E_{i,j} \neq \emptyset$.

3.5.1 Component ordering

Let $A(E_{i,j})$ denote the number of accesses required to fetch the set of elements $E_{i,j}$ from slow memory, assuming blocks are processed in increasing order from B_1 to B_p. Therefore, total number of slow memory accesses required is $\mathcal{A} = \sum_{j=1}^{p}(\sum_{i=1}^{p} A(E_{i,j})) = \sum_{i=1}^{p}(\sum_{j=1}^{p} A(E_{i,j}))$. Since the ordering of elements inside block B_i only affects the sum $\mathcal{A}_i = \sum_{j=1}^{p} A(E_{i,j})$, we simply need to optimize A_i independently for block B_i. Given this observation, we now formalize the block level ordering problem for an individual block.

Let B_i be the block under consideration. Without loss of generality, let $1, 2, \ldots, m$ be the elements of block B_i. Also, assume that none of $E_{i,j}$ $(j \neq i)$ are empty (if there is an empty $E_{i,j}$, then we simply remove it from consideration resulting in a smaller p). Similarly, assume that all the elements of B_i are in some $E_{i,j}$ (if not, such element(s) can be placed in a contiguous segment at the end without affecting the optimality of an ordering; for example these would be the components in

region 13 on the left of Figure 12). Let $I_{i,j}$ be the indicator function for whether $a \in E_{i,j}$ ($I_{i,j}(a) = 1$ iff $a \in E_{i,j}$). We now have the following lemma:

Lemma 1 *Let $1, 2, \ldots, m$ be the elements of block B_i. Add a dummy element $m+1$ to the block. Let $m+1 \notin E_{i,j}$ ($1 \leq j \leq p$). Given an ordering ρ of these $m+1$ elements such that $\rho(m+1) = m+1$, we have $A(E_{i,j}) = \sum_{k=1}^{m} I_{i,j}(\rho(k)) \cdot (1 - I_{i,j}(\rho(k+1)))$.*

Proof: $A(E_{i,j})$ is the same as the number of contiguous segments of the set $E_{i,j}$ under the ordering ρ. One way of counting this is to look at the elements of B_i in the order specified by ρ and add 1 to the count whenever we encounter a boundary, i.e., when $\rho(k) \in E_{i,j}$ and $\rho(k+1) \notin E_{i,j}$. Equivalently, for the k-th element in the ordering, we add $I_{i,j}(\rho(k)) \cdot (1 - I_{i,j}(\rho(k+1)))$ to the count. The reason we added a dummy element is to account for the case when the last contiguous segment ends at element m. So, we get $A(E_{i,j}) = \sum_{k=1}^{m} I_{i,j}(\rho(k)) \cdot (1 - I_{i,j}(\rho(k+1)))$. □

Using Lemma 1, we get

$$\begin{aligned} \mathcal{A}_i &= \sum_{j=1}^{p} \sum_{k=1}^{m} I_{i,j}(\rho(k)) \cdot (1 - I_{i,j}(\rho(k+1))) \\ &= \sum_{k=1}^{m} \left(\sum_{j=1}^{p} I_{i,j}(\rho(k)) \cdot (1 - I_{i,j}(\rho(k+1))) \right). \end{aligned}$$

Now, consider the weighted directed complete graph $\mathcal{G}_i = (V_i, E_i)$, $V_i = \{v_1, \ldots, v_{m+1}\}$ (one node for each element of block B_i, v_{m+1} for the dummy element in Lemma 1). Let $wt(v_a, v_b)$ (weight of edge from node v_a to node v_b) be $\sum_{j=1}^{p} I_{i,j}(a) \cdot (1 - I_{i,j}(b))$–the contribution to the total count of the number of disk accesses if element b is placed immediately after element a. Consider an ordering ρ_i of the elements of block B_i (such that $\rho_i(m+1) = m+1$). The total number of disk accesses due to this ordering is $\mathcal{A}_i = \sum_{k=1}^{m} \left(\sum_{j=1}^{p} I_{i,j}(\rho_i(k)) \cdot (1 - I_{i,j}(\rho_i(k+1))) \right)$. The total weight of the path $v_{\rho_i(1)}, v_{\rho_i(2)}, \ldots, v_{\rho_i(m+1)}$ is also $\mathcal{A}_i$. Equivalently, we can say that the Hamiltonian path ending at node v_{m+1} has the same weight as the number of disk accesses for the corresponding ordering. Thus, an optimal ordering (with the constraint of $m+1$ being the last element) corresponds to the lowest weight Hamiltonian path (with the constraint of v_{m+1} being the last node). By setting $wt(v_{m+1}, v_a)$ ($1 \leq a \leq m$) large enough, v_{m+1} will be the last node in any lowest weight Hamiltonian path. In fact, $wt(v_{m+1}, v_a) = 1 + \max_{k=1}^{m} \sum_{j=1}^{p} I_{i,j}(a)$ does the trick. Since $wt(v_{m+1}, v_a)$ is independent of a (for $1 \leq a \leq m$), the lowest weight Hamiltonian path corresponds to the lowest weight Travelling Salesman tour by using the edge between v_{m+1} (since it lies at the end in any optimal Hamiltonian path) and the first node in the Hamiltonian path. So, the problem can also be formulated as a Travelling Salesman problem.

In summary, to find the optimal ordering of the components of the block B_i, construct the graph $\mathcal{G}_i$ (discussed in the previous paragraph) and find the lowest weight Hamiltonian path. The ordering defined by the Hamiltonian path is the optimal ordering for the components of the block.

3.5.2 Reducing the problem size for component ordering

It appears that the size of the component ordering problem is $O(m^2)$, where m is the number of components of each block needed by other blocks. In general m will grow with problem dimension n and be quite large. Here we show how to reduce the TSP problem size for each block to at

most its number of neighboring blocks, which is usually quite small, for example $3^d - 1$ in the case of a d-dimensional mesh, independent of the number of mesh points. The intuition, again from Figure 12, is that the components can be put into equivalence classes (numbered there on the left from 1 through 12) each of which is needed by the same set of neighboring blocks, and then the equivalence classes ordered. So here we will formally construct the equivalence relation on components, and show that in any optimal ordering, equivalent components are numbered consecutively.

For the above graph $\mathcal{G}_i$, we say that node v_a is related to node v_b ($v_a R_i v_b$) iff $wt(v_a, v_b) = wt(v_b, v_a) = 0$. Clearly, the relation R_i is an equivalence relation. The next 3 lemmas make several observations about the relation R_i.

Lemma 2 *If $v_a R_i v_b$ then, $a \in E_{i,j}$ iff $b \in E_{i,j}$ and vice versa. In other words, the equivalence relation R_i is also defined as being contained in the same set of sets $E_{i,j}$.*

Proof: First of all, we note that v_{m+1} is not related to any other node, since $wt(v_{m+1}, v_a) \neq 0$ for any $1 \leq a \leq m$. Also, there is no $1 \leq a \leq m$ such that $a \in E_{i,j}$ iff $m+1 \in E_{i,j}$ since each a is in some $E_{i,j}$ but $m+1$ is not in any $E_{i,j}$. So, we only need to consider other elements/nodes. $wt(v_a, v_b) = \sum_{j=1}^{p} I_{i,j}(a) \cdot (1 - I_{i,j}(b)) = 0$ implies that for all $1 \leq j \leq p$, $I_{i,j}(a) \cdot (1 - I_{i,j}(b)) = 0$ (all these terms are non-negative). Similarly, $wt(v_b, v_a) = \sum_{j=1}^{p} I_{i,j}(b) \cdot (1 - I_{i,j}(a)) = 0$ implies that for all $1 \leq j \leq p$, $I_{i,j}(b) \cdot (1 - I_{i,j}(a)) = 0$. This implies that, for all $1 \leq j \leq p$, $I_{i,j}(a) = I_{i,j}(b)$ which means that $a \in E_{i,j}$ iff $b \in E_{i,j}$. Similarly, if $a \in E_{i,j}$ iff $b \in E_{i,j}$, then for all $1 \leq j \leq p$, $I_{i,j}(a) = I_{i,j}(b)$ which implies that $I_{i,j}(b) = I_{i,j}(b) \cdot I_{i,j}(a) = I_{i,j}(a)$ for all $1 \leq j \leq p$, which implies that $wt(v_a, v_b) = wt(v_b, v_a) = 0$. This completes the proof. □

From Lemma 2, we get the following–if $v_a R_i v_b$, then for any v_c, $wt(v_a, v_c) = wt(v_b, v_c)$ and $wt(v_c, v_a) = wt(v_c, v_b)$. Now, consider an optimal ordering ρ_i of the elements of B_i. Lemma 2 implies that there are optimal orderings in which all elements which are equivalent are placed contiguously.

Lemma 3 *For any 3 distinct nodes v_a, v_b and v_c, $wt(v_a, v_b) + wt(v_b, v_c) \geq wt(v_a, v_c)$.*

Proof: We consider 4 possible cases:

1. $a = m+1$: $wt(v_{m+1}, v_b) + wt(v_b, v_c) - wt(v_{m+1}, v_c) = wt(v_b, v_c) \geq 0$ ($wt(v_{m+1}, v_b)$ is the same for all $b \neq m+1$).

2. $b = m+1$: $wt(v_a, v_{m+1}) + wt(v_{m+1}, v_c) - wt(v_a, v_c) \geq wt(v_a, v_{m+1}) - wt(v_a, v_c)$. But, $I_{i,j}(a) - I_{i,j}(a) \cdot (1 - I_{i,j}(c)) \geq 0$ for all a, c. Since $wt(v_a, v_{m+1}) = \sum_{j=1}^{p} I_{i,j}(a)$ and $wt(v_a, v_c) = \sum_{j=1}^{p} I_{i,j}(a) \cdot (1 - I_{i,j}(b))$, the inequality holds true.

3. $c = m+1$: Consider

$$wt(v_a, v_b) + wt(v_b, v_{m+1}) - wt(v_a, v_{m+1}) = \sum_{j=1}^{p} \left(I_{i,j}(a) \cdot (1 - I_{i,j}(b)) + I_{i,j}(b) - I_{i,j}(a) \right).$$

   Since $I_{i,j}(a) \cdot (1 - I_{i,j}(b)) + I_{i,j}(b) - I_{i,j}(a) \geq 0$ for all a, b, the inequality holds true here too.

4. $a, b, c \neq m+1$: Consider

$$\begin{aligned} I_{i,j}(a)(1-I_{i,j}(b)) + I_{i,j}(b)(1-I_{i,j}(c)) - I_{i,j}(a)(1-I_{i,j}(c)) &= \\ I_{i,j}(b)(1-I_{i,j}(a)-I_{i,j}(c)) + I_{i,j}(a)I_{i,j}(c) &\geq \\ 1-I_{i,j}(a)-I_{i,j}(c)+I_{i,j}(a)I_{i,j}(c) &\geq 0. \end{aligned}$$

Since

$$wt(v_a,v_b)+wt(v_b,v_c)-wt(v_a,v_c) = \sum_{j=1}^{p} I_{i,j}(a)(1-I_{i,j}(b))+I_{i,j}(b)(1-I_{i,j}(c))-I_{i,j}(a)(1-I_{i,j}(c)),$$

the inequality holds.

□

Lemma 4 *There exists an optimal Hamiltonian path such that if $v_a R_i v_b$, and v_a comes before v_b in the path, then there is no v_c between v_a and v_b in the path such that $\neg v_a R_i v_c$.*

Proof: Consider an optimal Hamiltonian path such that the condition in the lemma does not hold true. By our choice of weights, v_{m+1} would be the last node in the path. Let v_a and v_b be nodes such that the following holds: v_a comes before v_b in the path, $v_a R_i v_b$ and $\neg v_a R_i v_c$ for any v_c which lies between v_a and v_b in the path. This must be possible simply because we assumed the condition in the lemma to be false. Let $v_d$be the node immediately after v_a, v_e be the node immediately after v_b and v_f be the node immediately before v_b. If we move node v_b to between v_a and v_d, then the change in the weight of the path is

$$\begin{aligned} wt(v_a,v_b)+wt(v_b,v_d)+wt(v_f,v_e)-wt(v_a,v_d)-wt(v_f,v_b)-wt(v_b,v_e) &= \\ 0+wt(v_a,v_d)+wt(v_f,v_e)-wt(v_a,v_d)-wt(v_f,v_b)-wt(v_b,v_e) &= \text{(Lemma 2)} \\ wt(v_f,v_e)-wt(v_f,v_b)-wt(v_b,v_e) &\leq 0 \text{ (Lemma 3)} \end{aligned}$$

Since the original path was optimal, the cost must not decrease, hence the change must be 0. Applying this procedure of putting together related nodes eventually terminates in a Hamiltonian path such that the condition in the lemma is true. Furthermore, after each application of this procedure the cost did not change, which implies that the final path is also optimal. □

Lemma 4 tells us that we can indeed reduce the problem size to ordering only equivalence classes of components, where two components are equivalent if and only if they are needed to compute the same set of blocks.

3.5.3 Block ordering

We now construct a weighted directed graph where there is one vertex per block B_i and one edge from every vertex to every other vertex. The weight of the edge e pointing from B_i to B_j will be the memory cost of processing B_j immediately after B_i. Clearly, given this graph, the goal is to find the lowest weight Hamiltonian path. As before, we can add a "dummy node" and appropriately weighted edges to convert to an instance of TSP.

Now we discuss the edge weights. If e points from B_i to B_j, so that B_j is processed immediately after B_i, the discussion in previous subsections tells us how to compute the minimum number of

slow memory accesses needed to process B_j. This could be used as a weight by itself, if latency were dominant. But we can easily take the full cost into account, including bandwidth and latency, by setting the edge weight to the sum of $\alpha\cdot$ (# slow memory accesses) and $\beta\cdot$ (# words fetched from slow memory), where α is latency and β is reciprocal bandwidth. The number of words fetched from slow memory does not depend on the ordering, and is a by-product of the graph traversals needed to compute all the regions $R_r^{(0)}(G_q)$ needed by SA2.

3.5.4 Other problem formulations

In the previous section, we only considered exact solutions, i.e., we fetch only those elements which are required. However, if we sometimes fetch more elements than needed when working on a block, we might be able to further lower the latency cost, at the price of higher bandwidth cost. For example, suppose we merged $E_{i,j}$ and $E_{i,j+1}$ ($E_{i,j} \neq E_{i,j+1}$), so that while computing block B_j we fetched the components in $E_{i,j} \cup E_{i,j+1}$, and similarly for block B_{j+1}. However, by merging the sets, we have reduced the number of equivalence classes in the component ordering problem, which might allow for fewer slow memory accesses.

If latency is the dominant cost we can go further and discuss solutions limited to a single slow memory access per E_{ij}, and choose the component ordering to minimize the bandwidth cost. In this case, the cost of accessing E_{ij} given a component ordering will be $h - l + 1$, where x_h is the highest numbered component in E_{ij} and x_l is the lowest numbered component (because all of $\{x_l, x_{l+1}, ..., x_h\}$ will needed to be fetched).

The best formulation may depend on other details of the performance model. Here we have been assuming a simple latency + bandwidth model, but depending on opportunities for overlap, parallelism, prefetching, etc. a different model may be appropriate. There are also many options for approximate solutions to the resulting combinatorial optimization problems, like TSP.

3.6 Stencils

Now we discuss how computing $[x, Ax, ..., A^k x]$ simplifies when A is truly a stencil operator, i.e. not only is the sparsity pattern described by mesh as before, but the values of the matrix are the same in every row (modulo mesh boundaries). As in the parallel case (section 2.5) this means that we do not need to store any matrix entries, but it also means that we do not need to get them from slow memory. This makes SA1 and the conventional approach identical, since they assume that all the vectors fit in fast memory, so there is no slow memory traffic to try to reduce.

SA2 simplifies by only needing to move vectors between fast and slow memory, otherwise leaving the algorithm unchanged. The entries for SA2 in Table 2 change by zeroing out both MWords (the number of matrix entries read from slow memory) and MMem (the number of words of fast memory needed to store matrix entries). The number of slow memory accesses will also decrease by p for meshes of dimension 2 and higher. SA2 thus has the same optimality property as before: it moves the vectors between fast and slow memory the minimal number of times (once). As in the parallel case, further simplifications are possible if only the final vector $A^k x$ is desired, or if some nonzero entries of the matrix are identical. See section 8 for related work.

4 Asymptotic Performance Models

We consider how to asymptotically minimize the time to compute $[Ax, ..., A^{\bar{k}}x]$ for matrices with stencil graphs. In other words, the graph of the matrix is assumed to be a d-dimensional mesh with a $(2b+1)^d$ point stencil. We will treat all dimensions d simultaneously by using the notation $\gamma = \frac{n}{p^{1/d}}$. We can think of γ as a measure of problem size per processor, since it is the d-th root of the number of components per vector per processor. We also note that $2d/\gamma$ is the surface-to-volume ratio of a d-dimensional cube of side length γ.

We will compare the conventional parallel method (run $k = \bar{k}$ times) with the new method, where $\bar{k}/k$ groups of k matrix-vector-products are computed using $O(\bar{k}/k)$ messages. We will choose k to minimize the time of the new algorithm. We also compare the new method with the conventional method using overlap of communication and computation, and ask when this is sufficient to hide communication costs.

We let α be the message latency, β be the reciprocal bandwidth (so it takes $\alpha + n\beta$ seconds to send a message of length n), and f be the time per flop. Sample values are $f = 1$ns, $\alpha = 190\mu$s and $\beta = 10^{-4}\mu$s/byte (for 10 Gigabit Ethernet running 802.3ae), and $\alpha = 5700\mu$s and $\beta = .016\mu$s/byte (for a 15000 RPM Seagate ST373307 disk). In both cases α exceeds β by at least five orders of magnitude, and f is much smaller again. Patterson [20] suggests that this gap between latency and bandwidth will continue to increase exponentially for a variety of technologies (memory, network and disk).

4.1 Parallel Algorithms

Combining this notation with the analysis leading to Table 1, we get that the running time for the conventional algorithm to compute $[Ax, ..., A^k x]$ is

$$T_{PA0}(k) = O(\alpha k + \beta bk\gamma^{d-1} + fb^d k\gamma^d) \tag{1}$$

and that the running time for either new algorithm is

$$T_{PA1,2}(k) = O(\alpha + \beta bk(\gamma^{d-1} + \delta_d bk\gamma^{d-2}) + fb^d k(\gamma^d + bk\gamma^{d-1})) \tag{2}$$

where $\delta_d = 0$ if $d = 1$ and $\delta_d = 1$ if $d > 1$. We use this notation in order to analyze all values of d at once.

The ultimate goal is to compute $[Ax, ..., A^{\bar{k}}x]$. The conventional algorithm will take time $T_{PA0}(\bar{k})$. We will use the new algorithm $\frac{\bar{k}}{k}$ times on chunks of size k, taking time $\frac{\bar{k}}{k}T_{PA1,2}(k)$. The optimization problem is to choose k to minimize this quantity:

$$\frac{\bar{k}}{k}T_{PA1,2}(k) = O(\alpha\frac{\bar{k}}{k} + \beta b\bar{k}(\gamma^{d-1} + \delta_d bk\gamma^{d-2}) + fb^d\bar{k}(\gamma^d + bk\gamma^{d-1})) \tag{3}$$

When α is sufficiently large (suppose we are doing message passing by the post office), and so latency dominates all the bandwidth and flop terms, it is clearly best to minimize the number of messages in the new algorithm, i.e. to set $k = \bar{k}$, leading to a speedup of $O(\bar{k})$. In other words, $[Ax, ..., A^{\bar{k}}x]$ can be computed in approximately the same time as Ax (or within a constant factor of this time, since constants are hidden by our use of $O()$).

If α is not this large, then choosing the best k is more interesting. The dominant bandwidth and flop terms in (3) (i.e. those proportional to β and f and with the highest powers of γ) are identical

to those in $T_{PA0}(\bar{k})$, and dependent only on $\bar{k}$. The latency term in (3) decreases proportionally to k, and the smaller bandwidth and flop terms increase proportionally to k. The minimizing value of k is easily found to be

$$k_{\min} = \min(k, \max(1, \left(\frac{f}{\alpha} b^{d+1}\gamma^{d-1} + \delta_d \frac{\beta}{\alpha} b^2 \gamma^{d-2}\right)^{-1/2})) \tag{4}$$

Notice that $k_{\min}$ increases with increasing latency α, and decreases with increasing problem size per processor γ. For $k_{\min}$ to exceed 1 requires both that $\alpha > fb^{d+1}\gamma^{d-1}$, roughly that α exceeds the floating point work on the boundary, and that $\alpha > \delta_d \beta b^2 \gamma^{d-2}$, which is likely.

The minimum running time with the new algorithm (assuming $1 < k_{\min} < k$) is therefore

$$\begin{aligned} T_{PA1,2,min}(\bar{k}) &\equiv \frac{\bar{k}}{k_{\min}} T_{PA1,2}(k_{\min}) \\ &= O(\bar{k}[(\alpha(fb^{d+1}\gamma^{d-1} + \delta_d \beta b^2 \gamma^{d-2}))^{1/2} + fb^d\gamma^d + \beta b \gamma^{d-1}]) \end{aligned} \tag{5}$$

When $k_{\min} = 1$, the new algorithm and conventional algorithm are equivalent; when $k_{\min} = k$, the time is given by (2).

When α is very large, the speedup is close to k as expected. When α is not that large, The best that we could hope for is for the new running time to be fast independent of the latency α. More precisely, we ask whether $T_{PA1,2,min}(\bar{k})$ is within a constant factor of the time it would take the conventional algorithm with $\alpha = 0$. In fact, when $\alpha = 0$ both $T_{PA0}(\bar{k})$ and $T_{PA1,2,min}(\bar{k})$ are $O(\bar{k}(fb^d\gamma^d + \beta b\gamma^{d-1}))$, so we need to ask whether the first term in $T_{PA1,2,min}(\bar{k})$ is smaller than this:

$$\text{when is} \quad (\alpha(fb^{d+1}\gamma^{d-1} + \delta_d \beta b^2 \gamma^{d-2}))^{1/2} \leq fb^d\gamma^d + \beta b\gamma^{d-1} \ ?$$

We consider the bandwidth and flop terms separately.

For bandwidth (which is only relevant when $d > 1$), the question is when $(\alpha\beta b^2\gamma^{d-2})^{1/2} \leq \beta b\gamma^{d-1}$, or when $\alpha \leq \beta\gamma^d$. This is easy to interpret: it holds when the time is takes to send the entire local content of a processor $\gamma^d = \frac{n^d}{p}$ is dominated by the bandwidth $\beta\frac{n^d}{p}$, not the latency. Once problem sizes are reasonably large, this is sure to be the case.

For the flop time, the question is when $(\alpha f b^{d+1}\gamma^{d-1})^{1/2} \leq fb^d\gamma^d$, or when $\alpha \leq fb^{d-1}\gamma^{d+1} = fb^{d-1}\left(\frac{n^d}{p}\right)^{1+\frac{1}{d}}$. This is also easy to interpret: it holds when the time it takes to run an $O(N^{1+\frac{1}{d}})$ algorithm on the entire local content of a processor (i.e. $N = \frac{n^d}{p}$) exceeds the latency of one message. When $d = 1$, this means an $O(N^2)$ algorithm, and is very likely to hold for large problem sizes.

In summary, the new algorithm can be used to make the cost of computing any number of matrix-vector products run at a speed that is roughly independent of the communication latency, provided $\alpha \lessapprox \min(\beta\gamma^{d-1}, fb^{d-1}\gamma^{d+1})$ when $d > 1$, or $\alpha \lessapprox f\gamma^2$ when $d = 1$. The limiting factor in achieving this will be the need for enough memory to store $k_{\min}$ vectors locally, since $k_{\min}$ grows as latency increases.

It is worth asking when just overlapping communication and computation in the conventional algorithm is good enough to hide all the latency, making our techniques unnecessary. This happens roughly when $\alpha < fb^d\gamma^d$, which is more restrictive than our condition $\alpha < fb^{d-1}\gamma^{d+1}$.

4.2 Sequential Algorithms

The corresponding asymptotic performance models for sequential algorithms are

$$\begin{aligned} T_{SA0}(k) &= O(\alpha kp + \beta b^d kn^d + fb^d kn^d) \\ T_{SA1}(k) &= O(\alpha p + \beta(b^d n^d + b^{d+1} kn^{d-1} p^{1/d}) + fb^d kn^d) \\ T_{SA2}(k) &= O(\alpha p + \beta((b^d + k)n^d + b^{d+1} kn^{d-1} p^{1/d}) + f(b^d kn^d + b^{d+1} k^2 n^{d-1} p^{1/d})) \end{aligned}$$

Since SA0 and SA1 use roughly the same amount of memory, we can compare their running times directly, and see that SA1 sends k-times fewer messages, sends roughly k-times fewer words, and does only slightly more floating point operations than SA0. So we expect SA1 to be uniformly superior to SA0.

SA2 is designed to use much less memory than either SA0 or SA1, and so a fair comparison is between SA2 and the conventional algorithm consisting of applying SA2(1) k times, where SA2(1) just computes Ax. The running time for this algorithm is easily seen to be $O(k\alpha p + k\beta(b^d n^d + b^{d+1} n^{d-1} p^{1/d}) + kf(b^d n^d + b^{d+1} n^{d-1} p^{1/d}))$, which has a k times larger latency term, up to k times larger bandwidth term (when $b^d \gg k$), and almost the same floating point term. Clearly overlapping communication and computation will benefit SA2(k) at least as much as SA2(1).

Another natural question is to ask under what circumstances SA2 is about as fast as a conventional algorithm with an infinite amount of fast memory available, but where the matrix and $x^{(0)}$ initially reside in slow memory, and the result is eventually supposed to reside in slow memory, an algorithm we call SA3:

$$T_{SA3}(k) = O(\alpha + \beta(b^d + k)n^d + fb^d kn^d)$$

Comparing $T_{SA2}(k)$ to $T_{SA3}(k)$ we see that the bandwidth and floating point costs of SA2 are only slightly larger than for SA3, so the only issue is latency, which is p-times lower for SA3. So a natural question is when SA2's latency cost is less than or equal to its bandwidth and floating point cost. This will be true when the cost of filling up all of fast memory with one fast memory access, and then performing the algorithm on the subset of matrix and vectors filling all of fast memory, is dominated by bandwidth and floating point, which is very likely to be true.

We now turn to the question of optimal speedup for SA2. It can be seen that the optimal speedup for SA2 for general sparse matrices (when $\beta/t_f = \infty$) is upper bounded by $2 + 1.5(nnz/n)$ (nnz/n denotes then number of nonzeros per row of the matrix A). The 1.5 term is due to the 1.5 words per entry of the A. A simple argument for this is that the speedup is roughly

$$\frac{\text{Time to read/write vector and matrix } k \text{ times}}{\text{Time to read 1 vector / write } k \text{ vectors and read matrix once}} = \frac{2k + 1.5(nnz/n)k}{k + 1 + 1.5(nnz/n)} < 2 + 1.5(nnz/n)$$

A similar argument gives us the following upper bound for optimal speedup in the case when β/t_f is finite:

$$\frac{(2(nnz/n) - 1)\, k + (2k + 1.5(nnz/n)k)\, \frac{\beta}{t_f}}{(2(nnz/n) - 1)\, k + (k + 1 + 1.5(nnz/n))\, \frac{\beta}{t_f}} < \frac{2(nnz/n) - 1 + (2 + 1.5(nnz/n))\, \frac{\beta}{t_f}}{2(nnz/n) - 1 + \frac{\beta}{t_f}.}$$

Note that $2(nnz/n) - 1$ is the number of flops performed per entry of the vector x. These upper bounds are tight for large memory size. However, they might not be close to the optimal speedup

for small memory size. To analyze this, we now state some bounds for the optimal speedup of SA2 for d-dimensional stencils with bandwidth b. The following are some scenarios for a d-dimensional stencil with bandwidth b.

1. $m = \infty$, $\frac{\beta}{t_f} = \infty$: In this case, the optimal speedup is the same as the upper bound, i.e., $2 + 1.5(2b+1)^d$.

2. $m = \infty$, $\frac{\beta}{t_f}$ finite: The optimal speedup again equals its upper bound

$$\frac{2(2b+1)^d - 1 + \left(2 + 1.5(2b+1)^d\right)\frac{\beta}{t_f}}{2(2b+1)^d - 1 + \frac{\beta}{t_f}}$$

3. m finite, $\frac{\beta}{t_f} = \infty$: In this case, the optimal speedup is lower bounded by

$$\max\left(\left(2 + 1.5(2b+1)^d\right)\left(1 + \frac{A}{m^{\frac{1}{d+1}}}\right)^{-\left(1+\frac{1}{d}\right)}, 1\right),$$

   where A is a constant[1] which depends on b and d. As can be seen, the lower bound approaches the upper bound as m is increased. For this case, the optimal speedup is obtained by choosing k such that $2bkd = n$.

4. m finite, $\frac{\beta}{t_f}$ finite: The optimal speedup is lower bounded by

$$\max\left(\frac{2(2b+1)^d - 1 + \left(2 + 1.5(2b+1)^d\right)\frac{\beta}{t_f}}{\left(2(2b+1)^d - 1\right)B + \left(1 + \frac{A}{m^{\frac{1}{d+1}}}\right)^{\left(1+\frac{1}{d}\right)}\frac{\beta}{t_f}}, 1\right),$$

   where A is the same constant as in the previous case and B is another constant[2] dependent on d. Figure 13 shows the optimal speedup (obtained from the analytical model) and the lower bounds for the speedup as function of the memory size. Figures 13(a) and 13(b) show the speedups if the β/t_f ratio is 64 (the OOC machine model in Section 5.2). Figures 13(a) and 13(b) show the speedups if the β/t_f ratio is 3.2 (the Clovertown machine model in Section 5.2). As can be seen in these figures, if the memory is not sufficiently large, the optimal speedup drops as the bandwidth b is increased. However, for large enough memory, we observe the speedup increase as the bandwidth b is increased. Another observation is that the lower bound is within a factor of 2 of the optimal speedup for the β/t_f ratios and the memory sizes considered. Furthermore, for the optimal speedup, the k is such that $2bkd \leq n$ (the computational cost of SA2 always increases with k, so it can only decrease the optimal k predicted by the previous case).

[1] $A = \frac{1.5(d+1)^d(2b)^{d/(d+1)}(1+(2b+1)^d)}{d^{\frac{d^2}{d+1}}}$.

[2] $B = \frac{(d+1)^d}{d^d} - \frac{d}{d+1}$.

The above model makes some approximations which are good if the optimal k and n are large enough. Given this, the optimal speedup predicted is still close to (but lower than) the speedup in the more detailed performance model discussed in Section 5.2. Except for the Clovertown model ($\beta/t_f = 3.2$, $mem = 10^6$, has an optimal $k = 3$ which is small) in Section 5.2, the optimal speedups in the analytical model (the plots in Figure 13) match the speedups in the detailed model very well. As for the measured speedup in Section 6, the analytical performance model overestimates the optimal speedup. The main reason for this is that the read and write bandwidths differ significantly, which is not handled by the analytical model.

5 Detailed Performance Modeling

In this section we present detailed performance models of matrices with 2D and 2D stencil graphs for PA2 and SA2 using realistic machine parameters, in order to identify situations where significant speedups are likely. The two parallel machines for which we model PA2 are called Peta (which is a model of a nominal 8100 processor petascale machine) and Grid (which is a model of 125 terascale machines connected over the internet). We consider both overlapping (non-blocking) and non-overlapping (blocking) communication models; only the former can overlap communication and computation. The two sequential machines for which we model SA2 are OOC (which models an out-of-core implementation, where fast memory is DRAM and slow memory is disk) and Clovertown, the Intel multicore processor (where fast memory is cache and slow memory is DRAM). This variety of models of course suggests that our techniques can be applied more than once, if there are several levels of memory hierarchy and possibly also parallelism.

Specifically, we consider matrices whose graphs are 2D $(2b+1)^2$ point and 3D $(2b+1)^3$ point stencils. As before, we assume that quantities like $p^{1/2}$ and $\frac{n}{p^{1/3}}$ are integers.

5.1 Performance Modeling of PA2

We consider parallel machines with the following parameters:

p_{max}: The maximum number of processors available. The actual number of processors used is $p \leq p_{max}$. We may choose $p < p_{max}$ if that is faster, or if $p_{max}^{1/2}$ is not an integer, etc.

t_f: The time per floating-point operation (in units of seconds), modeled as 10% of machine peak value, a typical value attainable for SpMV.

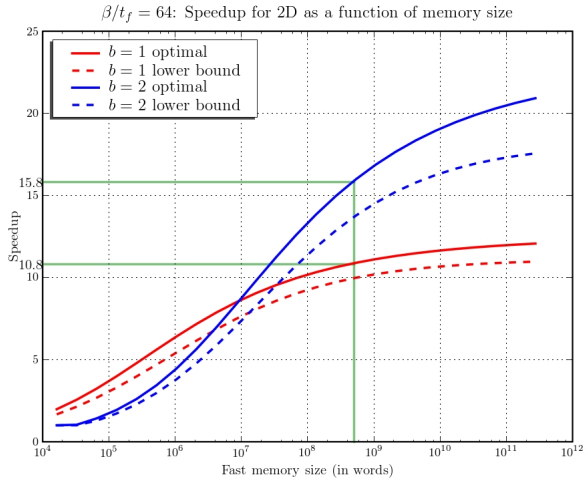
mem: The memory available per processor (in units of 8-byte words).

α: The network processor latency (in units of seconds).

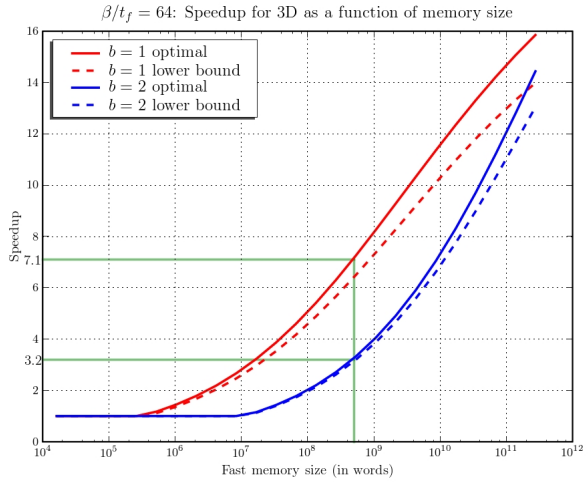
β: The inverse network bandwidth (in units of seconds/8-byte word).

Thus the time to send m words between any pair of processors is modeled as $\alpha + \beta m$ seconds. We modeled machines with the following parameter values:

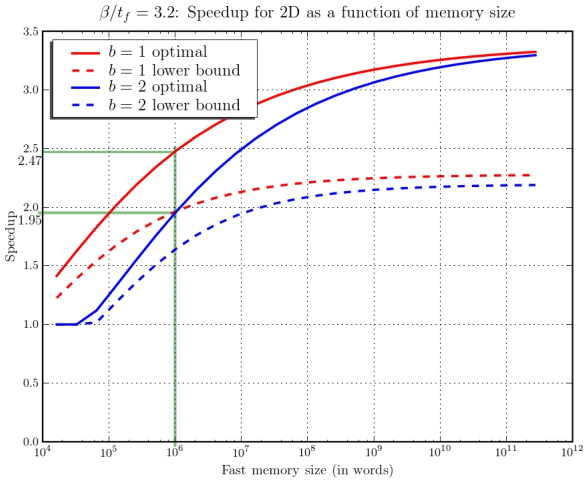
Peta: $p_{max} = 8100$, $t_f = 2 \cdot 10^{-11}$ secs ($1/t_f = 50$ GFlops/s), $mem = 62.5 \cdot 10^9$ words, $\alpha = 10^{-5}$ secs, $\beta = 2 \cdot 10^{-9}$ secs ($1/\beta = 500$ MWords/s = 4 GByte/s)



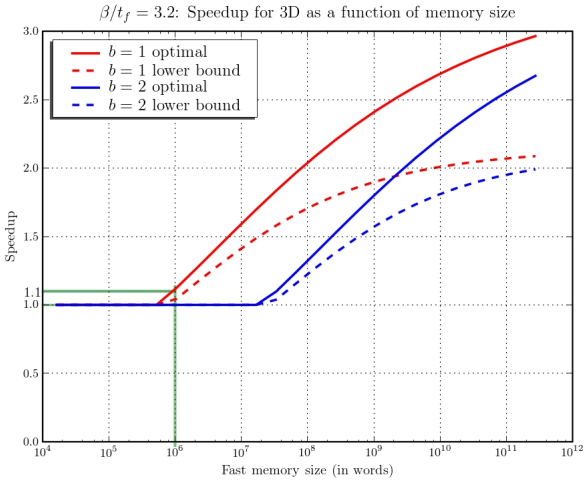
(a) Optimal speedups and lower bounds for 2D stencil with β/t_f ratio same as for OOC



(b) Optimal speedups and lower bounds for 3D stencil with β/t_f ratio same as for OOC



(c) Optimal speedups and lower bounds for 2D stencil with β/t_f ratio same as for Clovertown



(d) Optimal speedups and lower bounds for 3D stencil with β/t_f ratio same as for Clovertown

Figure 13: Optimal speedup (analytical model). The vertical green lines indicate the memory size for the architectures modeled in Section 5.2. Therefore, the speedups in Section 5.2 are expected to be close to the speedups at the point where the green line intersects the optimal speedup curve.

Grid: $p_{max} = 125$, $t_f = 10^{-12}$ secs ($1/t_f = 1TFlop/s$), $mem = 1.2 \cdot 10^{12}$ words, $\alpha = 10^{-1}$ secs, $\beta = 25 \cdot 10^{-9}$ secs ($1/\beta = 40$ MWords/s $= .32$ GBytes/s) (estimated by dividing the Teragrid backbone bandwidth of 40 GBytes/s by p_{max})

Note that each processor in Peta and Grid is assumed to be a significant parallel computer itself, but we are only modeling the parallelism between these processors, not within them. Again, one could potentially apply our techniques for each level of parallelism, but we have not modeled this here.

In section 2.4 we described the three computational phases of PA2: Phase I must be done before any communication can be initiated, Phase II can be fully overlapped with communication, and Phase III can only begin after communication is complete. This justifies the performance model for the case of overlapping communication below.

Let N_I, N_{II}, and N_{III} respectively denote the flop counts for Phases I, II, and III of PA2. Let N_w denote the total number of words sent by a processor. Let $T_{n,k,p}^{overlap}$ denote the time taken for PA2 when overlapping communication is used; in this case we assume all messages can be in-flight simultaneously while computation is occurring. Let $T_{n,k,p}^{nonoverlap}$ denote the time taken for PA2 when non-overlapping communication is used; in this case we assume only one message can be in flight at a time and not overlapped with computation. Our latency-avoiding algorithm may have more opportunity to demonstrate speedups in the non-overlapping case. In an actual machine the degree of overlap may lie somewhere between these two extremes. Let $M_{n,k,p}$ denote the memory required per processor when p processors are used. We let $T_{n,k,p}$ denote the time taken for the algorithm. So, if non-overlapping communication is used, then $T_{n,k,p} = T_{n,k,p}^{nonoverlap}$, else $T_{n,k,p} = T_{n,k,p}^{overlap}$.

We use the following formulas for these quantities (which are slightly more detailed than the entries in Table 1):

$$\begin{aligned}
N_I &= (8b^2 + 8b + 1) \cdot \left(\frac{n}{p^{1/2}} - bk \right) \cdot (bk^2 - 2bk) \\
N_{II} &= (8b^2 + 8b + 1) \cdot \left(\frac{3n^2}{p} - \frac{9bkn}{p^{1/2}} + 7b^2k^2 + 2b^2 \right) \cdot k/3 \\
N_{III} &= (8b^2 + 8b + 1) \cdot bk \cdot \left(\frac{9nk}{p^{1/2}} + \frac{6n}{p^{1/2}} - bk^2 - 6bk - 8b \right) /3 \\
N_w &= 2bk \cdot \left(\frac{2n}{p^{1/2}} + 3kb - 2b \right) \\
T_{n,k,p}^{overlap} &= (N_I + N_{III}) \cdot t_f + \max\left(N_{II} \cdot t_f, \alpha + \beta \cdot N_w\right) \\
T_{n,k,p}^{nonoverlap} &= (N_I + N_{II} + N_{III}) \cdot t_f + 8 \cdot \alpha + \beta \cdot N_w \\
M_{n,k,p} &= (k+1)\frac{n^2}{p} + 1.5(2b+1)^2 \left(\frac{n}{p^{1/2}} + bk \right)^2 + 1.5N_w
\end{aligned}$$

Note that the coefficient for the α term is 8 in $T_{n,k,p}^{nonoverlap}$ because each processor needs to communicate with 8 other processors. However, the coefficient for the α term is 1 in $T_{n,k,p}^{overlap}$ because all 8 sends to other processors can be overlapped.

Similarly, for the 3D stencils, we have the following formulas:

$$N_I = (2(2b+1)^3 - 1) \cdot (bk^2 - 2bk) \cdot \left(\frac{6n^2}{p^{2/3}} - \frac{12bkn}{p^{1/3}} + 7b^2k^2 - 2b^2k \right) /4$$

$$
\begin{aligned}
N_{II} &= (2(2b+1)^3-1)\cdot k\cdot\left(\frac{4n^3}{p}-\frac{18bkn^2}{p^{2/3}}+\frac{(28b^2k^2+8b^2)n}{p^{1/3}}+O(b^3k^3)\right)/4\\
N_{III} &= (2(2b+1)^3-1)\cdot bk\cdot\left(\frac{n^2}{p^{2/3}}(18k+12)-\frac{nb}{p^{1/3}}(4k^2+24k+32)+O(b^2k^3)\right)/4\\
N_w &= 2bk\cdot\left(\frac{3n^2}{p^{2/3}}+\frac{9bkn}{p^{1/3}}-\frac{6bn}{p^{1/3}}+O(b^2k^2)\right)\\
T_{n,k,p}^{overlap} &= (N_I+N_{III})\cdot t_f+\max\left(N_{II}\cdot t_f,\alpha+\beta\cdot N_w\right)\\
T_{n,k,p}^{nonoverlap} &= (N_I+N_{II}+N_{III})\cdot t_f+26\cdot\alpha+\beta\cdot N_w\\
M_{n,k,p} &= (k+1)\frac{n^3}{p}+1.5(2b+1)^3\left(\frac{n}{p^{1/3}}+bk\right)^3+1.5N_w
\end{aligned}
$$

For performance modeling, we also vary the parameter p within the range allowed to find the optimal value of p for specific n, k, b values. This range is limited by two parameters–p_{max} and mem. If p is made small, then the memory required per processor $M_{n,k,p}$ might exceed the memory available per processor mem. Furthermore, p can be at most p_{max}. Another limit we impose on p is the number of entries per dimension of the stencil should be be at least $2bk$ since our operation counts assume this condition. We may also round p down to the nearest perfect square or cube. The optimal p is strongly problem dependent, e.g., for small problem sizes, $p = 1$ might be sufficient and better since it avoids the overhead of communication. Therefore, a good measure of how well PA2 performs with respect to the conventional algorithm is the speedup with respect to the conventional algorithm assuming optimal p values were used for each algorithm:

$$speedup = \frac{\min_{1\le p\le p_{max}} T_{n,1,p}\cdot k}{\min_{1\le p\le p_{max}} T_{n,k,p}}$$

Note that we used $T_{n,1,p}\cdot k$ for the time taken for the conventional algorithm as the conventional algorithm turns out to be k invocations of PA2 with $k = 1$.

Another interesting metric for evaluating the algorithm is how well it can hide the communication cost. Specifically, we compare the time due to PA2 on a machine with the time if the same machine had zero communication overhead. If both the times are close, then our algorithm is latency and bandwidth insensitive, i.e., it can make the cost of communication disappear. So, we also plot this ratio in our plots. In addition, we also look at the additional floating-point operations performed by PA2 to hide latency.

We now discuss the performance modeling results for each combination of machine (Peta or Grid), communication style (overlapping or non-overlapping) and stencil (2D or 3D). There are 8 plots shown for each of these 8 combinations:

The first 4 plots of each group of 8 assume a bandwidth of $b = 1$, and respectively show for each combination of n and k (a) the best speedup attainable over all choices of $p \le p_{max}$, (b) the corresponding optimal choice of p, (c) the corresponding fraction of time spent in computation, and (d) the ratio of floating point operations done by the optimized algorithm to the number done by the conventional algorithm. The conventional algorithm corresponds to $k = 1$, the bottom row of each plot. Plots (c) and (d) show how successful our new algorithm is at reducing the fraction of time spent communicating, and the price paid in extra computation.

Now we describe the next 4 plots of each group of 8. For each combination of n and k, and for bandwidth $b = 1$, we show (a) the ratio of time taken by the new algorithm to the time that

would be taken on the same machine but with zero latency, and (b) the ratio of time taken by the new algorithm to the time that would be taken on the same machine but with zero latency and infinite bandwidth. We also show (c) for a fixed value of n, and each combination of k and bandwidth b, the best speedup attainable over all choices of $p \leq p_{max}$, and (d) the corresponding optimal choice of p. In plot (a) (or (b)) the time that would be taken with zero latency (or zero latency and infinite bandwidth, resp.) is measured for the same k but a possibly different optimal value of p. Plots (a) and (b) provide another metric of how well our algorithm does at reducing the cost of communication (the closer the ratios are to 1, the better).

5.1.1 2D Stencil on Peta Using Overlapping Communication

As can be seen in Figure 14(a), for smaller n and k (the bottom left corner), the speedup is close to linear in k, which is the best possible speedup. This is explained by Figure 14(c) which shows that almost all the time is spent in communication for these values of n and k.

The best speedup is 6.9x, which occurs when $n = 2^{11}$, $k = 12$, with an optimal $p = 7225 = 85^2$, a bit less than $p_{max} = 8100 = 90^2$; for these values of n, k and p the fraction of time spent in computation is 23% (versus 2% for the conventional $k = 1$ algorithm) and the number of floating point computations is 1.74x larger than the conventional algorithm (Figure 14(d)).

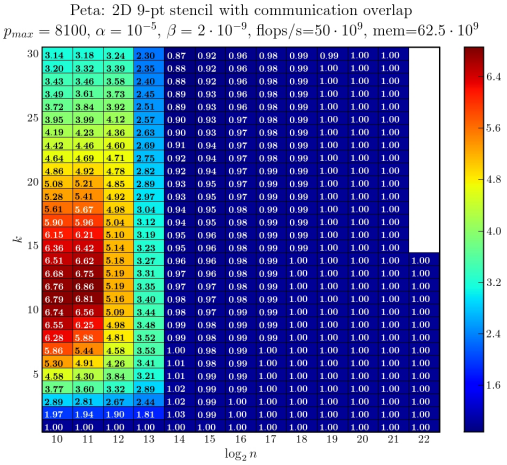
Indeed, for any problem size n, we can choose k to make the fraction of time spent doing arithmetic exceed 20% (up from under 1% for $k = 1$). And this never increases the number of floating point operations by more than 1.74x.

On the other hand, the algorithm has no benefit for large values of n, because computation totally dominates communication, as again shown by the bottom row of Figure 14(c); in this case no optimization is necessary either. We also note that for smaller n the speedup decreases as k is increased beyond a certain point, because the overhead of extra floating point operations exceeds the gains from reducing latency. The optimal p also decreases as k increases for some values of n because of the constraint $n/\sqrt{p} \geq 2bk$ we impose to guarantee that boundary regions only extend into the nearest neighboring processor.

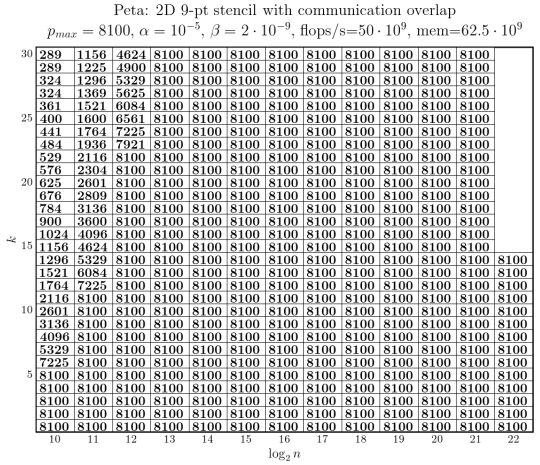
As can be seen in Figure 15(a), the latency has an enormous effect for small n and k, with the conventional algorithm ($k = 1$) running up to 89.39x slower than the 0-latency machine. But the algorithm can lower the latency to equal the floating point time even with small values of k (e.g., for $n = 2^{11}, k = 1$ we see that the conventional algorithm is 46.53x slower than the case if latency were 0, but raising k to 12 makes PA2 only 6.78x slower than the 0 latency case). For large n reducing latency to zero yields no speedups because computation dominates. Figure 15(b) tells a similar story as Figure 15(a), except that additionally increasing bandwidth to infinity would speed up the conventional code even more. In Figure 15(c) we see that for $n = 2^{12}$ the speedup due to PA2 decreases as b increases: this is because computation scales as b^2 which rapidly dominates communication.

5.1.2 2D Stencil on Peta Using Non-Overlapping Communication

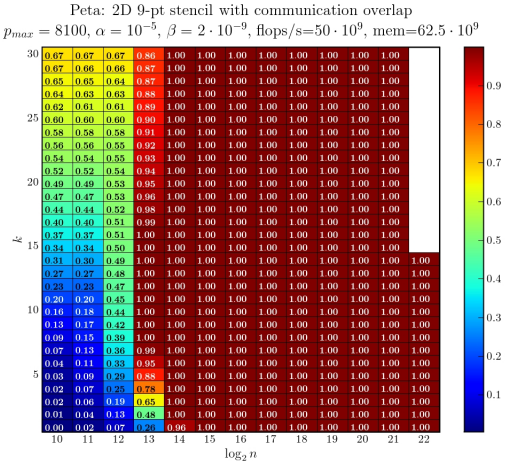
As can be seen in Figure 16(a), the algorithm is expected to obtain high speedups of up to 15.1x for smaller matrices. In fact, we get good speedups even for $n = 2^{14}$ in contrast to the case when overlapping communication was used. This is because non-overlapping communication has 8x higher latency than overlapping communication, making our latency-avoiding algorithm even more valuable. As in the overlapping case, for sufficiently large n computation dominates communication



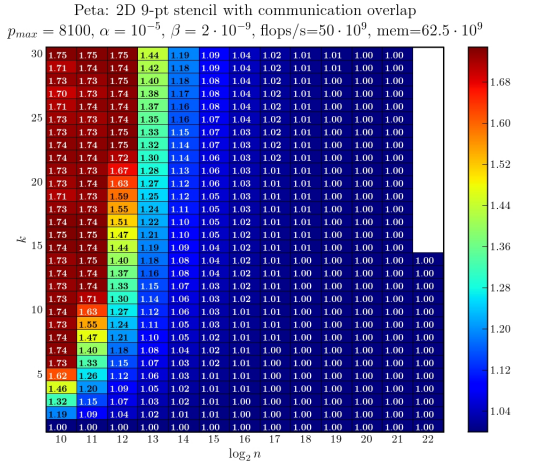
(a) Speedup



(b) Optimal p

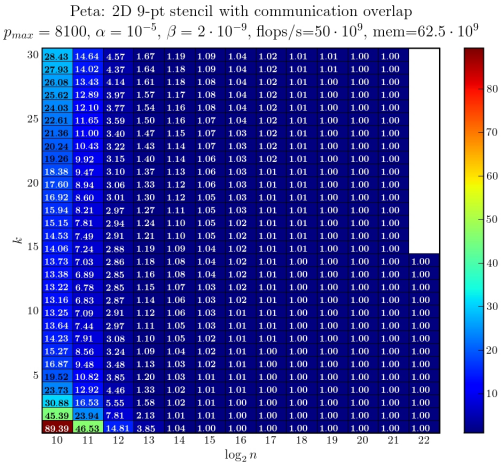


(c) Fraction of time in computation

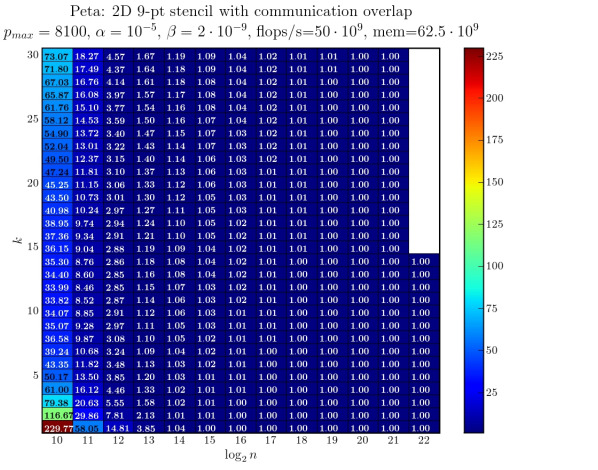


(d) Ratio of additional flops

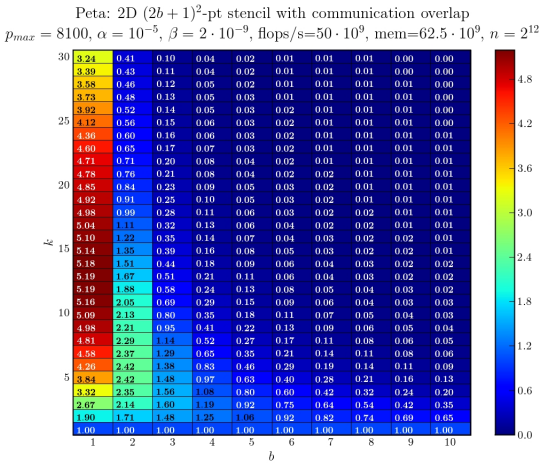
Figure 14: Plots for 2D stencil on Peta using overlapping communication.
Best speedup of 6.9x attained at $p = 7225 = 85^2, k = 12, n = 2^{11}$
For each n, the best k makes the fraction of time in computation $\geq 20\%$, up from $< 1\%$
For each n, the best k increases the number of flops by ≤ 1.74x



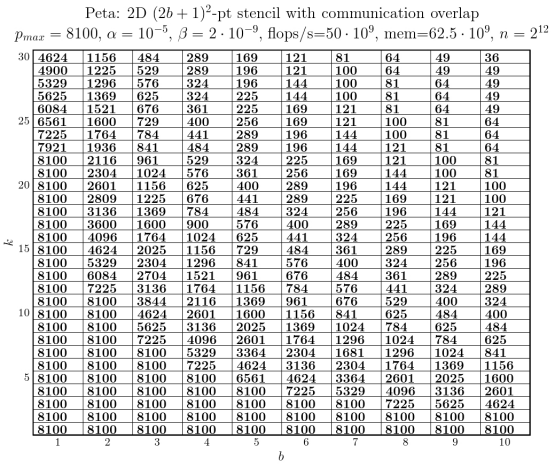
(a) Ratio of time w.r.t. 0 latency machine



(b) Ratio of time w.r.t. 0 latency, ∞ bandwidth machine



(c) Speedup as a function of matrix bandwidth ($n = 2^{12}$)



(d) Optimal p for (c)

Figure 15: Plots for 2D stencil on Peta using overlapping communication.
For each n, the best k makes the runtime ratio w.r.t. 0 latency ≤ 13.16, down from 89.39.
For each n, the best k makes the runtime ratio w.r.t. 0 latency/∞ BW ≤ 33.82, down from 229.77.

and there is no benefit from our algorithm. Figure 16(c) shows lower values when compared to Figure 14(c) because of the 8x larger latency. Figure 16(d) shows the same ratios of extra arithmetic as for the overlapping communication case, because the same optimal values of p are chosen (Figure 16(b)).

The comparisons to zero latency (Figure 17(a)) and zero latency/infinite bandwidth (Figure 17(b)) machines are even more extreme than in the overlapping case, because of the 8x higher latency assumed here. This also causes our algorithm to yield at least some speedup (1.28x) all the up to bandwidth $b = 10$ (Figure 17(c)).

5.1.3 2D Stencil on Grid Using Overlapping Communication

The white region in all the figures for $n = 2^{21}$ and $n = 2^{22}$ indicates that the problem needed too much memory to be solved by the machine.

As can be seen in Figure 18(a), the algorithm is expected to obtain an impressive speedup of up to 22.22x for large matrices ($n = 2^{17}$). Indeed, speedup is still increasing for the maximum value of k shown ($k = 30$), and larger k might show further improvements. The algorithm does not show any speedups for small values of n because the problem can be solved using only 1 processor and latency is too high to benefit from using more processors for $k \leq 30$. As before we can see that for very large problem sizes ($n \geq 2^{20}$), we also see no gains from our algorithm because computation dominates communication. In Figure 18(b) the optimal p takes on two values–either 1 or $p_{max} = 121$. Figure 18(c) shows the fraction of time in computation increasing from 2% at $k = 1$ to 53% at $k = 30$ for the value of $n = 2^{17}$ where speedup is best, but for smaller n and $k = 30$ the fraction of computation is still quite small; larger k might help. Figure 18(d) shows that in no case does the algorithm do more than 1.02x as many flops as the conventional algorithm.

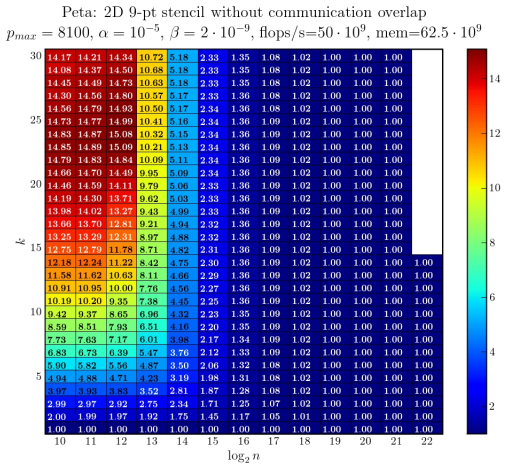
As can be seen in Figure 19(a), the ratio of time taken compared to the zero latency machine doughly doubles for each value of $\log_2 n$ from 10 to 16. The reason is that in this range, the optimal p for the Grid is 1, so all time is spent in computation, and for the 0 latency machine the optimal p is 121 and the largest part of the time is the bandwidth term. Thus doubling n quadruples the time on the Grid, but only doubles the time on the 0 latency machine, causing the ratio of these times to double. For larger n, computation begins to dominate both machines. It is at $n = 2^{16}$ and $n = 2^{17}$ that increasing k yields. the largest speedup. We note that many of the ratios in Figure 19(b) equal 121, because they correspond to cases where the optimal $p = 121$ for the 0 latency / infinite bandwidth machine, and $p = 1$ with nonzero latency and finite bandwidth.

Figure 19(c) shows that there is at least some speedup for $n = 2^{17}$ and all values of bandwidth b up to 10, although speedup decreases as b increases.

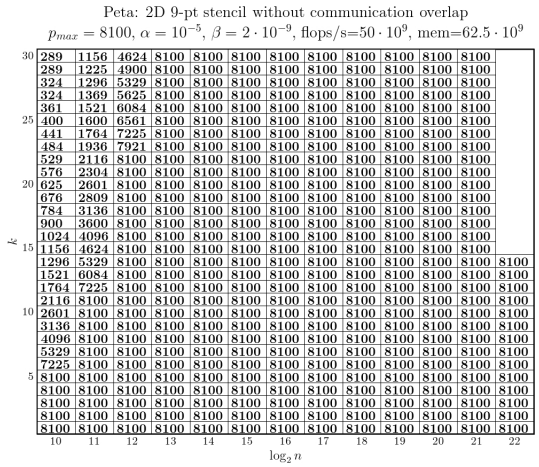
5.1.4 2D Stencil on Grid Using Non-overlapping Communication

As can be seen in Figure 20(a), the algorithm is expected to obtain speedups of up to 15.63x for large matrices, with speedup still increasing at the largest k shown. The speedups are sometimes larger and sometimes smaller than the overlapping case, depending on dimension. The extra expense of non-overlapping communication means that $p = 1$ is optimal for larger values of n than in the overlapping case (Figure 20(b)). The number of extra floating point operations never reaches 1% (Figure 20(d)).

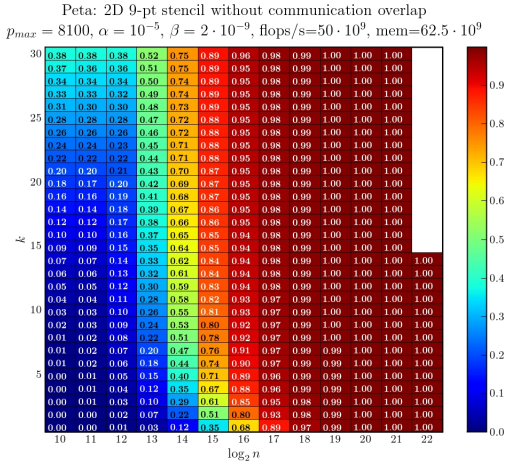
In Figure 21(b), many of the runtime ratios equal 121, because they correspond to cases where the optimal p was 1 for the actual Grid, whereas the optimal $p = 121$ for the 0 latency / infinite



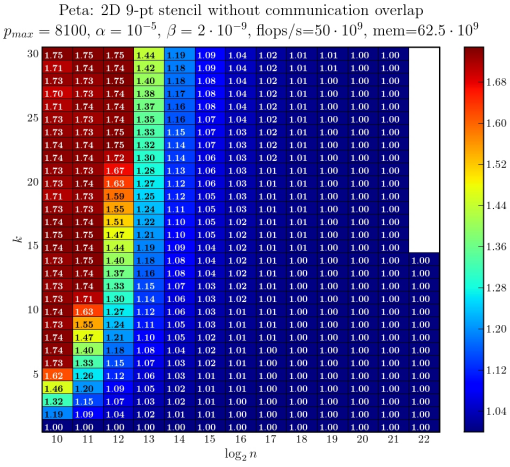
(a) Speedup



(b) Optimal p

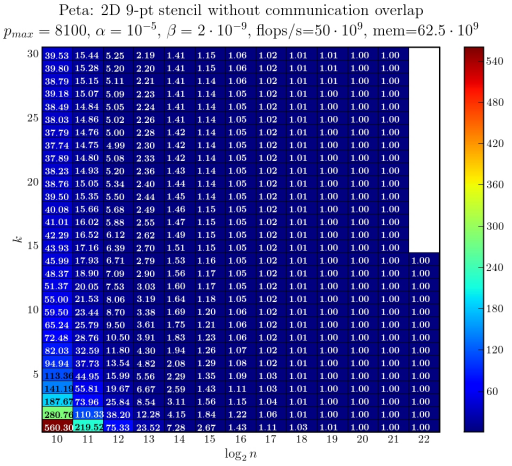


(c) Fraction of time in computation

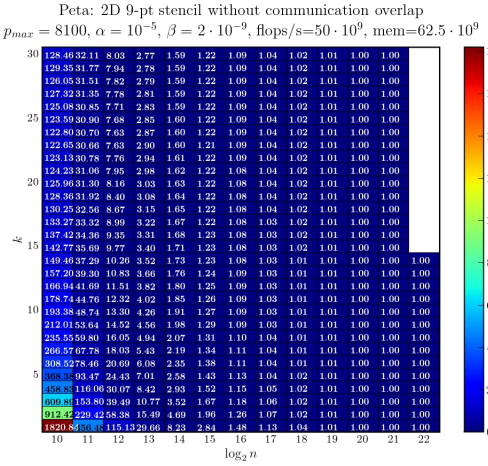


(d) Ratio of additional flops

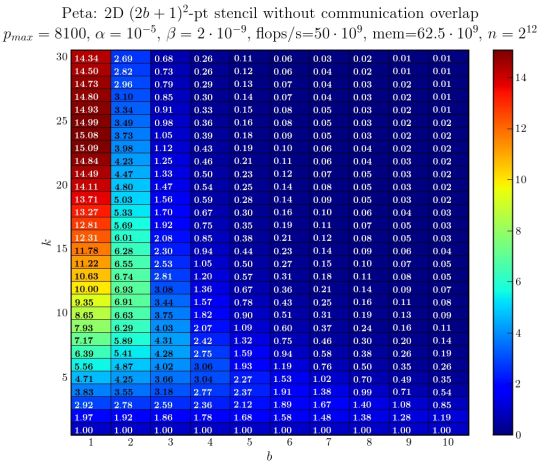
Figure 16: Plots for 2D stencil on Peta using non-overlapping communication.
Best speedup of 15.09x attained at $p = 7921 = 89^2, k = 23, n = 2^{12}$
For each n, the best k makes the fraction of time in computation $\geq 23\%$, up from $< 1\%$
For each n, the best k increases the number of flops by ≤ 1.75x



(a) Ratio of time w.r.t. 0 latency machine



(b) Ratio of time w.r.t. 0 latency, ∞ bandwidth machine



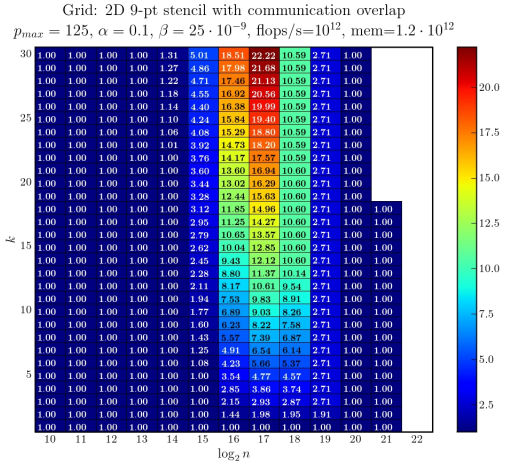
(c) Speedup as a function of matrix bandwidth ($n = 2^{12}$)

Peta: 2D $(2b+1)^2$-pt stencil without communication overlap
$p_{max} = 8100$, $\alpha = 10^{-5}$, $\beta = 2 \cdot 10^{-9}$, flops/s=$50 \cdot 10^9$, mem=$62.5 \cdot 10^9$, $n = 2^{12}$

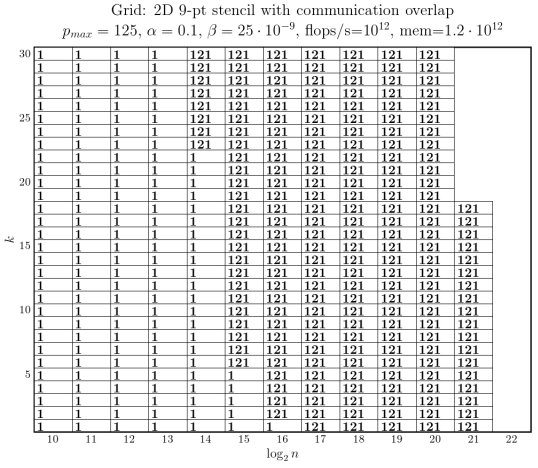
k \ b	1	2	3	4	5	6	7	8	9	10
30	4624	1156	484	289	169	121	81	64	49	36
	4900	1225	529	289	196	121	100	64	49	49
	5329	1296	576	324	196	144	100	81	64	49
	5625	1369	625	324	225	144	100	81	64	49
	6084	1521	676	361	225	169	121	81	64	49
25	6561	1600	729	400	256	169	121	100	81	64
	7225	1764	784	441	289	196	144	100	81	64
	7921	1936	841	484	289	196	144	121	81	64
	8100	2116	961	529	324	225	169	121	100	81
	8100	2304	1024	576	361	256	169	144	100	81
20	8100	2601	1156	625	400	289	196	144	121	100
	8100	2809	1225	676	441	289	225	169	121	100
	8100	3136	1369	784	484	324	256	196	144	121
	8100	3600	1600	900	576	400	289	225	169	144
	8100	4096	1764	1024	625	441	324	256	196	144
15	8100	4624	2025	1156	729	484	361	289	225	169
	8100	5329	2304	1296	841	576	400	324	256	196
	8100	6084	2704	1521	961	676	484	361	289	225
	8100	7225	3136	1764	1156	784	576	441	324	289
	8100	8100	3844	2116	1369	961	676	529	400	324
10	8100	8100	4624	2601	1600	1156	841	625	484	400
	8100	8100	5625	3136	2025	1369	1024	784	625	484
	8100	8100	7225	4096	2601	1764	1296	1024	784	625
	8100	8100	8100	5329	3364	2304	1681	1296	1024	841
	8100	8100	8100	7225	4624	3136	2304	1764	1369	1156
5	8100	8100	8100	8100	6561	4624	3364	2601	2025	1600
	8100	8100	8100	8100	8100	7225	5329	4096	3136	2601
	8100	8100	8100	8100	8100	8100	8100	7225	5625	4624
	8100	8100	8100	8100	8100	8100	8100	8100	8100	8100
	8100	8100	8100	8100	8100	8100	8100	8100	8100	8100

(d) Optimal p for (c)

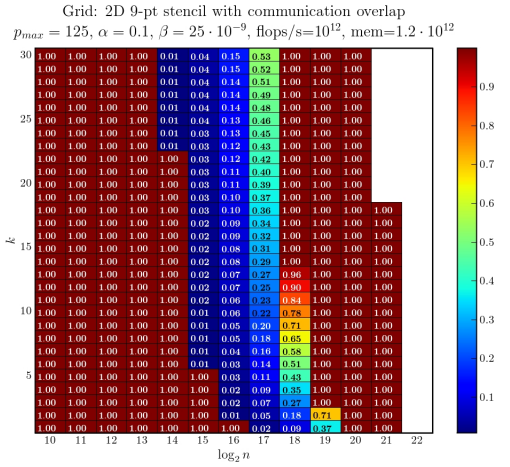
Figure 17: Plots for 2D stencil on Peta using non-overlapping communication.
For each n, the best k makes the runtime ratio w.r.t. 0 latency ≤ 37.71, down from 560.30.
For each n, the best k makes the runtime ratio w.r.t. 0 latency/∞ BW ≤ 122.65, down from 1829.84.



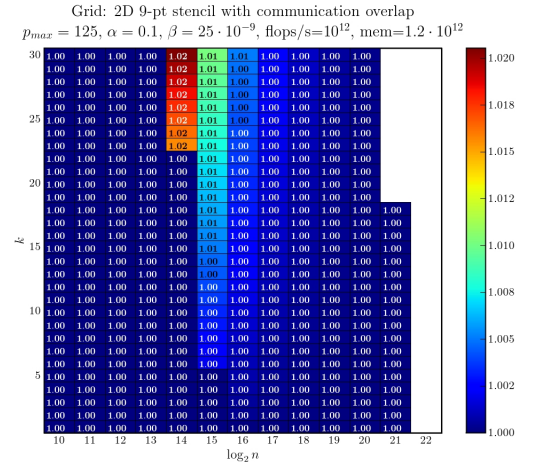
(a) Speedup



(b) Optimal p

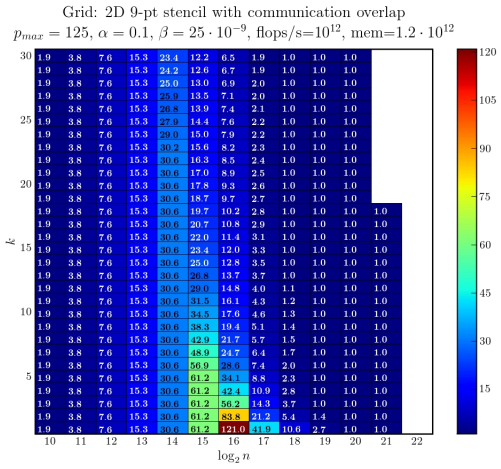


(c) Fraction of time in computation

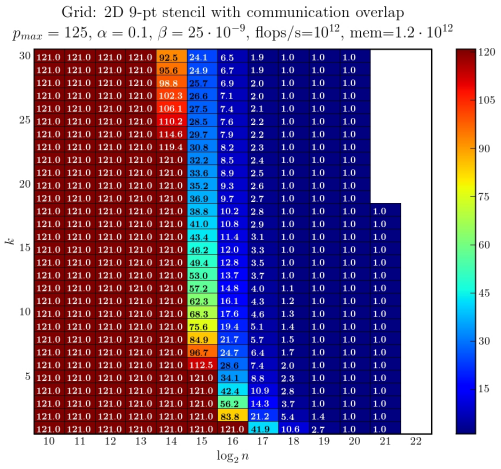


(d) Ratio of additional flops

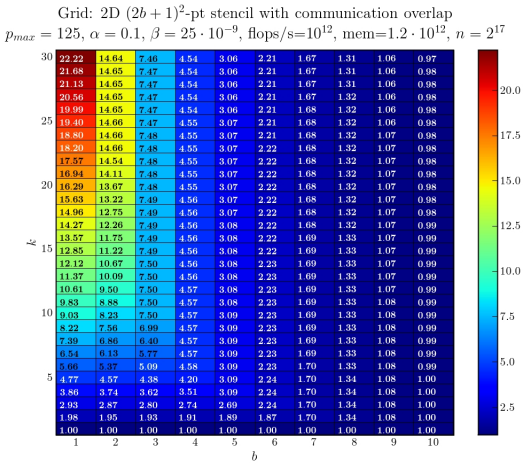
Figure 18: Plots for 2D stencil on Grid using overlapping communication.
Best speedup of 22.22x attained at $p = 121, k = 30, n = 2^{17}$.
For each n, the best k makes the fraction of time in computation $\geq 1\%$
For each n, the best k increases the number of flops by ≤ 1.02x



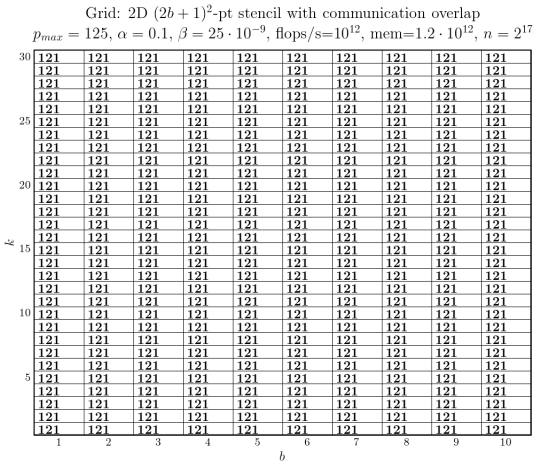
(a) Ratio of time w.r.t. 0 latency machine



(b) Ratio of time w.r.t. 0 latency, ∞ bandwidth machine



(c) Speedup as a function of matrix bandwidth ($n = 2^{17}$)



(d) Optimal p for (c)

Figure 19: Plots for 2D stencil on Grid using overlapping communication.
For each n, the best k makes the runtime ratio w.r.t. 0 latency ≤ 23.4, down from 121.

bandwidth Grid.

Figure 21(c) shows that there is speedup for all values of bandwidth b, but it is not a monotonic decreasing function of b, rather peaking (at least for $n = 2^{17}$) at 16.67x for $b = 3$ and $k = 30$.

5.1.5 3D Stencil on Peta Using Overlapping Communication

In contrast to the 2D case, in the 3D case no speedup is possible using our new algorithm (with the exception of a 2% speedup for $n = 2^9$ and $k = 2$). The reason is that the conventional $k = 1$ algorithm is already completely dominated by computation (Figure 22(c)), and indeed running nearly as fast as a zero latency machine (Figure 23(a)) or even a zero latency / infinite bandwidth machine (Figure 23(b)). Increasing the bandwidth b (Figure 23(c)) only makes it more computation dominated. We also note that the for larger b, the problem quickly becomes too large to be solved by the machine–this is evident by the large "whitespaces" in Figures 23(c) and 23(d).

5.1.6 3D Stencil on Peta Using Non-overlapping Communication

In contrast to the last case with overlapping communication, PA2 achieves a speedup of up to 3.58x, for $n = 2^9$ and $k = 8$, running only 3.04x slower than a zero latency machine (down from 10.89x) and only 4.50x slower than a zero latency / infinite bandwidth machine (down from 16.10x).

5.1.7 3D Stencil on Grid Using Overlapping Communication

In this case we get speedups of up to 4.41x for $n = 2^{10}$ and $k = 30$, doing only 1.29x as much arithmetic as the conventional algorithm, and running only 2.03x slower than a zero latency machine (down from 8.94x). However, it is 28.35x slower than a zero latency / infinite bandwidth machine, showing that bandwidth is the bottleneck. Some speedups are possible up to bandwidth $b = 5$.

5.1.8 3D Stencil on Grid Using Non-overlapping Communication

Not overlapping communication on the Grid yields a higher speedup of 7.79x for $n = 2^{12}$ and $k = 30$, running only 1.76x slower than a 0 latency machine (down from 13.73x), and only 7.87x slower than a 0 latency / infinite bandwidth machine (down from 61.26x). Speedups up to 7.04x are possible for higher bandwidth b.

5.2 Performance Modeling of SA2

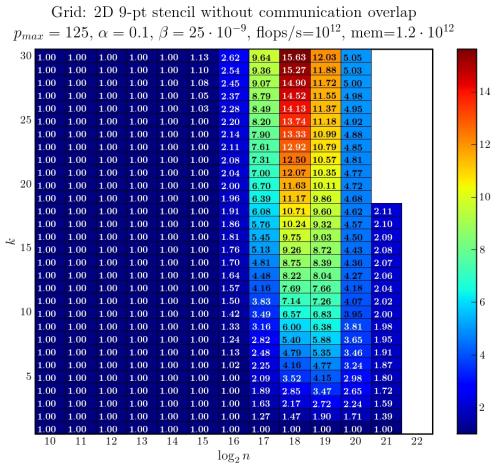
We consider uniprocessor machines with the following parameters:

p_{max}: The maximum number of number of blocks to be used. The actual number of blocks used is $p \leq p_{max}$. We may choose $p < p_{max}$ if that is faster, or if $p_{max}^{1/2}$ is not an integer, etc. In our simulations we choose p_{max} extremely large, so that the optimal p is sure to satisfy $p < p_{max}$.

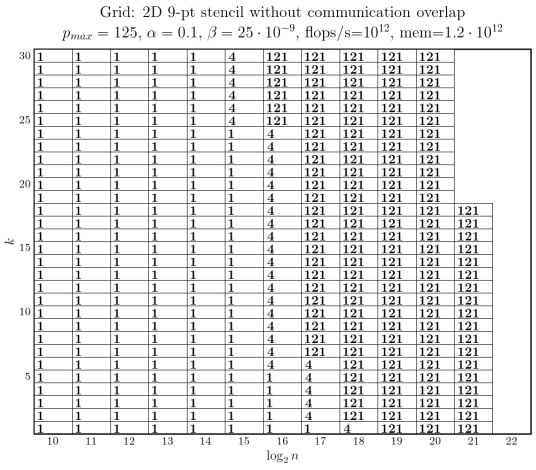
t_f: The time per floating-point operation (in units of seconds), modeled either as 10% of machine peak value (a typical value attainable for SpMV), or a median of measured values.

mem: The size of fast memory (in units of 8-byte words).

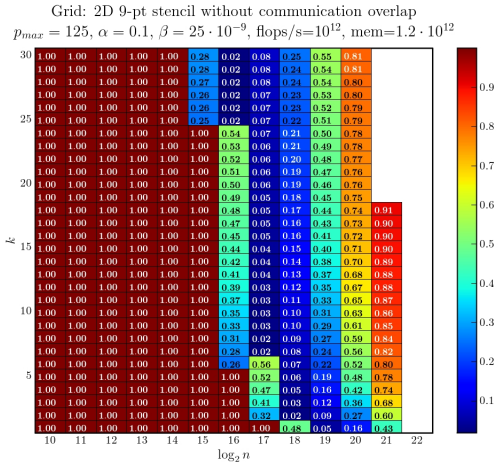
α: The slow memory latency (in units of seconds).



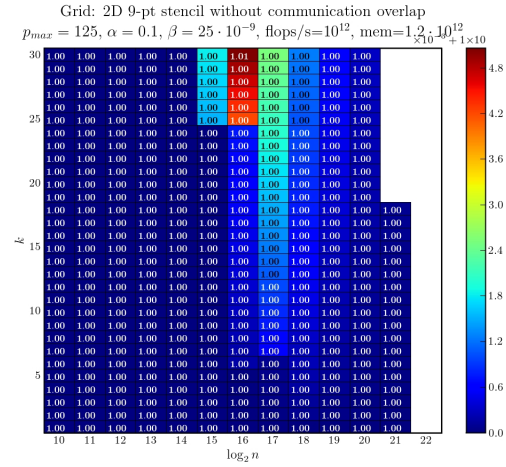
(a) Speedup



(b) Optimal p

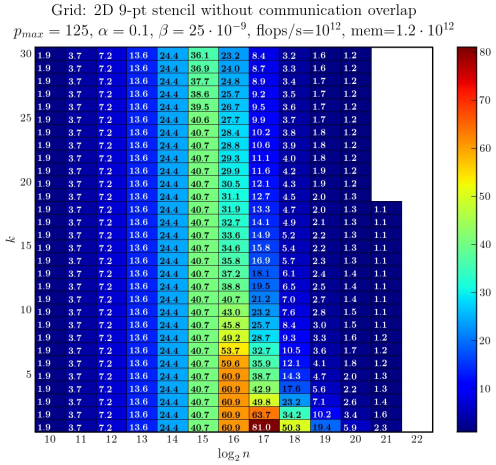


(c) Fraction of time in computation

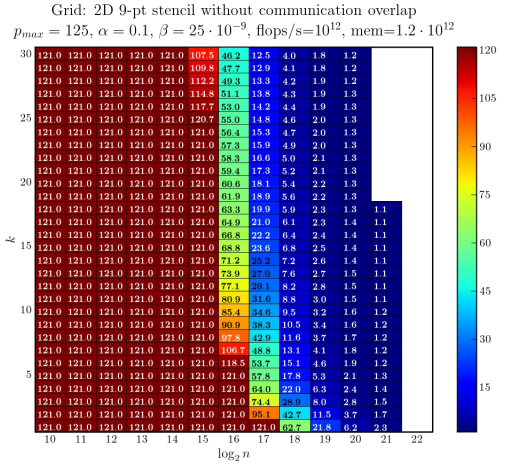


(d) Ratio of additional flops

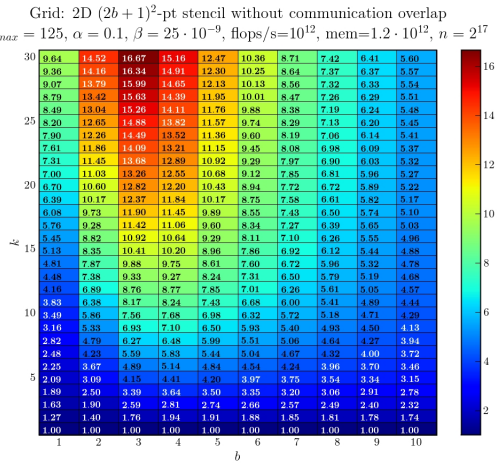
Figure 20: Plots for 2D stencil on Grid using non-overlapping communication. Best speedup of 15.63x attained at $p = 121, k = 30, n = 2^{18}$.
For each n, the best k makes the fraction of time in computation $\geq 2\%$
For each n, the best k increases the number of flops by ≤ 1.01x



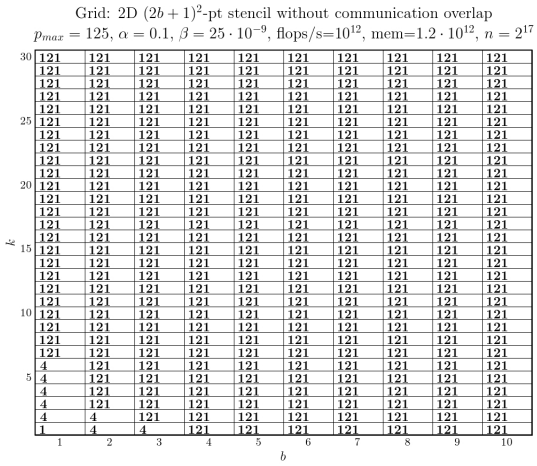
(a) Ratio of time w.r.t. 0 latency machine



(b) Ratio of time w.r.t. 0 latency, ∞ bandwidth machine

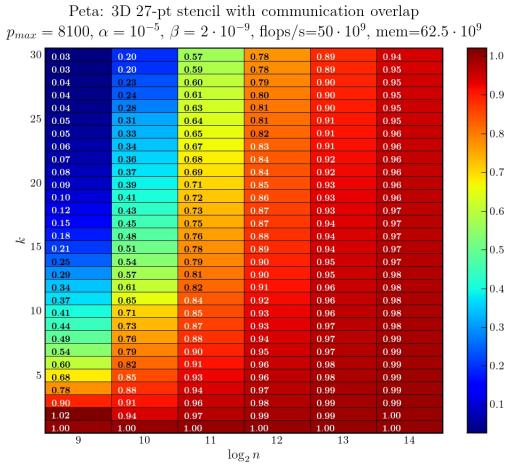


(c) Speedup as a function of matrix bandwidth ($n = 2^{17}$)

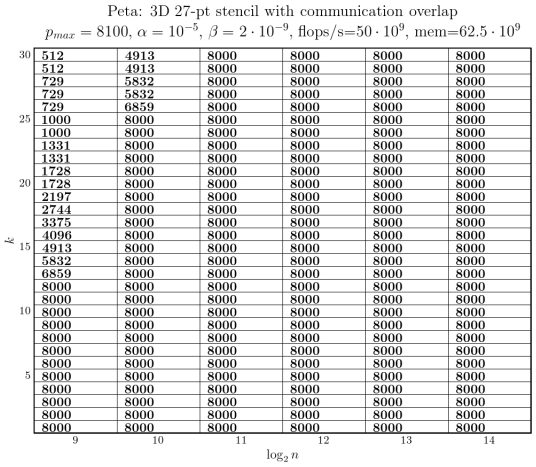


(d) Optimal p for (c)

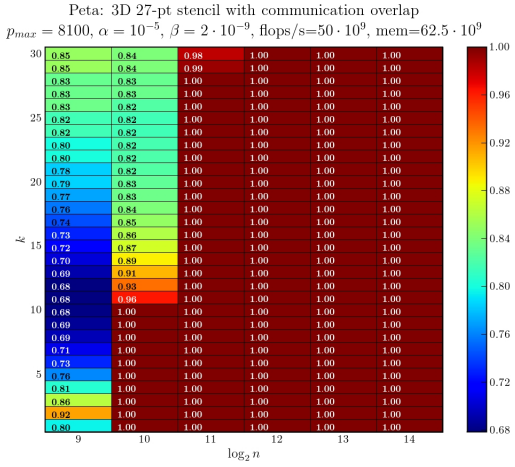
Figure 21: Plots for 2D stencil on Grid using non-overlapping communication.
For each n, the best k makes the runtime ratio w.r.t. 0 latency ≤ 36.1, down from 81.0.



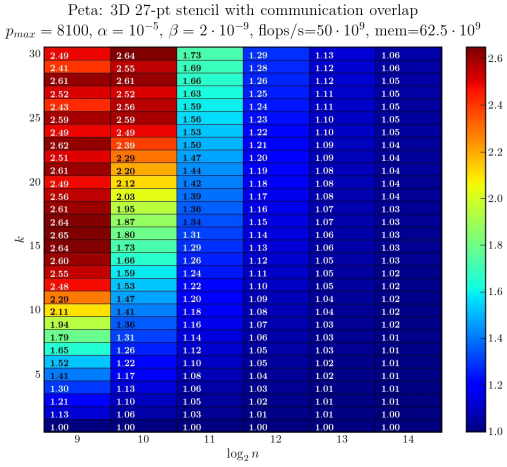
(a) Speedup



(b) Optimal p

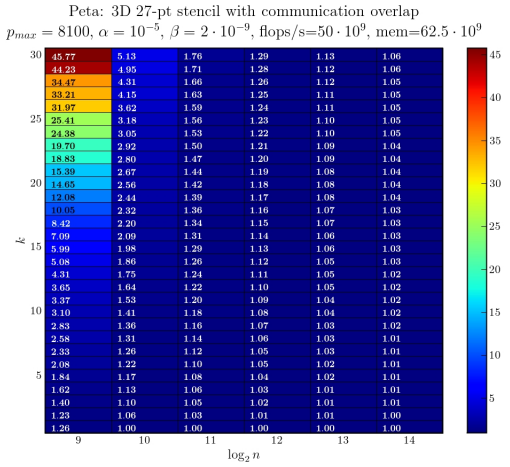


(c) Fraction of time in computation

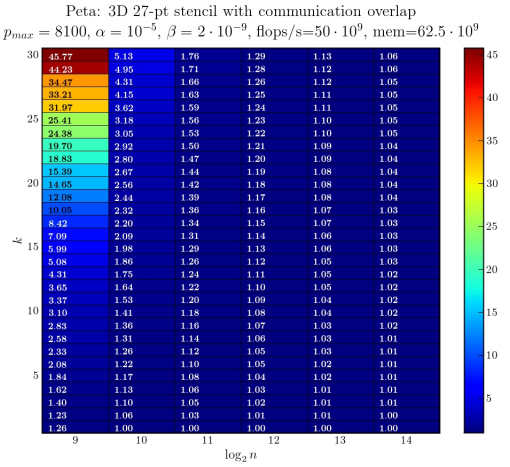


(d) Ratio of additional flops

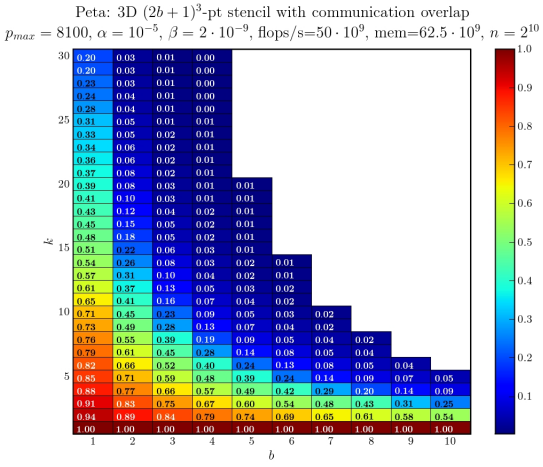
Figure 22: Plots for 3D stencil on Peta using overlapping communication.
Best speedup of 1.02x attained at $p = 8000, k = 2, n = 2^9$.
For each n, $k = 1$ makes the fraction of time in computation $\geq 80\%$



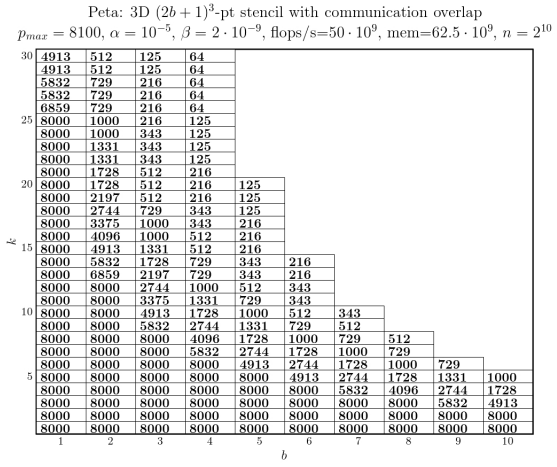
(a) Ratio of time w.r.t. 0 latency machine



(b) Ratio of time w.r.t. 0 latency, ∞ bandwidth machine

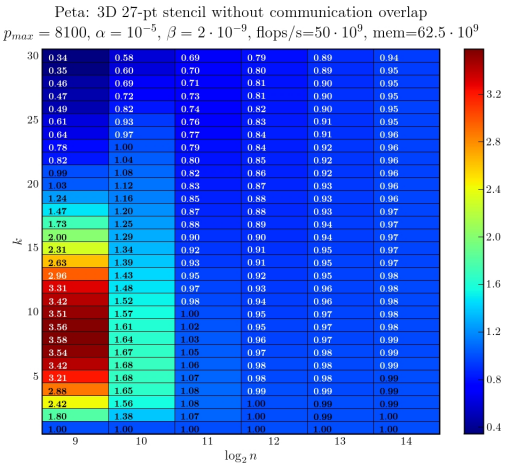


(c) Speedup as a function of matrix bandwidth ($n = 2^{10}$)

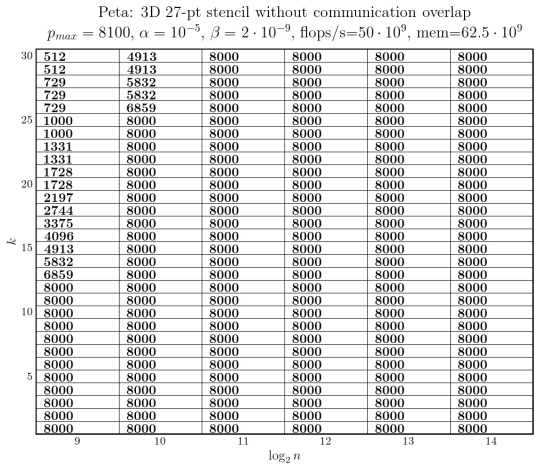


(d) Optimal p for (c)

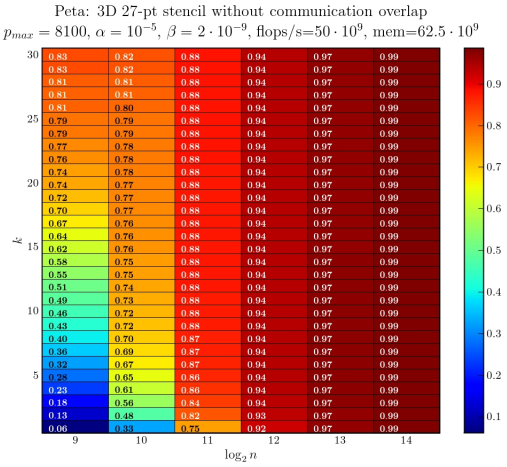
Figure 23: Plots for 3D stencil on Peta using overlapping communication.
For each n, $k = 1$ makes the runtime ratio w.r.t. 0 latency ≤ 1.26.
For each n, $k = 1$ makes the runtime ratio w.r.t. 0 latency/∞ BW ≤ 1.26.



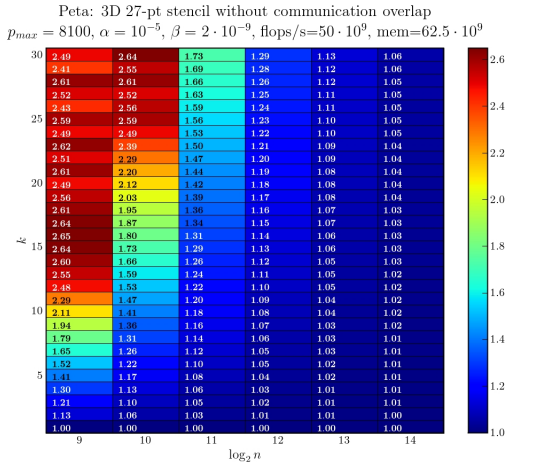
(a) Speedup



(b) Optimal p

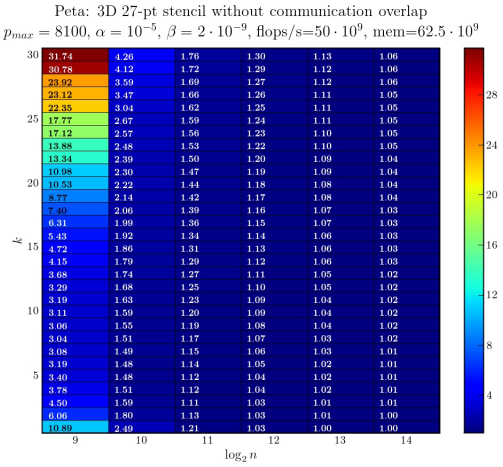


(c) Fraction of time in computation

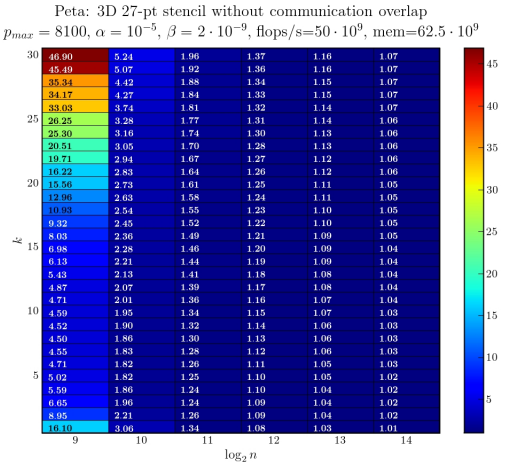


(d) Ratio of additional flops

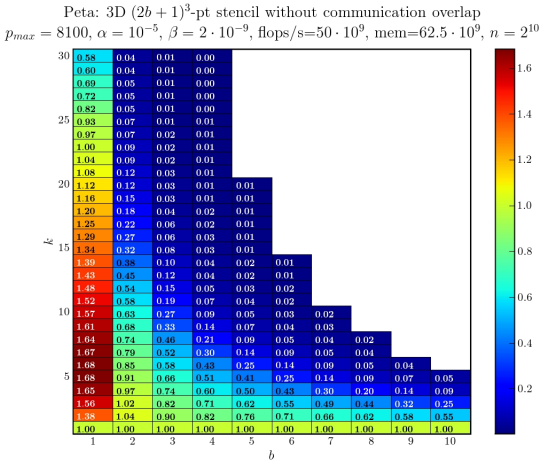
Figure 24: Plots for 3D stencil on Peta using non-overlapping communication.
Best speedup of 3.56 attained at $p = 8000, k = 8, n = 2^9$.
For each n, the best k makes the fraction of time in computation $\geq 40\%$, up from 6%.
For each n, the best k increases the number of flops by ≤ 1.79x



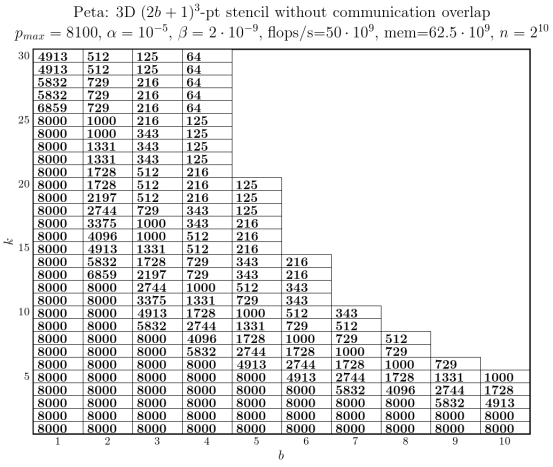
(a) Ratio of time w.r.t. 0 latency machine



(b) Ratio of time w.r.t. 0 latency, ∞ bandwidth machine

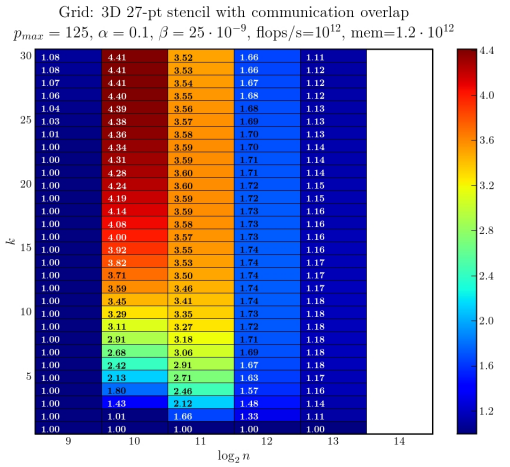


(c) Speedup as a function of matrix bandwidth ($n = 2^{10}$)

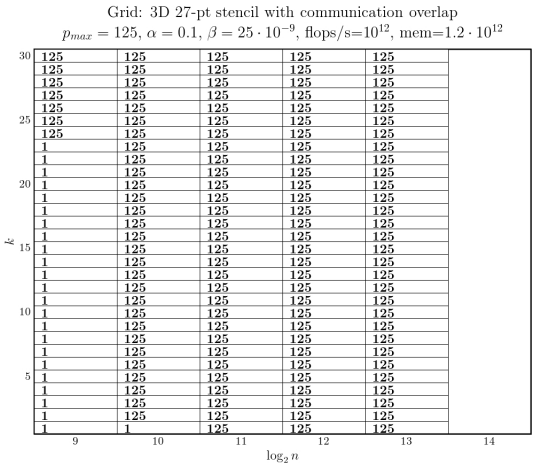


(d) Optimal p for (c)

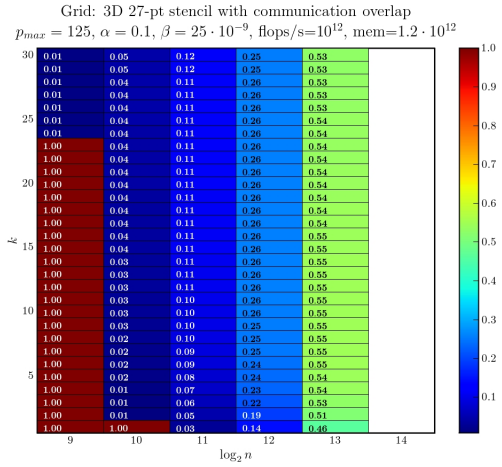
Figure 25: Plots for 3D stencil on Peta using non-overlapping communication.
For each n, the best k makes the runtime ratio w.r.t. 0 latency ≤ 3.04, down from 10.89.
For each n, the best k makes the runtime ratio w.r.t. 0 latency/∞ BW ≤ 4.50, down from 16.10.



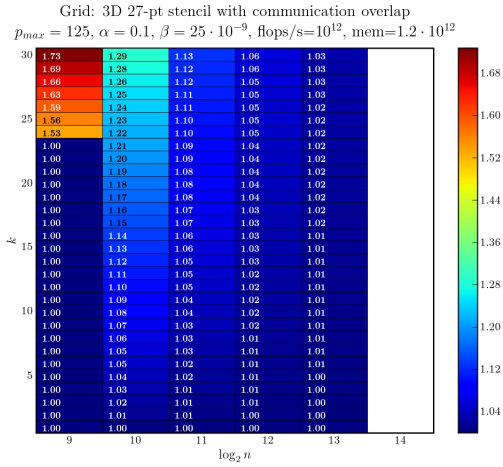
(a) Speedup



(b) Optimal p

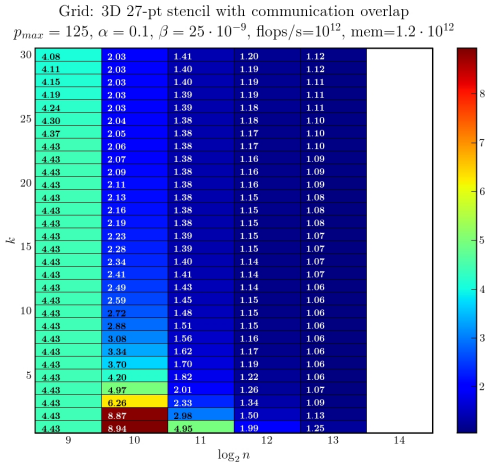


(c) Fraction of time in computation

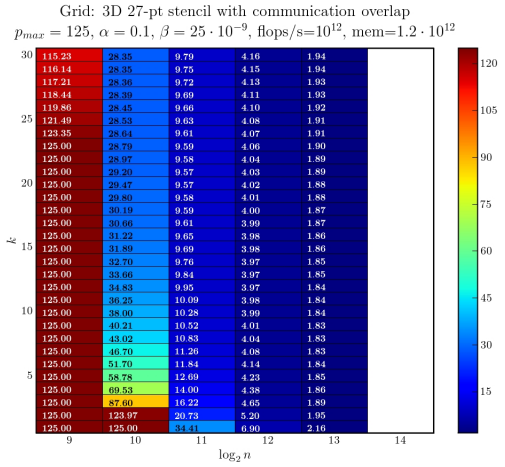


(d) Ratio of additional flops

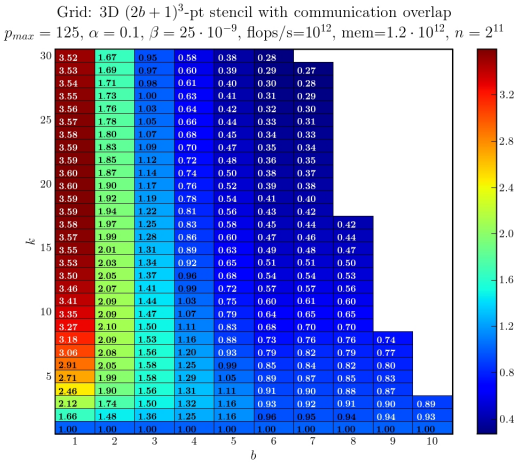
Figure 26: Plots for 3D stencil on Grid using overlapping communication. Best speedup of 4.41x attained at $p = 125, k = 30, n = 2^{10}$.
For each n, the best k makes the fraction of time in computation $\geq 1\%$
For each n, the best k increases the number of flops by ≤ 1.73x



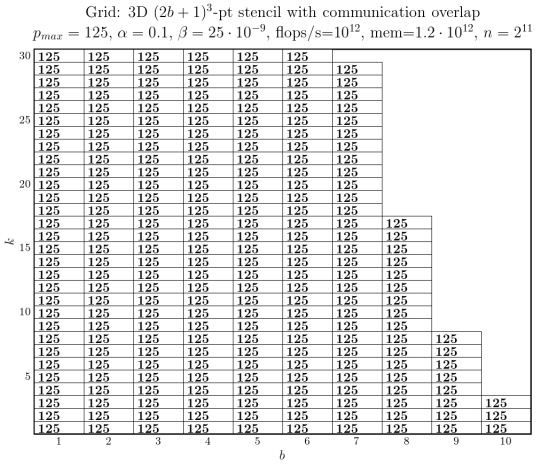
(a) Ratio of time w.r.t. 0 latency machine



(b) Ratio of time w.r.t. 0 latency, ∞ bandwidth machine

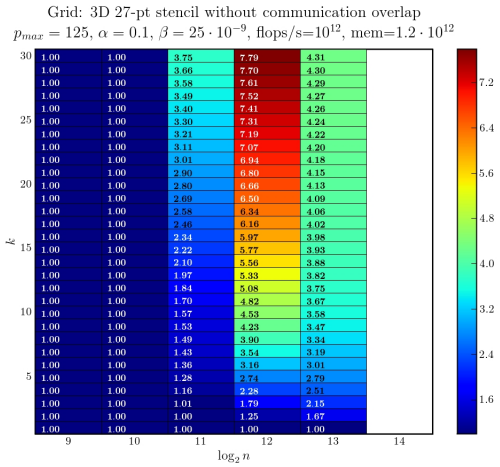


(c) Speedup as a function of matrix bandwidth ($n = 2^{11}$)

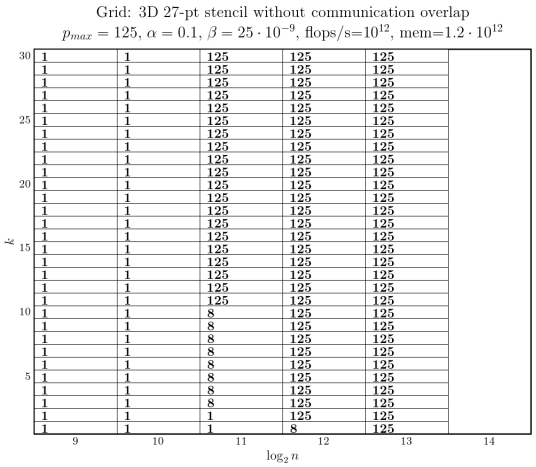


(d) Optimal p for (c)

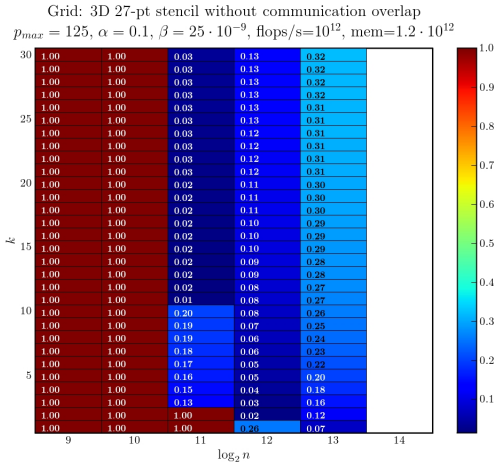
Figure 27: Plots for 3D stencil on Grid using overlapping communication.
For each n, the best k makes the runtime ratio w.r.t. 0 latency ≤ 4.08, down from 8.94.
For each n, the best k makes the runtime ratio w.r.t. 0 latency/∞ BW ≤ 115.23, down from 125.



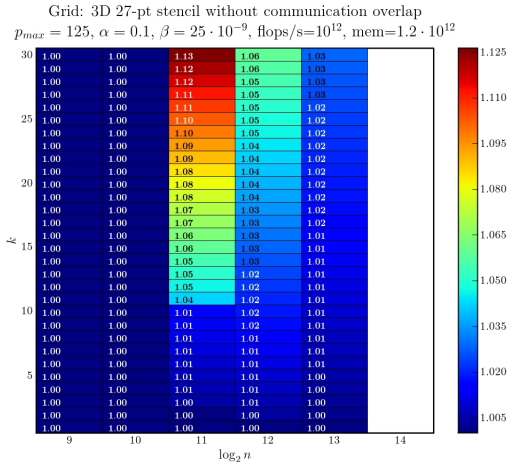
(a) Speedup



(b) Optimal p

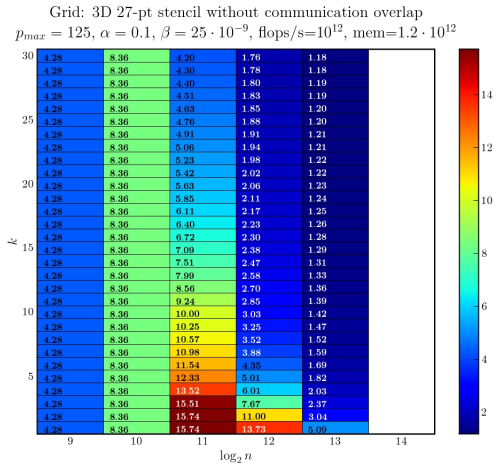


(c) Fraction of time in computation

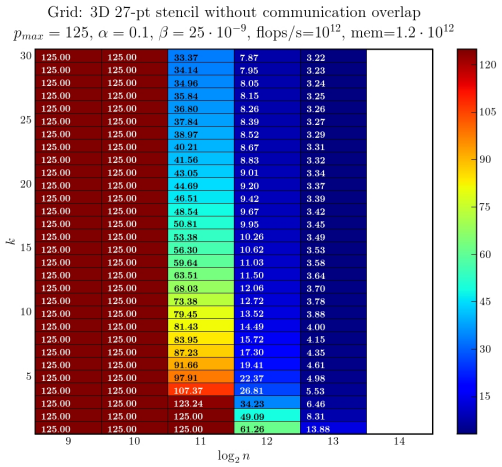


(d) Ratio of additional flops

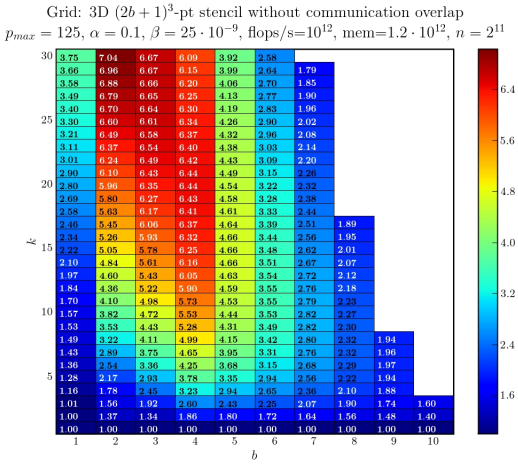
Figure 28: Plots for 3D stencil on Grid using non-overlapping communication.
Best speedup of 7.79x attained at $p = 125, k = 30, n = 2^{12}$.
For each n, the best k makes the fraction of time in computation $\geq 3\%$
For each n, the best k increases the number of flops by ≤ 1.13x



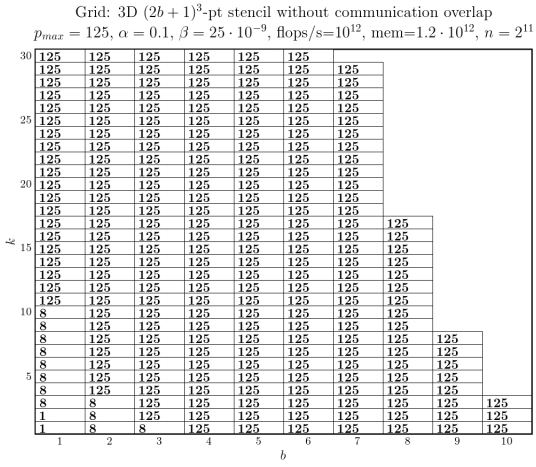
(a) Ratio of time w.r.t. 0 latency machine



(b) Ratio of time w.r.t. 0 latency, ∞ bandwidth machine



(c) Speedup as a function of matrix bandwidth ($n = 2^{11}$)



(d) Optimal p for (c)

Figure 29: Plots for 3D stencil on Grid using non-overlapping communication.
For each n, the best k makes the runtime ratio w.r.t. 0 latency ≤ 8.36, down from 15.74.

β: The inverse bandwidth available between slow and fast memory (in units of seconds/8-byte word).

Specifically, we model two machines with the following parameters:

OOC: This out-of-core implementation models a 500 MFlop/s uniprocessor with DRAM as fast memory and a 15000 RPM Seagate ST373307 disk as slow memory, with $p_{max} = 10^7$, $t_f = 2 \cdot 10^{-9}$ secs (500 MFlops/s), $mem = 5 \cdot 10^8$ words, $\alpha = 5.7 \cdot 10^{-3}$ secs, $\beta = 1.28 \cdot 10^{-7}$ secs ($1/\beta = 7.8$ MWords/s $= 62.5$ MB/s).

Clovertown: This multicore implementation models a quad-core Intel Clovertown chip with on-chip cache as fast memory and DRAM as slow memory, with $p_{max} = 10^7$, $t_f = 5 \cdot 10^{-10}$ secs (2 GFlops/s) (based on measurements in [34]), $mem = 10^6$ words, $\alpha = 2 \cdot 10^{-7}$ secs, $\beta = 1.6 \cdot 10^{-9}$ secs ($1/\beta = 625$ MWords/s $= 5$ GB/s).

In our performance models below, we assume the entire matrix and vector x are initially stored in slow memory, and that $[Ax, A^2x, ..., A^kx]$ is eventually stored in slow memory at the end of the computation. Also, we only model non-overlapping communication and computation.

Let N denote the number of floating-point operations performed by SA2 (we sometimes write $N_{b,n,k,p}$ to indicate functional dependencies). Let N_a denote the number of slow memory accesses. Let N_w denote the number of words transferred between fast and slow memory. Let $T_{b,n,k,p,\alpha,\beta}$ denote the time taken by SA2. Let $M_{b,n,k,p}$ denote the main memory required. Formulas for these are given below.

Given the machine and problem parameters for 2D stencil matrices, we state the following formulas (which are slightly more detailed than the formulas in Table 2):

$$
\begin{aligned}
N_{b,n,k,p} &= (8b^2 + 8b + 1) \cdot k \left(3n^2 + 6b(k-1)(p^{1/2} - 1)n + 2b^2(2k-1)(k-1)(p^{1/2} - 1)^2 \right) /3 \\
N_a &= 11p \\
N_w &= (k+1)n^2 + 1.5(2b+1)^2 \left(n + 2b(k-1)(p^{1/2} - 1) \right)^2 + 6bk(p^{1/2} - 1)(n + bk(p^{1/2} - 1)) \\
M_{b,n,k,p} &= 1.5 \left(\left(\frac{n}{p^{1/2}} + 2bk \right)^2 - \frac{n^2}{p} \right) + (k+1)\frac{n^2}{p} + 1.5(2b+1)^2 \left(\frac{n}{p^{1/2}} + 2b(k-1) \right)^2 \\
T_{b,n,k,p,\alpha,\beta} &= N \cdot t_f + N_a \cdot \alpha + \beta \cdot N_w
\end{aligned}
$$

Similarly, we state the following formulas for 3D stencil (which are also slightly more detailed than in Table 2):

$$
\begin{aligned}
N_{b,n,k,p} &= (2(2b+1)^3 - 1) \cdot k \left(n^3 + 3b(k-1)(p^{1/3} - 1)n^2 + O(b^2k^2np^{2/3}) \right) \\
N_a &= 44p \\
N_w &= (k+1)n^3 + 1.5(2b+1)^3 \left(n + 2b(k-1)(p^{1/3} - 1) \right)^3 + 1.5 \left(\left(n + 2bk(p^{1/3} - 1) \right)^3 - n^3 \right) \\
M_{b,n,k,p} &= 1.5 \left(\left(\frac{n}{p^{1/3}} + 2bk \right)^3 - \frac{n^3}{p} \right) + (k+1)\frac{n^3}{p} + 1.5(2b+1)^3 \left(\frac{n}{p^{1/3}} + 2b(k-1) \right)^3 \\
T_{b,n,k,p,\alpha,\beta} &= N \cdot t_f + N_a \cdot \alpha + \beta \cdot N_w
\end{aligned}
$$

Given b, n, k, α and β, we want to choose p to minimize the run time $T_{b,n,k,p,\alpha,\beta}$, subject to the memory constraint $M_{b,n,k,p} < mem$; write this optimal runtime as

$$T^{SA2,opt}_{b,n,k,\alpha,\beta,mem} = \min_{p:M_{b,n,k,p}<mem} T_{b,n,k,p,\alpha,\beta}$$

with the p achieving the minimum written $p^{opt}_{b,n,k,\alpha,\beta,mem}$, and the corresponding number of arithmetic operations written $N^{SA2,opt}_{b,n,k,\alpha,\beta,mem} = N_{b,n,k,p^{opt}_{b,n,k,\alpha,\beta,mem}}$.

We present performance modeling data of SA2 for each combination of machine (OOC and Clovertown) and matrix (2D and 3D). We present 6 plots for each of these 4 combinations. These plots are slightly different from the ones shown for PA2, since we want to evaluate the savings in both latency and bandwidth costs. The first 5 plots are all for stencil bandwidth $b = 1$, and the last plot is for other bandwidths $b > 1$. Note that along vertical axis in each plot data may only be shown for odd values of k.

1. The speedup $k \cdot T^{SA2,opt}_{1,n,1,\alpha,\beta,mem}/T^{SA2,opt}_{1,n,k,\alpha,\beta,mem}$ of the new algorithm versus the conventional algorithm run k times (with $b = 1$). The conventional algorithm corresponds to $k = 1$, and so reads the matrix and a vector from slow to fast memory (at least) k times, and writes a vector from fast to slow memory (at least) k times.

2. The minimizing $p^{opt}_{1,n,k,\alpha,\beta,mem}$ for the new algorithm (with $b = 1$), in the denominator in the fraction in the bullet (1).

3. The fraction of time spent by the new algorithm in arithmetic (with $b = 1$); when this ratio is close to 1, it tells us that we are running close to the peak floating point speed.

4. The ratio of floating point operations done by the new algorithm to the number done by the conventional algorithm (with $b = 1$): $\frac{N^{SA2,opt}_{1,n,k,\alpha,\beta,mem}}{k \cdot N^{SA2,opt}_{1,n,1,\alpha,\beta,mem}}$. Note that the optimizing p is chosen independently for new algorithm and the conventional algorithm; this is true in later formulas as well. This ratio tells us how much redundant work is done by the new algorithm in order to achieve the best possible speedup.

5. The ratio $k \cdot T^{SA2,opt}_{1,n,1,0,0,mem}/T^{SA2,opt}_{1,n,k,\alpha,\beta,mem}$ of the time of the conventional algorithm run on a machine with zero latency and infinite bandwidth (to "slow" memory) to the time of the new algorithm. The time on a zero latency / infinite bandwidth machine is a lower bound on what the new algorithm can achieve, so this ratio tells us well our new algorithm has succeeded in avoiding most cost of accessing slow memory (the ratio is less than 1, and the closer it is to 1, the better). It can also be interpreted as the fraction of peak performance attained by the new algorithm.

6. The speedup $k \cdot T^{SA2,opt}_{b,n,1,\alpha,\beta,mem}/T^{SA2,opt}_{b,n,k,\alpha,\beta,mem}$ of the new versus conventional algorithm for a fixed n and varying k and stencil size b.

We now present the performance modeling results for SA2.

5.2.1 2D Stencil on OOC

Figure 30(a) shows that for every value of n modeled, a maximum speedup of 10.2 is attained by choosing $k = 59$. Indeed, the speedup is still increasing slowly at $k = 59$, the largest value of k modeled, and so further speedups may be possible. Figure 30(c) shows that for each n, increasing k from 1 to 59 raised the fraction of time spent in computation from 2% to 18%. Since Figure 30(d) shows that this is accomplished without ever increasing the number of flops by more than 1.05x, we know further speedup would be limited to $1.05/.18 \approx 5.8x$. This same potential further speedup (actually the reciprocal, .17) is also expressed in Figure 30(e), which shows the time of the conventional algorithm running on a 0 latency / infinite bandwidth machine divided by the new algorithm's running time. Finally, Figure 30(f) shows that even higher speedups are possible for larger matrix bandwidths b, up to 13.8x speedup for $b = 3$.

We note that our new sequential algorithm provides speedups that are more uniform across values of n and b than our new parallel algorithm. The reason is that both the number of arithmetic operations and the number of words transferred grow proportionally to b^2n^2, making it always advantageous to avoid bandwidth costs.

5.2.2 3D Stencil on OOC

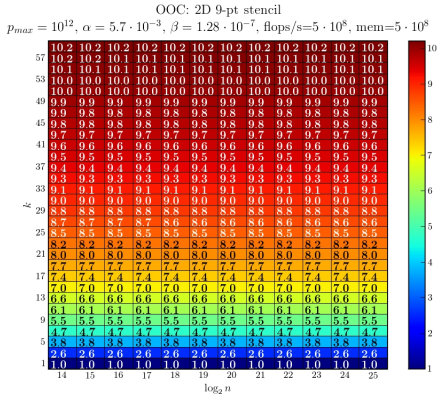
The 3D case is broadly similar to the 2D case. Figure 31(a) shows that for every value of n modeled, a maximum speedup in the range $[7.39, 9.51]$ is attainable, where the best k depends on n and varies in the range [23,43]. Figure 31(c) shows that for each n, choosing the best k raised the fraction of time spent in computation from 2% to at least 21%. Figure 31(d) shows that this is accomplished without ever increasing the number of flops by more than 1.57x. This potential fraction of peak is shown in Figure 31(e), which lies in the range [14%,18%], up from 2%. Figure 30(f) shows that good speedups are possible for larger matrix bandwidths b, but not as high as in the 2D case.

5.2.3 2D Stencil on Clovertown

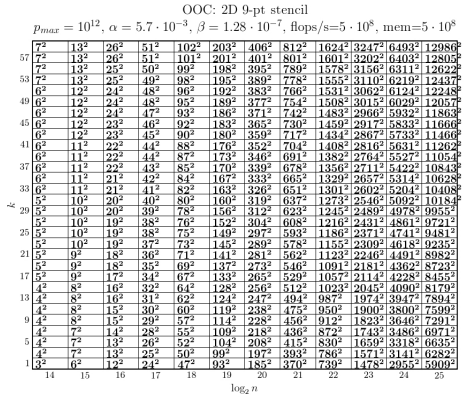
Figure 32(a) shows that for every value of n modeled, a maximum speedup in the range $[2.45, 2.58]$ is attainable. Figure 32(c) shows that for each n, choosing the best k raised the fraction of time spent in computation from 25% up to at least 71%. Figure 32(d) shows that this is accomplished without ever increasing the number of flops by more than 1.14x. The fraction of peak is expressed in Figure 32(e), and is in the range [.62,.65] when choosing the best value of k. Finally, Figure 32(f) shows that some speedups are possible for larger matrix bandwidths b.

5.2.4 3D Stencil on Clovertown

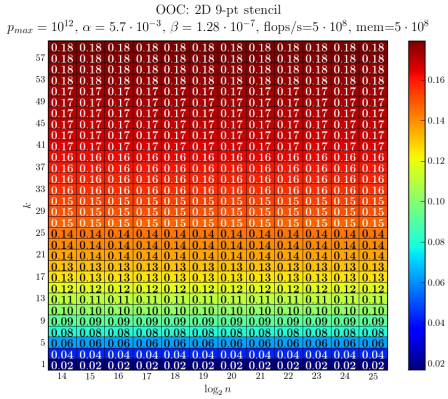
Figure 33(a) shows that for every value of n modeled, a maximum speedup of 1.34x to 1.36x is attainable, by choosing $k = 3$, Figure 33(c) shows that for each n, choosing $k = 3$ raised the fraction of time spent in computation from 28% to 48%. Figure 33(d) shows that this is accomplished by increasing the number of flops by 1.27x. This potential fraction of peak is shown in Figure 33(e), and is 38%, up from 28%. Figure 33(f) shows that speedups are not possible for larger matrix bandwidths b. The speedups in this case are not as impressive as other cases because of the fast memory being too small (this is also confirmed by the analytical model in Figure 13(d)). Figure 13(d) also tells us that even doubling the fast memory size is not expected to give good gains.



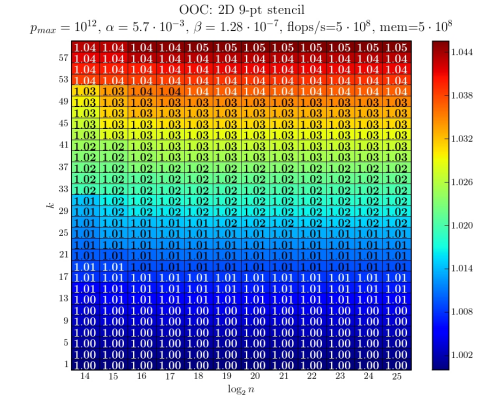
(a) Speedup

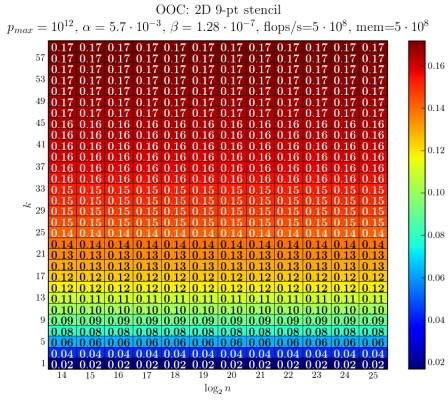


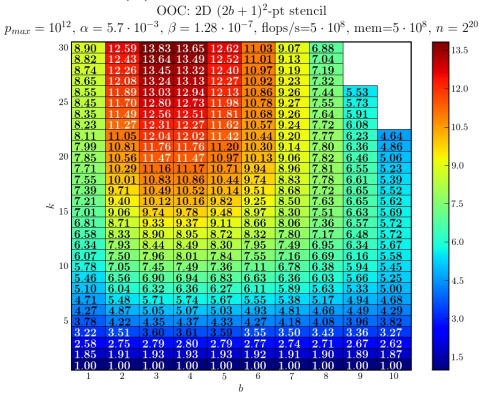
(b) Optimal p



(e) Slowdown vs Conventional Alg with $\alpha = \beta = 0$ (fraction of peak)





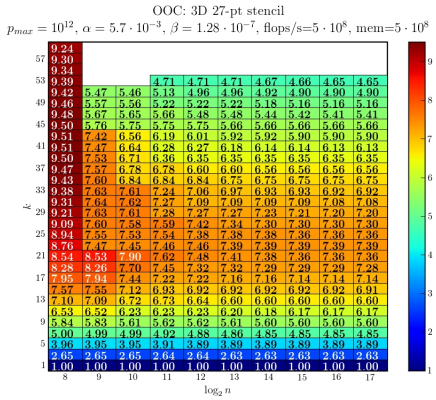


(f) Speedup as a function of matrix bandwidth ($n = 2^{20}$)

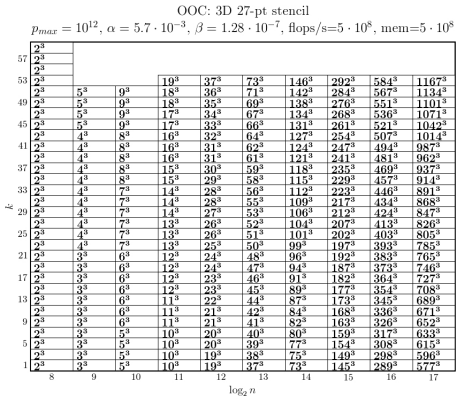
Figure 30: Plots for 2D stencil on OOC. Only odd k shown.

For all n, choosing $k = 59$ yields a speedup of 10.2

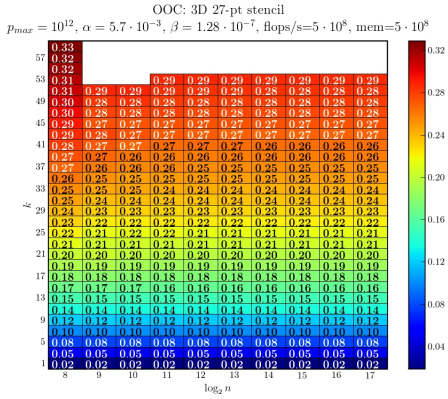
For all n, choosing $k = 59$ yields a fraction of peak of 17%, up from 2%



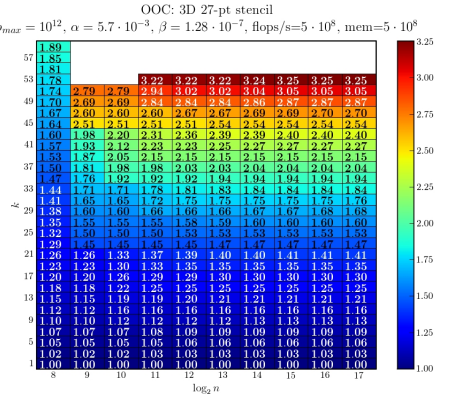
(a) Speedup



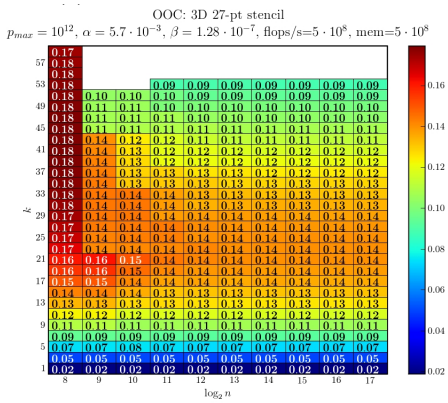
(b) Optimal p



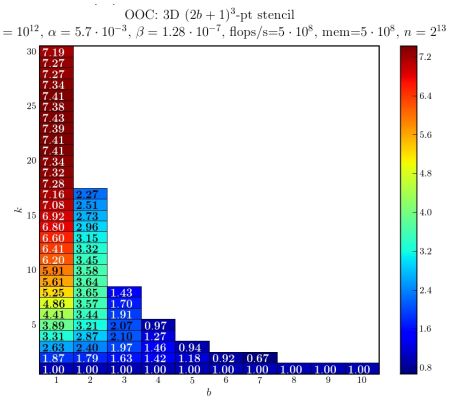
(c) Slowdown vs Conventional Alg with $\alpha = \beta = 0$



(d) Speedup as a function of matrix bandwidth ($n = 2^{13}$)

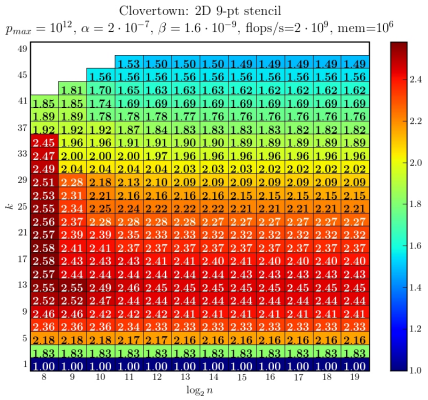


(e) Slowdown vs Conventional Alg with $\alpha = \beta = 0$

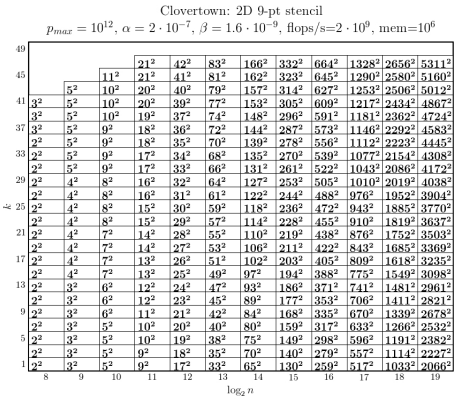


(f) Speedup as a function of matrix bandwidth ($n = 2^{13}$)

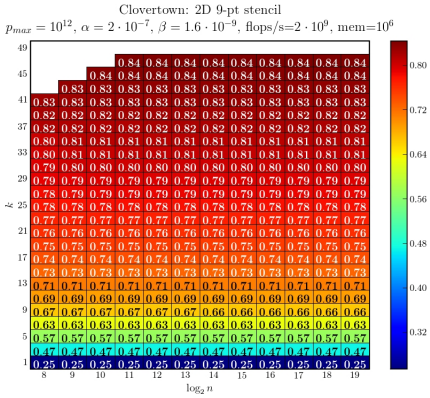
Figure 31: Plots for 3D stencil on OOC. Only odd k shown.
For all n the best k yields a speedup in the range [7.39,9.51]
For all n the best k yields a fraction of peak in the range [14%,18%]



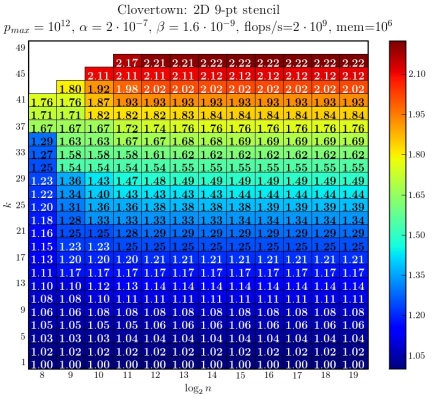
(a) Speedup



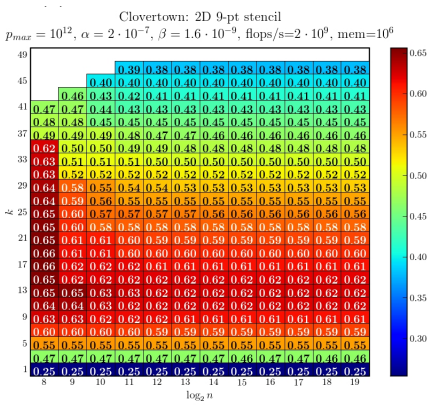
(b) Optimal p



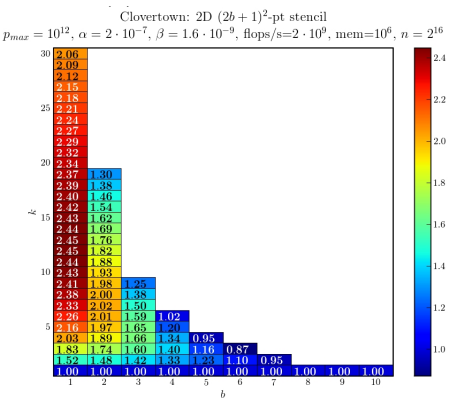
(c) Slowdown vs Conventional Alg with $\alpha = \beta = 0$



(d) Speedup as a function of matrix bandwidth ($n = 2^{16}$)

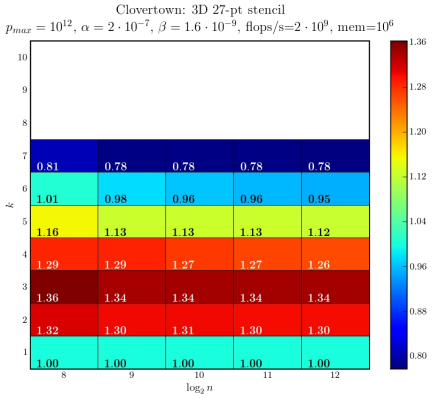


(e) Slowdown vs Conventional Alg with $\alpha = \beta = 0$

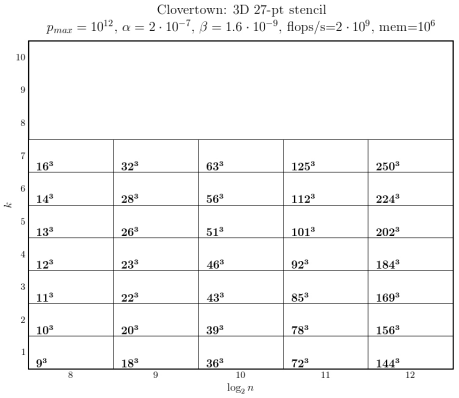


(f) Speedup as a function of matrix bandwidth ($n = 2^{16}$)

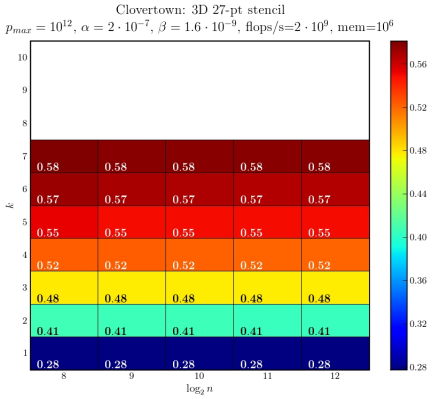
Figure 32: Plots for 2D stencil on Clovertown.
For all n the best k yields a speedup in the range [2.45,2.58]
For all n the best k yields a fraction of peak in the range [.62,.65]

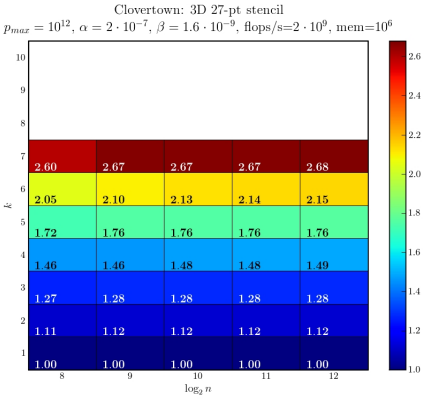


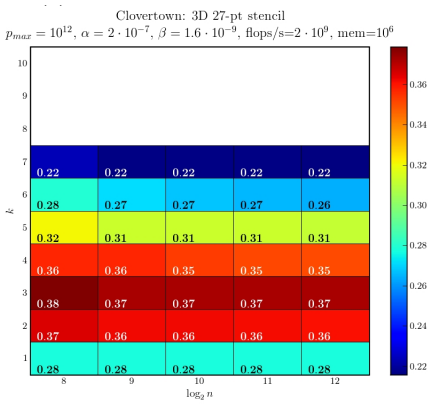
(a) Speedup



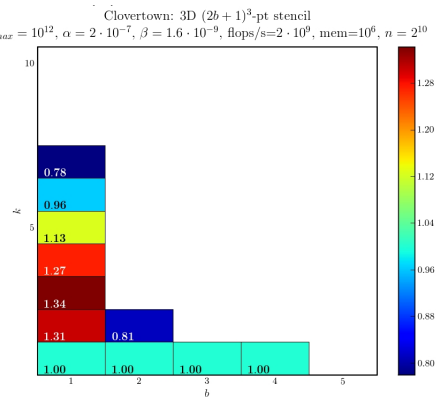
(b) Optimal p







(e) Slowdown vs Conventional Alg with $\alpha = \beta = 0$



(f) Speedup as a function of matrix bandwidth ($n = 2^{10}$)

Figure 33: Plots for 3D stencil on Clovertown.
For all n the best k yields a speedup in the range [1.34,1.36]
For all n the best k yields a fraction of peak of .38.

6 Measured Performance

We have a working implementation of PA1 and PA2 written in UPC [10] that works for arbitrary sparse matrices. Here we report on our implementation of SA2, for which we also have performance measurements.

For the implementation of SA2, we needed to solve the ordering problem described in Section 3.5. Rather than use the TSP formulation described there, we used a simple random sampling strategy to choose a best ordering from a sequence of random orderings. As mentioned in the description of SA2, the computations in SA2 can be ordered in such a way so that they can be done through calls to separate, tuned sparse matrix vector multiplication (SpMV) routines. In our implementation we used the OSKI library [32].

We tested our implementation on the UC Berkeley CITRIS cluster–a cluster of Itanium 2 nodes each with a theoretical peak performance of 5.2 GFlops/s. Each node has 2 Itanium processors with 4 gigabytes of memory per processor.

For performance measurements and parameter estimation, the read and write routines were timed to get the disk read and write bandwidth numbers. The SpMV routine was also timed to obtain the actual floating-point rate for the SpMV routine.

Our test problem was a matrix with a 27 point stencil on a 3D mesh (stored as a general sparse matrix) with $n = 368$ and $p = 64$. Thus the matrix had dimension $368^3 = 49,836,032$ with 27 nonzeros in most rows, broken into $4^3 = 64$ blocks of $(\frac{368}{4})^3 = 92^3 = 778,688$ rows each. The value of p was chosen to optimize performance of the model.

To obtain the most accurate performance model, we used measured values for all important machine parameters: time per floating point operation and disk bandwidth. The disk bandwidth differs significantly for reads and writes, so we augmented our model to distinguish reads and writes. Disk latency turned out to play a negligible role.

- $t_f = 3.12$ ns ($1/t_f = 321$ Mflops/s): This is the measured flop rate for SA2. This was taken as the median of the flop rates observed for the computational phases in SA2.
- $\beta_r = 56$ ns ($1/\beta_r = 143$ MBytes/s): This is the measured read bandwidth.
- $\beta_w = 240$ ns ($1/\beta_w = 33$ MBytes/s): This is the measured write bandwidth.

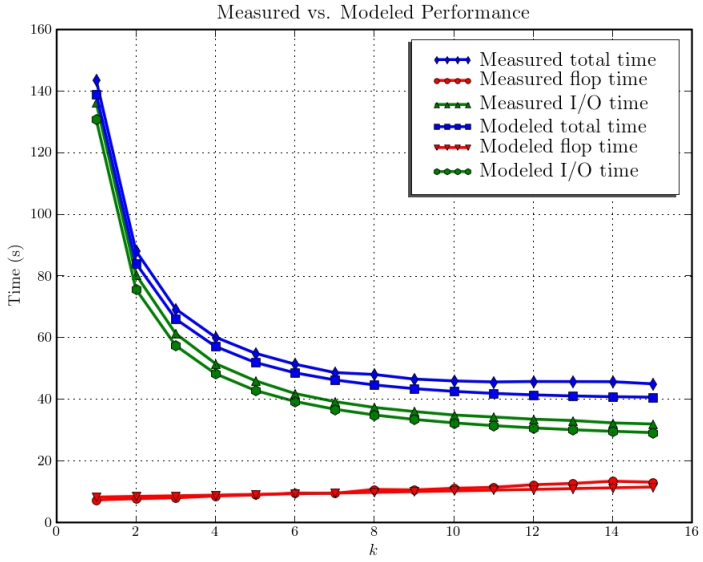
Figures 34(a) and 34(b) show the results, both modeled and measured, which closely match. Figure 34(a) breaks the total runtime down into computation and communication, and Figure 34(b) shows the speedup, which reaches 3.2x at $k = 15$, and is at least $3x$ for $k \geq 8$.

We may also compare our algorithm to one with infinite DRAM, so that the the entire computation can proceed in main memory. Such an algorithm obviously provides an upper bound on our speed. We go from running 20x slower than this algorithm at $k = 1$ to just 6x slower at $k = 15$ (these are measured values).

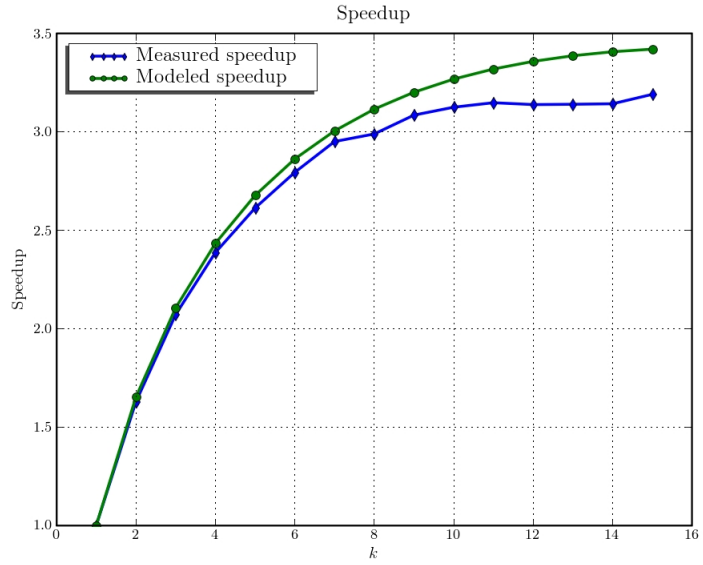
7 Implementing the Preconditioned Kernel $[Ax, MAx, ..., (MA)^k x]$.

Accelerating the kernel $[Ax, A^2x, ..., A^kx]$ is motivated by its potential use in Krylov subspace methods for solving $Ax = b$ or $Ax = \lambda x$. Since preconditioned KSMs are widely used for these problems, we now consider accelerating the analogous "preconditioned kernel"

$$[Ax, MAx, AMAx, MAMAx, ..., A(MA)^{k-1}x, (MA)^k x] \ .$$



(a) Measured vs. Modeled SA2 Performance.



(b) Measured vs. Modeled SA2 speedup.

Figure 34: Measured performance plots for SA2 on Itanium2 CITRIS cluster.

where M is the preconditioner, e.g. we are (implicitly) solving $MAx = Mb$ with a KSM. The intuition to see that this kernel is the right one for preconditioned KSMs comes from examining preconditioned KSMs, which apply A and M to the starting vector in alternation; details will be discussed in another paper. We note that this is the most general kernel, needed by left preconditioned CG or Lanczos; when using either a right preconditioned KSM or a left preconditioned KSM for nonsymmetric matrices (like GMRES), only powers like $(AM)^k x$ or $(MA)^k x$ are needed. Our discussion below applies to these cases as well.

Under certain mathematical conditions on A and its preconditioner M, we will show how to evaluate this kernel for $O(1)$ latency cost, i.e. independent of k. These mathematical conditions are satisfied by sparse matrices like the ones considered so far, and by a wide class of preconditioners (but not all).

We note that the above kernel can be thought of as the pair of kernels

$$[Ax, (AM)Ax, (AM)^2Ax, ..., (AM)^{k-1}Ax] \text{ and } [x, (MA)x, (MA)^2x, ..., (MA)^k x] \ .$$

This means that all the prior techniques in this paper can be applied to both the matrices AM and MA in order to compute the preconditioned kernel. In particular, when A and M are both suitably sparse (such as when M is (block) diagonal), then our prior techniques apply (see section 7.1).

However, many good preconditioners M are in fact dense, even if they are cheap to apply. Again using tridiagonals as a motivating example, consider M as the inverse of a tridiagonal: it may be applied in linear time, but is dense in general, making our techniques presented so far inapplicable. But the inverse of a tridiagonal has another important property: any submatrix strictly above or strictly below the diagonal has rank 1. This off-diagonal low-rank property is shared by many good preconditioners, and we may exploit it to avoid communication too (see section 7.2).

7.1 Exploiting Sparsity of A and M

Our first approach to accelerating the preconditioned kernel will exploit sparsity in A and in MA. This is likely to be the most effective approach for the simplest preconditioners: diagonal, or block diagonal with small blocks.

As stated above, we could consider the two subbases $[Ax, (AM)Ax, (AM)^2Ax, ..., (AM)^{k-1}Ax]$ and $[x, (MA)x, (MA)^2x, ..., (MA)^k x]$ separately, based on the sparsity patterns of AM and MA respectively. Here is another way to look at it. Let B be a matrix whose sparsity pattern contains the union of the sparsity patterns of A and MA. In other words $B_{ij} \neq 0$ if $A_{ij} \neq 0$ or $(MA)_{ij} \neq 0$. For example, let $|A|$ and $|M|$ be the matrices whose entries are the absolute values of A's and M's entries, resp., and let $B = |A| + |M| \cdot |A|$. Then B has the appropriate sparsity pattern (or at least a superset, if cancellation makes $M \cdot A$ sparser than $|M| \cdot |A|$).

Now consider Parallel Approach 1 (PA1) applied to the kernel $[Bx, B^2x, ..., B^k x]$. The remotely stored components of x needed to compute the local components of this kernel are by construction a superset of the ones needed to compute the preconditioned kernel. Thus PA1 extends naturally to preconditioned kernel by using B to determine the communication needed.

In the special case of M diagonal, there is no change to the communication of PA1 at all. The same comments apply to SA1 and SA2, which are based on PA1.

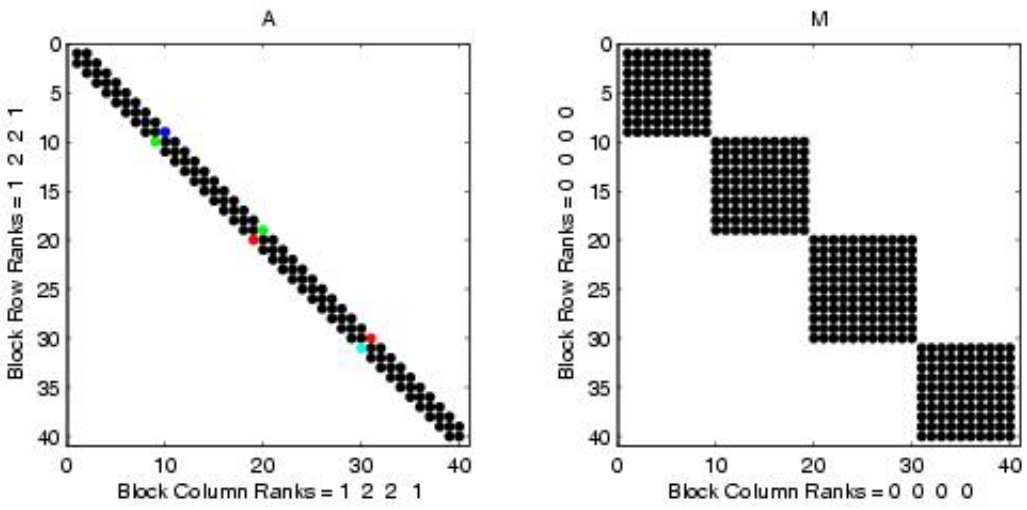


Figure 35: Spy plots of A and M, with A_{diag} in black and $A_{off,i*}$ in colors. The ranks of $A_{off,i*}$ are shown at the left, and the ranks of $A_{off,*j}$ are shown at the bottom. M is displayed similarly.

7.2 Exploiting Low Rank Off-Diagonal Blocks of A and M

For our second approach to accelerating the preconditioned kernel $[Ax, MAx, AMAx, ..., (MA)^k x]$, we will define a property of block matrix A (or M) that limits the rank of its off-diagonal blocks. Suppose A is n-by-n, and let $n = n_1 + n_2 + \cdots + n_p$ be a partitioning, so that we can write A as a matrix with block entries A_{ij} of dimensions n_i-by-n_j. We will also write the i-th block row of A as $A_{i*} = [A_{i1}, ..., A_{ip}]$ and the j-th block column as $A_{*j} = [A_{1j}^T, ..., A_{pj}^T]^T$. We also write $A = A_{off} + A_{diag}$, where A_{off} equals A but with all diagonal blocks $A_{off,ii}$ set to zero, and A_{diag} contains just the diagonal blocks of A. We will also let $x_{(i)}$ refer to the i-th block of the vector x, so we can write $(Ax)_{(i)} = \sum_{j=1}^{p} A_{ij} x_{(j)}$.

Given this notation, we are ready to state

Definition 7.1 *Matrix A has* Property $S(r, n_1, ..., n_p)$ *(with respect to a partitioning $n = n_1 + \cdots + n_p$) if each block row $A_{off,i*}$ and each block column $A_{off,*j}$ of A_{off} has rank at most r. If the partition $n_1, ..., n_p$ is clear from context, we will write $S(r)$. We call r the* off-diagonal rank *of A.*

To illustrate this definition in the simplest possible case, consider a 40-by-40 tridiagonal matrix A (with off-diagonal rank = 2) and a block diagonal preconditioner M (with off-diagonal rank = 0) both with partitioning 40 = 9+10+11+10, as shown in Figure 35. These spyplots show the nonzero entries of A and M, those within the diagonal blocks are shown as black dots, and those within block rows $A_{off,1*}$ through $A_{off,4*}$ shown in blue, green, red and cyan, resp. The ranks of the block rows $A_{off,i*}$ are given at the left of each plot, and the ranks of the block columns $A_{off,*j}$ are given at the bottom of each plot.

Next Figure 36 displays the initial matrices in the preconditioned kernel in the same way: A, MA, AMA,..., $(MA)^4$. We see that the matrices quickly become dense, but their off-diagonal ranks increase only linearly (this will be proven in Lemma 6 below). So even though $(MA)^4$ and higher

powers are dense, the off diagonal parts are low rank, and this will let us store them, multiply by them, and communicate the resulting products inexpensively.

Not just tridiagonals but all our model matrices based on meshes have Property $S(r)$ for various off-diagonal ranks r as shown in the table below. For the 1D meshes the partitions n_i all equal $\frac{n}{p}$, for the 2D meshes the n_i all equal $\frac{n^2}{p}$, and for the 3D meshes the n_i all equal $\frac{n^3}{p}$, with the matrix ordered so that processors own contiguously numbered rows and and columns.

Table 3: Off-diagonal ranks	
1D mesh $b = 1$	2
1D mesh $b \geq 1$	$2b$
2D mesh, 5 pt stencil	$4\frac{n}{p^{1/2}}$
2D mesh, 9 pt stencil	$4(\frac{n}{p^{1/2}} + 1)$
2D mesh, $(2b+1)^2$ pt stencil	$4b(\frac{n}{p^{1/2}} + b)$
3D mesh, 9 pt stencil	$6\frac{n^2}{p^{2/3}}$
3D mesh, 27 pt stencil	$6\frac{n^2}{p^{2/3}} + 12\frac{n}{p^{1/3}} + O(1)$
3D mesh, $(2b+1)^3$ pt stencil	$6b\frac{n^2}{p^{2/3}} + 12b^2\frac{n}{p^{1/3}} + O(b^3)$

Note that the off-diagonal ranks in Table 3 are identical to the entries showing "Words communicated" (for $k = 1$) in the column of Table 1 for the Conventional Approach, and are also equal to the number of nonzero columns in any $A_{off,i*}$, or nonzero rows in any $A_{off,*j}$. This is not a coincidence; our next goal is to show that given any matrix A with Property $S(r)$, we can compute Ax in parallel with the number of words sent from any processor to any other processor bounded by r. This will be true even if A is dense.

To accomplish this, we need a special data structure for A. The diagonal blocks A_{ii} may represented in any desired way, such as by triangular factors L and U with $A_{ii}x = U^{-1}(L^{-1}x)$ being computed by triangular substitution, or with any block-box subroutine that multiplies A_{ii} times a vector $x_{(i)}$. Each off diagonal block A_{ij} will be represented using a rank-r representation $A_{ij} = U_i S_{ij} V_j^T$, where U_i is n_i-by-r, S_{ij} is r-by-r and V_j is n_j-by-r (U_i, S_{ij} and V_j will depend on i and j as described below). Thus for processor i to compute $(Ax)_{(i)}$ it will compute

$$(Ax)_{(i)} = A_{ii}x_{(i)} + \sum_{j \neq i} A_{ij}x_{(j)} = A_{ii}x_{(i)} + U_i \sum_{j \neq i} S_{ij}(V_j^T x_{(j)})$$

As we will see, processor j will own V_j and $x_{(j)}$, and be responsible for computing $V_j^T x_{(j)}$ (a vector of length r) and broadcasting it to all other processors, which will evaluate the rest of the above formula.

In other words, the storage of each offdiagonal block $A_{ij} = U_i S_{ij} V_j^T$ is split between processor i and processor j. In fact the amount of storage is small, as the following lemma shows.

Lemma 5 *Suppose A has Property $S(r)$. Then each A_{ij} may be factored as $A_{ij} = U_i S_{ij} V_j^T$ where U_i is n_i-by-r, S_{ij} is r-by-r, and V_j is n_j-by-r. U_i and V_j may be taken to be orthonormal.*

Proof: By the definition of Property $S(r)$, $A_{off,*j}$ has rank at most r, so its SVD can be written $A_{off,*j} = U_{(j)}\Sigma_{(j)}V_{(j)}^T$, where we may assume w.l.o.g. that $V_{(j)}$ is n_j-by-r and orthonormal. Similarly $A_{off,i*} = U^{(i)}\Sigma^{(i)}(V^{(i)})^T$ where we may assume $U^{(i)}$ is n_i-by-r and orthonormal. Thus each

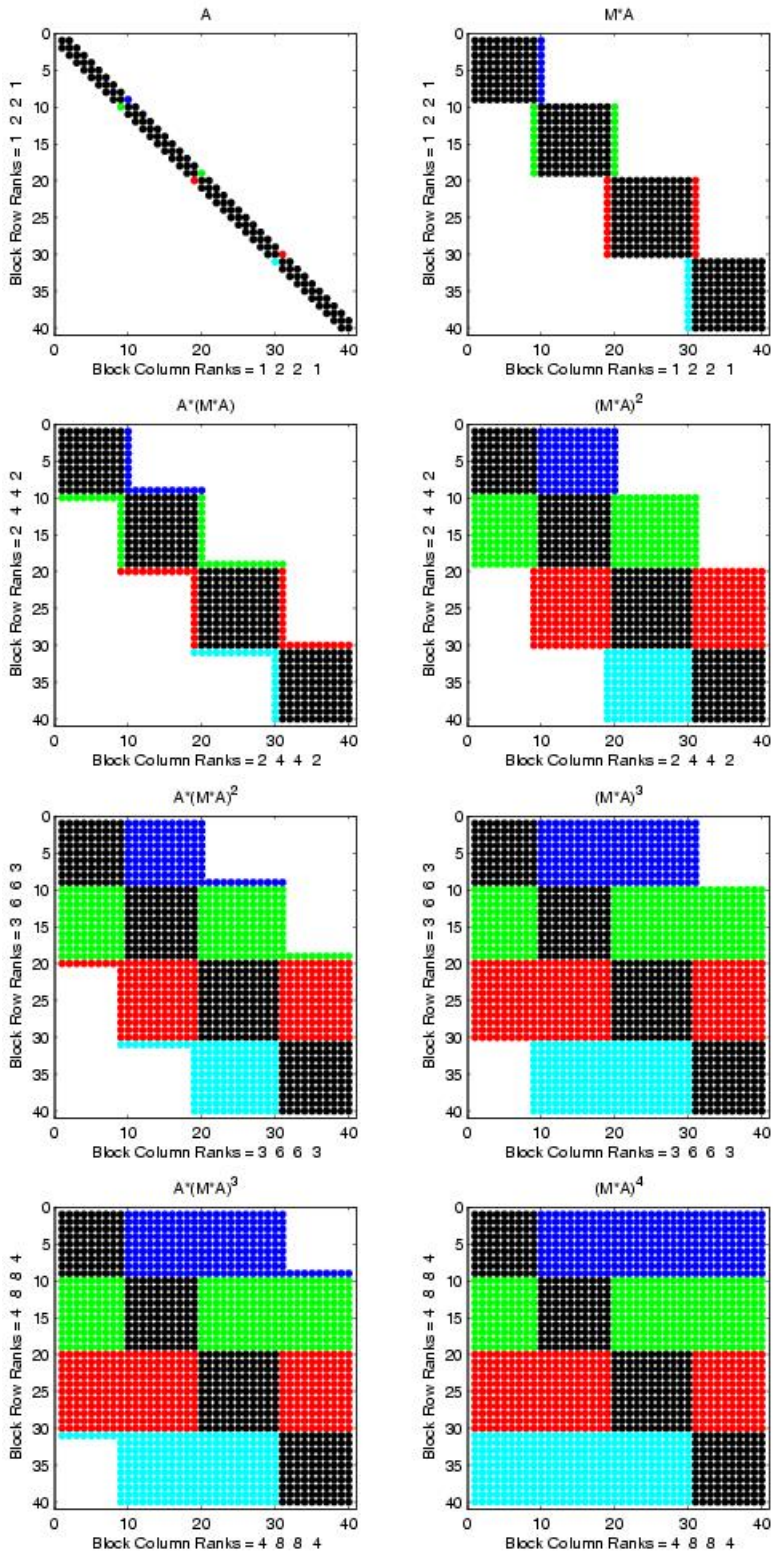


Figure 36: Spy plots of A, MA, AMA, $(MA)^2$, $A(MA)^2$, $(MA)^3$, $A(MA)^3$, and $(MA)^4$, displayed analogously to Figure 35.

A_{ij} has column space spanned by the columns of $U^{(i)}$ and row space spanned by the rows of $V_{(j)}^T$, which means that we can write

$$A_{ij} = U^{(i)}[(U^{(i)})^T A_{ij} V_{(j)}]V_{(j)}^T \equiv U_i[S_{ij}]V_j^T$$

as desired. □

This implies that processor i needs to store V_i ($n_i \cdot r$ words), U_i ($n_i \cdot r$ words), and S_{ij} for $j \neq i$ (up to $(p-1)r^2$ words, if all $A_{off,ij}$ are nonzero), for a total of at most $r(n_i+n_j)+(p-1)r^2$ words, along with A_{ii} (using whatever representation is most natural). This yields the following parallel algorithm for multiplication by a single matrix A with Property $S(r)$:

Alg P_$S(r)$: Parallel multiplication $y = Ax$ where A has Property $S(r)$
... assume matrix A and vectors x and y are stored as described above
each processor j locally computes the r-vector $w_{(j)} = V_j^T x_{(j)}$,
each processor j broadcasts $w_{(j)}$ to all other processors i (with nonzero A_{ij})
each processor j locally computes $y_{(j)} = A_{jj}x_{(j)} + U_j \cdot \sum_{i \neq j, S_{ji} \neq 0} S_{ji} w_{(i)}$.

The flops performed by processor j are the flops for $A_{jj}x_{(j)}$ plus at most $2(n_j+n_i)\cdot r + 2r^2(p-1)$. (The factor $p-1$ may be reduced to the number of nonzero S_{ji} in the summation.) The communication cost for processor j is a broadcast of a vector of length r to (at most) all $p-1$ other processors. When A is sparse, then of course the vector of length r need only be sent to a subset of the other processors (the "nearest neighbors" in the case of the matrices in Table 3). When A is symmetric, then we may take $U_i = V_i$ and so save half the corresponding storage.

We can now compute the kernel $[Ax, MAx, AMAx, ..., (MA)^k x]$ by straightforwardly using Alg P_$S(r)$ $2k$ times, to multiply by A and by M in alternation, assuming A has Property $S(r_A)$ and M has Property $S(r_M)$, both with respect to the same partition $n = n_1 + \cdots + n_p$:

Alg P_Old_Precond_Kernel: Parallel computation of $[Ax, MAx, ..., (MA)^k x]$
where A has Property $S(r_A)$ and M has Property $S(r_M)$
... assume A and M stored as described above
... denote $x^{(0)} = x$, $x^{(m)} = (MA)^m x$ and $y^{(m)} = A(MA)^m x$
for $m = 0$ to $k-1$
use Alg P_$S(r_A)$ to compute $y^{(m)} = A \cdot x^{(m)}$
use Alg P_$S(r_M)$ to compute $x^{(m+1)} = M \cdot y^{(m)}$
end for

The cost of Alg P_Old_Precond_Kernel is k calls to Alg P_$S(r_A)$ and k calls to Alg P_$S(r_M)$, for a total cost to processor j of k multiplications by A_{jj}, k multiplications by M_{jj}, $4kn_j(r_A + r_M) + 2k(p-1)(r_A^2 + r_M^2)$ additional flops, (again, the factor $p-1$ can be reduced to the maximum number of nonzero off-diagonal blocks in either A or M), k "all-to-all" communications in which each processor broadcasts r_A words to all other processors, and another k "all-to-all" communications with r_M words per processor, for a total of $k(r_A + r_M)$ words broadcast per processor.

We will build on Alg P_$S(r)$ to more efficiently implement the kernel $[Ax, MAx, ..., (MA)^k x]$ by using the following observations: (1) The offdiagonal rank of a product is the sum of the off diagonal ranks of the factors (so the offdiagonal rank of $(MA)^k$ grows linearly with k), and (2) the spaces spanned by the block rows and columns of products are nested in a way that limits the communication needed for all the products to just the communication for the longest product.

Lemma 6 *If A has Property $S(r_A)$ and M has property $S(r_M)$ (both respect to the same partition $n = n_1 + \cdots + n_p$) then MA has Property $S(r_A + r_M)$ (all with respect to the same partition $n = n_1 + \cdots + n_p$). Furthermore, let $U_{M,i}$ and $V_{M,j}$ be the matrices from Lemma 5 applied to M, and let $U_{A,i}$ and $V_{A,j}$ be the matrices from Lemma 5 applied to A. Then the columns of the corresponding (orthonormal) matrices $U_{MA,i}$ and $V_{MA,j}$ can be chosen to include the columns of the (orthonormal) matrices $U_{M,i}$ and $V_{A,j}$, respectively.*

Furthermore, each diagonal block $(MA)_{jj}$ has the property that $(MA)_{jj} - M_{jj} \cdot A_{jj}$ has rank at most $r_A + r_M$, with columns and rows also spanned by the columns of $U_{M,j}$ (or $U_{MA,j}$) and rows of $V_{A,j}$ (or $V_{MA,j}$), respectively.

Informally, we say that the block row spaces of MA include the block row spaces of M, and the block column spaces of MA include the block column spaces of A. This is just informal, since cancellation for a particular choice of M and A could lead to the off diagonal rank of MA being smaller than expected. The above and later lemmas work independent of such cancellation.

Proof: Let M' equal M except for zeroing out the j-th block row: $M'_{j*} = 0$. Similarly, let A' equal A except for zeroing out the i-th block column, $A'_{*i} = 0$,

Then the j-th block column of $(MA)_{off}$ can be written

$$(MA)_{off,*j} = M' \cdot A_{*j} = M' \cdot A_{off,*j} + M'_{off,*j} \cdot A_{jj} = M' \cdot (XV_{A,j}^T) + (YV_{M,j}^T) \cdot A_{jj}$$

where X is an n-by-r_A matrix and Y is an n-by-r_M matrix. Thus, the space spanned by the rows of $(MA)_{off,*j}$ is also spanned by the r_A rows of $V_{A,j}^T$ and r_M rows of $V_{M,j}^T A_{jj}$, so $(MA)_{off,*j}$ is of rank at most $r_A + r_M$ as desired. Furthermore, the matrix $V_{MA,j}^T$ whose rows span the row space of $(MA)_{off,*j}$ can be chosen to also span the row space of $A_{off,*j}$, and in fact the (orthonormal) rows of $V_{MA,j}^T$ can be chosen to include the (orthonormal) rows of $V_{A,j}^T$.

Similarly, we can write the i-th block row of $(MA)_{off}$ as

$$(MA)_{off,i*} = M_{i*} \cdot A' = M_{off,i*} \cdot A' + M_{ii} \cdot A'_{off,i*} = (U_{M,i}W^T) \cdot A' + M_{ii} \cdot (U_{A,i}Z^T)$$

where W is an n-by-r_M matrix and Z is an n-by-r_A matrix. Thus, the space spanned by the columns of $(MA)_{off,i*}$ is also spanned by the r_M columns of $U_{M,i}$ and the r_A columns of $M_{ii}U_{A,i}$, so $(MA)_{off,i*}$ is of rank at most $r_A + r_M$ as desired. Furthermore, the matrix $U_{MA,i}$ whose columns span the columns space of $(MA)_{off,i*}$ can be chosen to also span the column space of $M_{off,i*}$, and in fact the (orthonormal) columns of $U_{MA,i}$ can be chosen to include the (orthonormal) columns of $U_{M,i}$.

Now consider the the j-th diagonal block of MA, which can be written

$$(MA)_{jj} = M_{j*} \cdot A_{*j} = M_{off,j*} \cdot A_{off,*j} + M_{jj} \cdot A_{jj}$$

so that $(MA)_{jj} - M_{jj} \cdot A_{jj}$ has columns spanned by the columns of $U_{M,j}$ or $U_{MA,j}$, and rows spanned by the rows of $V_{A,j}^T$ or $V_{MA,j}^T$. □

Lemma 7 *Suppose A has Property $S(r)$. Then A^k has Property $S(kr)$, and furthermore the matrices $U_{A^k,i}$ and $V_{A^k,j}$ whose columns span the block column and block row spaces of A^k, respectively, can be chosen to be n-by-rk, and so their columns span all the spaces spanned by the columns of $U_{A^m,i}$ and $V_{A^m,j}$, respectively, for $m = 1$ to k. Furthermore, the j-th diagonal block $(A^k)_{jj}$ has the property that $(A^k)_{jj} - (A_{jj})^k$ has rank as most kr, with its column space spanned by the columns of $U_{A^k,j}$ and row space spanned by the columns of $V_{A^k,j}$.*

Informally, we say that the row (column) spaces of A^k include all the row (column) spaces of lower powers A, A^2, ..., A^{k-1}.

Proof: To show the desired properties of $(A^k)_{off}$, we use induction. The base case is $k = 2$, which follows directly from Lemma 6. For the induction step, assume the result is true for $k \geq 2$, and apply Lemma 6 to $A^{k+1} = A \cdot A^k$, to conclude that the block column spaces of A^{k+1} include the block column spaces of A^k, which by induction include the block column spaces of A through A^{k-1}. Applying Lemma 6 again to the product $A^{k+1} = A^k \cdot A$ shows that the block row space of A^{k+1} includes the block row spaces of A^k, which by induction include the block row spaces of A through A^{k-1}.

Now consider the j-th diagonal block $(A^k)_{jj}$. The result again follows by Lemma 6 and induction. □

Theorem 1 *Suppose A has property $S(r_A)$ and M has property $S(r_M)$, both with respect to the partition $n = n_1 + \cdots + n_p$. Then*

1. *The i-th block column spaces of MA, $(MA)^2$, ... , $(MA)^k$ are all spanned by the columns of a single n_i-by-$[k(r_A + r_M)]$ (orthonormal) matrix $U_{(MA)^k,i}$.*

2. *The j-th block row spaces of MA, $(MA)^2$, ... , $(MA)^k$ are all spanned by the columns of a single n_j-by-$[k(r_A + r_M)]$ (orthonormal) matrix $V_{(MA)^k,j}$.*

3. *There exist square matrices $T_{ij,m}$ of dimension $k(r_A + r_M)$ such that for $i \neq j$ and $m = 1$ to $m = k$*
$$((MA)^m)_{ij} = U_{(MA)^k,i} \cdot T_{ij,m} \cdot V^T_{(MA)^k,j} \ .$$

4. *There exist square matrices $T_{ii,m}$ of dimension $k(r_A + r_M)$ such that for $m = 1$ to $m = k$ the i-th diagonal block of $(MA)^m$ satisfies*
$$((MA)^m)_{ii} = (M_{ii}A_{ii})^m + U_{(MA)^k,i} \cdot T_{ii,m} \cdot V^T_{(MA)^k,i} \ .$$

5. *The i-th block column spaces of A, $A(MA)$, $A(MA)^2$, ... , $A(MA)^{k-1}$ are all spanned by the columns of a single n_i-by-$[kr_A + (k-1)r_M]$ (orthonormal) matrix $U_{A(MA)^{k-1},i}$.*

6. *The j-th block row spaces of A, $A(MA)$, $A(MA)^2$, ... , $A(MA)^{k-1}$ are all spanned by the columns of a single n_j-by-$[kr_A + (k-1)r_M]$ (orthonormal) matrix $V_{A(MA)^{k-1},j}$.*

7. *There exist square matrices $S_{ij,m}$ of dimension $kr_A + (k-1)r_M$ such that for $i \neq j$ and $m = 0$ to $m = k - 1$*
$$(A(MA)^m)_{ij} = U_{A(MA)^{k-1},i} \cdot S_{ij,m} \cdot V^T_{A(MA)^{k-1},j} \ .$$

8. *There exist square matrices $S_{ii,m}$ of dimension $kr_A + (k-1)r_M$ such that for $m = 0$ to $m = k - 1$ the i-th diagonal block of $A(MA)^m$ satisfies*
$$(A(MA)^m)_{ii} = A_{ii}(M_{ii}A_{ii})^m + U_{A(MA)^{k-1},i} \cdot S_{ii,m} \cdot V^T_{A(MA)^{k-1},i} \ .$$

Proof: Claims 1 and 2 follow directly from Lemma 7. Claim 3 follows from the proof of Lemma 5. Claim 4 follows from Lemma 7 and Lemma 5. Claims 5 and 6 follow from Claims 1 and 2, Lemma 7 and Lemma 6. Claim 7 again follows from the proof of Lemma 5, and Claim 8 from from Lemma 7 and Lemma 5. □

Theorem 1 justifies the correctness of the following parallel algorithm for computing the kernel $[Ax, MAx, AMAx, MAMAx, ..., A(MA)^{k-1}x, (MA)^k x]$.

Alg P_New_Precond_Kernel: Parallel computation of $[Ax, MAx, AMAx, ..., (MA)^k x]$
with one round of communication
... assume A and M have offdiagonal ranks r_A and r_M, resp.
... assume same notation as Theorem 1
... additionally we denote $x^{(m)} = (MA)^m x$ and $y^{(m)} = A(MA)^m x$
each processor j locally computes the $k(r_A + r_M)$-vector $w_{(j)} = V^T_{(MA)^k,j} x_{(j)}$
each processor j locally computes the $(kr_A + (k-1)r_M)$-vector $z_{(j)} = V^T_{A(MA)^{k-1},j} x_{(j)}$
each processor j broadcasts $[w_{(j)}, z_{(j)}]$ to all other processors i
each processor j locally computes the following:
$q = A_{jj} \cdot x_{(j)}$
for $m = 1$ to $k-1$
$q = M_{jj} \cdot q$
$x^{(m)}_{(j)} = q + U_{(MA)^k,j} \cdot (\sum^p_{i=1,T_{ji,m}\neq 0} T_{ji,m} \cdot w_{(i)})$
$q = A_{jj} \cdot q$
$y^{(m)}_{(j)} = q + U_{A(MA)^{k-1},j} \cdot (\sum^p_{i=1,S_{ji,m}\neq 0} S_{ji,m} \cdot z_{(i)})$
end for
$q = M_{jj} \cdot q$
$x^{(k)}_{(j)} = q + U_{(MA)^k,j} \cdot (\sum^p_{i=1,T_{ji,k}\neq 0} T_{ji,k} \cdot w_{(i)})$

The cost of this algorithm for processor j is k local matrix-vector multiplications by M_{jj}, k local matrix-vector multiplications by A_{jj}, another

$$\begin{aligned} 2pk^3(r_A + r_M)^2 &+ 2n_j k^2(r_A + r_M) + 2p(k-1)(kr_A + (k-1)r_M)^2 \\ &+ 2n_j(k-1)(kr_A + (k-1)r_M) + 2n_j k(k+1)(r_A + r_M) \\ &\leq 4pk^3(r_A + r_M)^2 + 4n_j k^2(r_A + r_M) + 2n_j k(k+1)(r_A + r_M) \\ &\leq 6k(k+1)n_j(r_A + r_M) + 4pk^3(r_A + r_M)^2 \end{aligned}$$

flops, and one "all-to-all" communication of $2kr_A + (2k-1)r_M$ words per processor.

Table 4 compares the costs of the straightforward parallel algorithm Alg P_Old_Precond_Kernel and our new algorithm Alg P_New_Precond_Kernel. The memory costs excludes the memory needed for A_{jj} and M_{jj}, but includes the memory for all the vectors and locally stored S, T, U and V matrices. The cost of (pre)computing these matrices is not included, since it is a one-time cost amortized over all the iterations.

Table 4: Summary Table of Costs of Parallel Preconditioned Kernel		
	P_Old_Precond_Kernel	P_New_Precond_Kernel
Mults by A_{jj}	k times	k times
Mults by M_{jj}	k times	k times
Other Flops (besides A_{jj}, M_{jj})	$4kn_j(r_A + r_M)$ $+2(p-1)k(r_A^2 + r_M^2)$	$6k(k+1)n_j(r_A + r_M)$ $+4pk^3(r_A + r_M)^2$
# All-to-all bcasts	$2k$	1
Words bcast	$kr_A + kr_M$	$2kr_A + (2k-1)r_M$
Memory (besides A_{jj}, M_{jj})	$(2k+1)n_j + 2\max(r_A, r_M)$ $+2n_j(r_A + r_M)$ $+(p-1)(r_A^2 + r_M^2)$	$(2k+1)n_j + 2(2kr_A + (2k-1)r_M)$ $+2n_j(2kr_A + (2k-1)r_M)$ $+p(kr_A + kr_M)^2 + p(kr_A + (k-1)r_M)^2$

The costs in Table 4 are worst case, and do not include a variety of optimizations such as noting that the $S_{ij,m}$ and $T_{ij,m}$ matrices themselves have low rank for small m, and that $A(MA)^{m-1}$ and $(MA)^m$ may be sparse for small m (see Figure 36), thereby reducing the factor p in the flops and memory counts. Still, we may use it to make the following observations.

Like the kernel $[Ax, A^2x, ..., A^kx]$, Table 4 shows that the number of messages is independent of k, as desired. Unlike $[Ax, A^2x, ..., A^kx]$, the bandwidth term (number of words broadcast) roughly doubles, rather than growing only by lower order "boundary" terms.

Most significantly, the number of extra flops can be large. The "diagonal work" (multiplications by A_{jj} and M_{jj} is identical, but in the two terms contributing to "Other Flops", the first one (the one proportional to n_j) increases by a factor of $1.5(k+1)$, and the second one increases by a factor of about $2k^2$.

To make a concrete comparison let us consider a 3D n-by-n-by-n mesh with a $(2b+1)^3$ stencil and block-Jacobi preconditioning, i.e. preconditioned by a block diagonal preconditioner M with $M_{jj} = A_{jj}^{-1}$, implemented by doing triangular substitution with the precomputed sparse LU factorization of A_{jj}. Initially we will not count the cost of computing this triangular factorization, assuming it is amortized over the subsequent multiple iterations. Our basic algorithm (that we will compare to P_New_Precond_Kernel) will ignore the low off-diagonal rank of A and M, and simply alternatingly multiply by A (using standard parallel sparse matrix-vector multiplication) and multiply by M (by local triangular substitution). To do a $O()$ analysis, we will reduce the factor p in the flop count to $O(k^d)$, where d is the dimension of the mesh, which better describes how the number of nonzero offdiagonal blocks grows in this case. To simplify, we will only display the leading terms in the number of flops, words communicated, and number of messages, assuming $n^3 \gg p \gg b, k$.

For $b = 1$, the number of nonzeros in the triangular factors of A_{jj} is known to be $O((\dim(A_{jj}))^{4/3}) = O(\frac{n^4}{p^{4/3}})$, so for the sake of approximate comparison we will take the number of nonzeros for general b to be $O(b^3\frac{n^4}{p^{4/3}})$. The number of nonzeros is half the number of flops needed to multiply by M_{jj}.

Table 5: Comparison Table for n-by-n-by-n mesh with $(2b+1)^3$ stencil		
	Conventional Algorithm	P_New_Precond_Kernel
Flops from M_{jj}	$O(b^3k\frac{n^4}{p^{4/3}})$	$O(b^3k\frac{n^4}{p^{4/3}})$
Other Flops	$16b^3k\frac{n^3}{p}$	$24bk^2\frac{n^5}{p^{5/3}} + 144b^2k^3\frac{n^4}{p^{1/3}}$
# All-to-all bcasts	0	1
Words bcast	0	$12bk\frac{n^2}{p^{2/3}}$
# pt-to-pt msgs	$26k$	0
Words sent	$6bk\frac{n^2}{p^{2/3}}$	0

As can be seen from Table 5, the number of "other flops" in P_New_Precond_Kernel is of higher order than the number of flops in the Conventional Algorithm, namely

$$O(b^3k\frac{n^4}{p^{4/3}} + bk^2\frac{n^5}{p^{5/3}} + b^2k^3p\frac{n^4}{p^{1/3}}) = O(b^3k\frac{n^4}{p^{4/3}}(1 + \frac{k}{b^2}\frac{n}{p^{1/3}} + \frac{pk^2}{b})) \quad \text{versus} \quad O(b^3k\frac{n^4}{p^{4/3}}) \ .$$

Thus when k and b are both $O(1)$, the P_New_Precond_Kernel will do $O(\frac{n}{p^{1/3}} + p)$ times as many flops as the Conventional Algorithm. Unless the latency is truly large, this is probably too high a penalty to pay.

Fortunately, Table 4's flop count bounds may be reduced. The factor p above is a worst case, assuming k is so large that most powers $(MA)^m$ are dense. In fact when m is smaller than p, $(MA)^m$ can be sparse, as Figure 36 shows. Since we expect $k \ll p$, the factor p can be significantly reduced.

On the other hand, the factor $\frac{n}{p^{1/3}}$ (which ultimately comes from the factor $n_j = \frac{n^3}{p}$ in the "Flops" row in Table 4) probably cannot be reduced. This is because multiplying by a matrix of low rank r which is also dense has a cost proportional to its dimension, in contrast to multiplying by a sparse matrix with r nonzeros (and so at most the same rank). As Figure 36 shows, the off-diagonal blocks are indeed dense. Reducing the arithmetic complexity further will require introducing further approximations of these updates.

This discussion has so far not counted the cost of computing the triangular factorization of each A_{jj}, namely $O(\frac{b^3n^6}{p^2})$, assuming it is amortized over the subsequent solves. If we include this cost in the "flops from M_{jj}", we see that the cost of computing and applying the preconditioner dominates the flop costs of both the Conventional Algorithm and P_New_Precond_Kernel, making the reduced latency costs of P_New_Precond_Kernel attractive.

In other words, if the flop cost of the block diagonal preconditioner is high enough, the extra flops introduced by our scheme in order to lower latency costs will not be dominant.

Finally, we note that if M is block diagonal with smaller blocks, or indeed completely diagonal, then the complexity decreases even further.

8 Related Work

The optimizations described in this paper belong to a collection of techniques for improving the performance of applying a stencil repeatedly to a regular discrete domain, or multiplying a vector repeatedly by a sparse matrix. They, in turn, are a subset of various methods known as *tiling* or *blocking*. They all involve decompositions of the d-dimensional domain into d-dimensional subdomains, and rearranging the order of arithmetic operations in order to exploit the parallelism and/or

temporal locality implicit in those subdomains. The different order of operations does not change the result in exact arithmetic, although it may affect rounding error in important ways.

Tiling research falls into three general categories. The first encompasses performance-oriented implementations and implementation suggestions, as well as practical performance models thereof. See, for example, [22, 21, 14, 35, 23, 19, 27, 36, 9, 28, 8, 37, 33, 17, 16]. Many of these techniques have been independently rediscovered several times. The second category consists of theoretical algorithms and asymptotic performance analyses. These are based on sequential or parallel processing models which account for the memory hierarchy and/or inter-processor communication costs. Works that specifically discuss stencils or more general sparse matrices include [13], [18], and [29]. Works which may be more generally applicable include [1, 3, 4]. The third category contains suggested applications that call for repeated application of a stencil (resp. sparse matrix) to a domain (resp. vector). See, for example, [30, 24, 6, 7, 2, 28].

Tiling was possibly inspired by two existing techniques. The first, *domain decompositions* for solving partial differential equations (PDEs), originated in an 1870 paper by H. A. Schwarz[25]. Typical domain decompositions work directly with the continuous PDE. They iterate between invoking an arbitrary PDE solver on each of the subdomains, and correcting for the subdomain boundary conditions. Unlike tiling, domain decompositions change the solution method (even in exact arithmetic) as well as the order of computations. The second inspiration may be *out-of-core* (OOC) stencil codes[3], which date back to the early days of electronic computing [15, 22]. Small-capacity primary memories and the lack of a virtual memory system on early computers necessitated operating only on small subsets of a domain at a time.

Tiling of stencil codes became popular for performance as a result of two developments in computer hardware: increasingly deep memory hierarchies, and multiple independent processors [14]. Existing techniques for reorganizing stencil codes on vector supercomputers divided up a d-dimensional domain into $d-1$-dimensional hyperplanes. The resulting long streams of data suited the vector architectures of the time, which were typically cacheless and relied on a careful balance between memory bandwidth and floating-point rate in order to achieve maximum performance. In contrast, cache-based architectures usually lack the memory bandwidth necessary to occupy their floating-point units. Achieving near-peak floating-point performance thus requires exploiting locality. Furthermore, iterating over a domain by hyperplanes failed to take advantage of the potential temporal locality of many stencil codes. Tiling decreases the number of connections between subdomains, which reduces the number of cache misses, and decouples data dependencies that hinder extraction of non-vector parallelism.

The idea of using redundant computation to avoid communication or slow memory accesses in stencil codes may be as old as OOC stencil codes themselves. Leiserson et al. cite a reference from 1963 [18, 22], and Hong and Kung analyze a typical strategy on a 2-D grid [13]. (We codify this strategy in our PA1 algorithm.) Nevertheless, many tilings do not involve redundant computation. For example, Douglas et al. describe a parallel tiling algorithm that works on the interiors of the tiles in parallel, and then finishes the boundaries sequentially [9]. Many sequential tilings do not require redundant computations [16]; our SA1 algorithm does not.

However, at least in the parallel case, tilings with redundant computation have the advantage of requiring only a single round of messages, if the stencil is applied several times. The latency penalty is thus independent of the number of applications, though the bandwidth requirements increase.

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

Furthermore, Strout et al. point out that the sequential fill-in of boundary regions suggested by Douglas et al. suffers from poor locality [28]. Most importantly, redundant computation to save messages is becoming more and more acceptable, given the divergence in hardware improvements between latency, bandwidth, and floating-point rate.

Application of stencil tiling to more general sparse matrices seems natural, as many types of sparse matrices express the same kind of local connections and domain topology as stencils. However, typical tiling approaches for stencils rely on static code transformations, based on analysis of loop bounds. Sparse matrix-vector multiplication usually requires indirect memory references in order to express the structure of the matrix, and loop bounds are dependent on this structure. Thus, implementations usually require runtime analysis of the sparse matrix structure, using a graph partitioner to decompose the domain implied by the matrix. This was perhaps somewhat unnatural to researchers pursuing a comiler-based approach to tuning stencil codes. Furthermore, graph partitioning has a nontrivial runtime cost, and finding an optimal one is an NP-complete problem which must be approximated in practice. Theoretical algorithms for the out-of-core sequential case already existed (see e.g., [18]), but Douglas et al. were apparently the first to attempt an implementation of parallel tiling of a general sparse matrix, in the context of repeated applications of a multigrid smoother [9]. This was extended by Strout et al. into a sequential cache optimization which is most like our SA1 algorithm.

Our work differs from existing approaches in many ways. First, we developed our methods in tandem with an algorithmic justification: communication-avoiding (also called s-step) Krylov subspace methods. Toledo had suggested an s-step variant of conjugate gradient iteration, based on a generalization of PA1, but he did not supply an implementation for matrices more general than tridiagonal matrices [29]. We have a full implementation of PA1 for general sparse matrices, and have demonstrated significant performance increases on a wide variety of platforms. Douglas et al. and Strout developed their matrix powers kernel for classical iterations like Gauss-Seidel [9, 28]. However, these iterations' most common use in modern linear solvers are as multigrid smoothers. The payoff of applying a smoother s times in a row decreases rapidly with s; this is, in fact, the main inspiration for multigrid. Douglas et al. acknowledge that usually $1 \leq s \leq 5$ [9]. In contrast, communication-avoiding Krylov subspace methods are potentially much more scalable in s. Saad also suggested applying something like a matrix powers kernel to polynomial preconditioning, but here again, increasing the degree of the polynomial preconditioner tends to have a decreasing payoff, in terms of the number of CG iterations required for convergence [24].

We have also expanded the space of possible algorithms by including PA2 and SA2. PA2 avoids some redundant computation, but offers less opportunity for overlapping communication and computation. SA2 extends SA1 for the case in which the vectors (as well as the matrix) do not fit entirely in fast memory. As far as we can tell, PA2 and SA2 are novel, including the TSP formulation for optimizing SA2.

We also have variants of the matrix powers kernel which directly address numerical stability concerns by generating a different basis than the usual "monomial basis" $[x, Ax, A^2x, \ldots]$. Previous authors had expressed these concerns (see e.g., [6]) and suggested bases that use "shifts" based on eigenvalue estimates, such as the Chebyshev or Newton bases [7, 2]. The shifts can be chosen and improved using runtime information which is a practically free byproduct of the Krylov subspace method that uses the kernel. However, these authors did not suggest an optimized matrix powers kernel which could accommodate these bases.

In addition, we have developed a novel "preconditioned matrix powers kernel," for use in precon-

ditioned Krylov subspace methods. Previous authors had only vaguely addressed preconditioning in these algorithms. We are also the first, as far as we can tell, to identify a set of nontrivial preconditioners – those which can be partitioned such that their off-diagonal blocks are low rank – as best suited for the preconditioned matrix powers kernel. This lets us exploit a variety of preconditioners, including $\mathcal{H}$ and $\mathcal{H}^2$ matrices (see e.g., [12, 11]), that are highly effective for solving a large class of problems with hierarchical structure. Such preconditioners are especially useful for performance tuning because they expose the tradeoff between preconditioner effectiveness and performance.

Furthermore, our performance models use latency, bandwidth, and floating-point rates from actual or predicted machines. Where possible, these rates were measured rather than peak rates. The models show that our techniques can be useful in a wide variety of situations. These include single-processor out-of-core, distributed-memory parallel, and shared-memory parallel. The latter case is especially relevant considering the advent of multi- and manycore shared-memory parallel processors, and predictions that Moore's Law performance increases will now be expressed in "number of processors per chip." Finally, the models demonstrate that applying our algorithms may often be just as good or better for performance than improving the communication hardware while using the naive algorithm. This opens the door to parallel configurations previously thought impractical for Krylov methods due to high latency, such as internet-based grid computing or wireless devices.

References

[1] Matthew Andrews, Tom Leighton, P. Takis Metaxas, and Lisa Zhang. Automatic methods for hiding latency in high bandwidth networks (extended abstract). In *STOC: ACM Symposium on Theory of Computing (STOC)*, 1996.

[2] Z. Bai, D. Hu, and L. Reichtel. A Newton basis GMRES implementation. *IMA Journal of Numerical Analysis*, 14:563–581, 1994.

[3] Gianfranco Bilardi and Franco P. Preparata. Processor–time tradeoffs under bounded-speed message propagation: Part i, upper bounds. *Theory of Computing Systems*, 30(6), November 1997.

[4] Gianfranco Bilardi and Franco P. Preparata. Processor–time tradeoffs under bounded-speed message propagation: Part ii, lower bounds. *Theory of Computing Systems*, 32(5), September 1999.

[5] S. Börm, L. Grasedyck, and W. Hackbusch. *Hierarchical Matrices.* Max Planck Institute for Mathematics in the Sciences, 2006. www.mis.mpg.de/preprints/ln/lecturenote-2103.abstr.html.

[6] A. T. Chronopoulos and C. W. Gear. s-step iterative methods for symmetric linear systems. *J. Comput. Appl. Math.*, 25(2):153–168, 1989.

[7] Eric de Sturler. A parallel variant of GMRES(m). In J. J. H. Miller and R. Vichnevetsky, editors, *Proceedings of the 13th IMACS World Congress on Computation and Applied Mathematics*, Dublin, Ireland, 1991. Criterion Press.

[8] C. Ding and Y. He. A ghost cell expansion method for reducing communications in solving PDE problems. In *Proceedings. of SC2001*, November 2001.

[9] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss. Cache optimization for structured and unstructured grid multigrid. *Electronic Transaction on Numerical Analysis*, 10:21–40, February 2000.

[10] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared-Memory Programming.* Wiley-Interscience, May 2005.

[11] Lars Grasedyck and Wolfgang Hackbusch. Construction and arithmetics of $\mathcal{H}$-matrices. *Computing*, 70:295–334, 2003.

[12] Wolfgang Hackbusch, Boris Khoromskij, and Stefan Sauter. On $\mathcal{H}^2$-matrices. In *Lectures on Applied Mathematics*, pages 9–29. Springer, Berlin, Germany, 2002.

[13] Jai-Wei Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. 13th Ann. ACM Symp. on Theory of Computing (May 11-13, 1981)*, pages 326–333, 1981.

[14] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 319–329. ACM Press, 1988.

[15] W. Kahan. Out-of-core stencil code in the 1950's, 2007.

[16] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *Memory Systems Performance and Correctness*, San Jose, CA, October 2006.

[17] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *3rd Annual ACM SIGPLAN Workshop on Memory Systems Performance*, Chicago, IL, 2005.

[18] Charles E. Leiserson, Satish Rao, and Sivan Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers (extended abstract). In *IEEE Symposium on Foundations of Computer Science*, pages 704–713, 1993.

[19] J. McCalpin and D. Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. Technical Report DCS-TR-379, Department of Computer Science, Rutgers University, 1999.

[20] D. Patterson. Latency lags bandwidth. *CACM*, 47(10):71–75, Oct 2004.

[21] J.-K. Peir. *Program partitioning and synchronization on multiprocessor systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, March 1986.

[22] C. J. Pfeifer. Data flow and storage allocation for the PDQ-5 program on the Philco-2000. *Communications of the ACM*, 6(7):365–366, 1963.

[23] E. J. Rosser. *Fine-grained analysis of array computations*. PhD thesis, Dept. of Computer Science, University of Maryland, September 1998.

[24] Youcef Saad. Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM J. Sci. Stat. Comput.*, 6(4), October 1985.

[25] H. A. Schwarz. Über einen grenzubergang durch alternierendes verfahren. *Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich*, 15:272–286, 1870. Available in [**?**].

[26] M. Snir and S. Graham, editors. *Getting up to speed: The Future of Supercomputing*. National Research Council, 2004. 227 pages.

[27] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, 1999.

[28] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Rescheduling for locality in sparse matrix computations. In V. N. Alexandrov and J. J. Dongarra, editors, *Lecture Notes in Computer Science*. Springer, 2001.

[29] Sivan Toledo. *Quantitative performance modeling of scientific computations and creating locality in numerical algorithms*. PhD thesis, Massachusetts Institute of Technology, June 1995.

[30] J. van Rosendale. Minimizing inner product data dependence in conjugate gradient iteration. In *Proc. IEEE Internat. Confer. Parallel Processing*, 1983.

[31] R. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, Computer Science Division, University of California, Berkeley, 2003.

[32] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. of SciDAC 2005, J. of Physics: Conference Series*. Institute of Physics Publishing, June 2005.

[33] Richard Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California Berkeley, December 2003.

[34] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Supercomputing 07*. IEEE, 2007. to appear.

[35] M. E. Wolf. *Improving locality and parallelism in nested loops*. PhD thesis, Stanford University, 1992.

[36] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedings of the Fourteenth International Parallel and Distributed Processing Symposium (IPDPS)*, pages 171–180, 2000.

[37] P. R. Woodward and S. E. Anderson. Scaling the Teragrid by latency tolerant application design. In *Proc. of NSF / Department of Energy Scaling Workshop*, Pittsburg, CA, May 2002.