Acta Numerica (2014), pp. 1–155
doi:10.1017/S0962492914000038

Communication lower bounds and optimal algorithms for numerical linear algebra*†

G. Ballard[1], E. Carson[2], J. Demmel[2,3], M. Hoemmen[4], N. Knight[2] and O. Schwartz[2]

[1] *Sandia National Laboratories, Livermore, CA 94551, USA*
E-mail: gmballa@sandia.gov

[2] *EECS Department, UC Berkeley, Berkeley, CA 94704, USA*
E-mail: ecc2z@cs.berkeley.edu, knight@cs.berkeley.edu, odedsc@cs.berkeley.edu

[3] *Mathematics Department, UC Berkeley, Berkeley, CA 94704, USA*
E-mail: demmel@cs.berkeley.edu

[4] *Sandia National Laboratories, Albuquerque, NM 87185, USA*
E-mail: mhoemme@sandia.gov

* We acknowledge funding from Microsoft (award 024263) and Intel (award 024894), and matching funding by UC Discovery (award DIG07-10227). Additional support comes from ParLab affiliates National Instruments, Nokia, NVIDIA, Oracle and Samsung, as well as MathWorks. Research is also supported by DOE grants DE-SC0004938, DE-SC0005136, DE-SC0003959, DE-SC0008700, DE-SC0010200, DE-FC02-06-ER25786, AC02-05CH11231, and DARPA grant HR0011-12-2-0016. This research is supported by grant 3-10891 from the Ministry of Science and Technology, Israel, and grant 2010231 from the US–Israel Bi-National Science Foundation. This research was supported in part by an appointment to the Sandia National Laboratories Truman Fellowship in National Security Science and Engineering, sponsored by Sandia Corporation (a wholly owned subsidiary of Lockheed Martin Corporation) as Operator of Sandia National Laboratories under its US Department of Energy Contract DE-AC04-94AL85000.
† Colour online for monochrome figures available at journals.cambridge.org/anu.

The traditional metric for the efficiency of a numerical algorithm has been the number of arithmetic operations it performs. Technological trends have long been reducing the time to perform an arithmetic operation, so it is no longer the bottleneck in many algorithms; rather, *communication*, or moving data, is the bottleneck. This motivates us to seek algorithms that move as little data as possible, either between levels of a memory hierarchy or between parallel processors over a network. In this paper we summarize recent progress in three aspects of this problem. First we describe lower bounds on communication. Some of these generalize known lower bounds for dense classical ($O(n^3)$) matrix multiplication to all direct methods of linear algebra, to sequential and parallel algorithms, and to dense and sparse matrices. We also present lower bounds for Strassen-like algorithms, and for iterative methods, in particular Krylov subspace methods applied to sparse matrices. Second, we compare these lower bounds to widely used versions of these algorithms, and note that these widely used algorithms usually communicate asymptotically more than is necessary. Third, we identify or invent new algorithms for most linear algebra problems that do attain these lower bounds, and demonstrate large speed-ups in theory and practice.

CONTENTS

1. Introduction

1.1. Motivation

Linear algebra problems appear throughout computational science and engineering, as well as the analysis of large data sets (Committee on the Analysis of Massive Data; Committee on Applied and Theoretical Statistics; Board on Mathematical Sciences and Their Applications; Division on Engineering and Physical Sciences; National Research Council 2013), so it is important to solve them as efficiently as possible. This includes solving systems of linear equations, least-squares problems, eigenvalue problems, the singular

value decomposition, and their many variations that can depend on the structure of the input data.

When numerical algorithms were first developed (not just for linear algebra), efficiency was measured by counting arithmetic operations. Over time, as technological trends such as Moore's law kept making operations faster, the bottleneck in many algorithms shifted from arithmetic to *communication*, that is, moving data, either between levels of the memory hierarchy such as DRAM and cache, or between parallel processors connected over a network. Communication is necessary because arithmetic can only be performed on two operands in the same memory at the same time, and (in the case of a memory hierarchy) in the smallest, fastest memory at the top of the hierarchy (*e.g.*, cache). Indeed, a sequence of recent reports (Graham, Snir and Patterson 2004, Fuller and Millett 2011) has documented this trend. Today the cost of moving a word of data (measured in time or energy) can exceed the cost of an arithmetic operation by orders of magnitude, and this gap is growing exponentially over time.

Motivated by this trend, the numerical linear algebra community has been revisiting all the standard algorithms, direct and iterative, for dense and sparse matrices, and asking three questions: Are there lower bounds on the amount of communication required by these algorithms? Do existing algorithms attain the lower bounds? If not, are there new algorithms that do?

The answers, which we will discuss in more detail in this paper, are briefly as follows. There are in fact communication lower bounds for most direct and iterative (*i.e.*, Krylov subspace) algorithms. These lower bounds apply to dense and sparse matrices, and to sequential, parallel and more complicated computer architectures. Existing algorithms in widely used libraries often do asymptotically more communication than these lower bounds require, even for heavily studied operations such as dense matrix multiplication (matmul for short). In many cases there are new algorithms that do attain the lower bounds, and show large speed-ups in theory and practice (even for matmul). These new algorithms do not just require 'loop transformations' but sometimes have different numerical properties, different ways to represent the answers, and different data structures.

Historically, the linear algebra community has been adapting to rising communication costs for a long time. For example, the level-1 Basic Linear Algebra Subroutines (BLAS1) (Lawson, Hanson, Kincaid and Krogh 1979) were replaced by BLAS2 (Dongarra, Croz, Hammarling and Hanson 1988*a*, 1988*b*) and then BLAS3 (Dongarra, Croz, Duff and Hammarling 1990*a*, 1990*b*), and EISPACK (Smith *et al.* 1976) and LINPACK (Dongarra, Moler, Bunch and Stewart 1979) were replaced by LAPACK (Anderson *et al.* 1992) and ScaLAPACK (Blackford *et al.* 1997), to mention a few projects. So it may be surprising that large speed-ups are still possible.

1.2. Modelling communication costs

More precisely, we will model the cost of communication as follows. There are two costs associated with communication. For example, when sending n words from one processor to another over a network, the words are first packed into a contiguous block of memory called a *message*, which is then sent to the destination processor. There is a fixed overhead time (called the *latency cost* or α) required for the packing and transmission over the network, and also time proportional to n needed to transmit the words (called the *bandwidth cost* or βn). In other words, we model the time to send one message of size n by $\alpha + \beta n$, and the time to send S messages containing a total of W words by $\alpha S + \beta W$.

Letting γ be the time to perform one arithmetic operation, and F the total number of arithmetic operations, our overall performance model becomes $\alpha S + \beta W + \gamma F$. The same technological trends cited above tell us that $\alpha \gg \beta \gg \gamma$. This is why it is important to count messages S and words W separately, because either one may be the bottleneck. Later we will present lower bounds on both S and W, because it is of interest to have algorithms that minimize both bandwidth and latency costs.

On a sequential computer with a memory hierarchy, the model $\alpha S + \beta W + \gamma F$ is enough to model two levels of memory, say DRAM and cache. When there are multiple levels of memory, there is a cost associated with moving data between each adjacent pair of levels, so there will be an $\alpha S + \beta W$ term associated with each level.

On a parallel computer, $\alpha S + \beta W + \gamma F$ will initially refer to the communication and arithmetic done by one processor only. A lower bound for one processor is (sometimes) enough for a lower bound on the overall algorithm, but to upper-bound the time required by an entire algorithm requires us to sum these terms along the *critical path*, that is, a sequence of processors that must execute in a linear order (because of data dependences), and that also maximizes the sum of the costs. Note that there may be different critical paths for latency costs, bandwidth costs and arithmetic costs.

We note that this simple model may be naturally extended to other kinds of architectures. First, when the architecture can overlap communication and computation (*i.e.*, perform them in parallel), we see that $\alpha S + \beta W + \gamma F$ may be replaced by $\max(\alpha S + \beta W, \gamma F)$ or $\max(\alpha S, \beta W, \gamma F)$; this can lower the cost by at most a factor of 2 or 3, and so does not affect our asymptotic analysis. Second, on a *heterogeneous* parallel computer, that is, with different processors with different values of α, β, γ, memory sizes, *etc.*, one can still use $\alpha_i S_i + \beta_i W_i + \gamma_i F_i$ as the cost of processor i, and take the maximum over i or sum over critical paths to get lower and upper bounds. Third, on a parallel machine with local memory hierarchies (the usual case), one can include both kinds of costs.

We call an algorithm *communication-optimal* if it asymptotically attains communication lower bounds for a particular architecture (we sometimes allow an algorithm to exceed the lower bound by factors that are polylogarithmic in the problem size or machine parameters). More informally, we call an algorithm *communication-avoiding* if it is communication-optimal, or if it communicates significantly less than a conventional algorithm.

Finally, we note that this timing model may be quite simply converted to an energy model. First, interpret α_E, β_E and γ_E as joules per message, per word and per flop, respectively, instead of seconds per message, per word and per flop. The same technological trends as before tell us that $\alpha_E \gg \beta_E \gg \gamma_E$ and are all improving, but growing apart exponentially over time. Second, for each memory unit we add a term $\delta_E M$, where M is the number of words of memory used and δ_E is the joules per word per second to store data in that memory. Third, we add another term $\epsilon_E T$, where T is the run time and ϵ_E is the number of joules per second spent in 'leakage', cooling and other activities. Thus a (single) processor may be modelled as using $\alpha_E S + \beta_E W + \gamma_E F + \delta_E M + \epsilon_E T$ joules to solve a problem. Lower and upper bounds on S, W, and T translate to energy lower and upper bounds.

1.3. Summary of results for direct linear algebra

We first summarize previous lower bounds for direct linear algebra, and then the new ones. Hong and Kung (1981) considered any matmul algorithm that has the following properties.

(1) It requires the usual $2n^3$ multiplications and additions to multiply two $n \times n$ matrices $C = A \cdot B$ on a sequential machine with a two-level memory hierarchy.

(2) The large (but slow) memory level initially contains A and B, and also C at the end of the algorithm.

(3) The small (but fast) memory level contains only M words, where $M < 3n^2$, so it is too small to contain A, B and C simultaneously.

Then Hong and Kung (1981) showed that any such algorithm must move at least $W = \Omega(n^3/M^{1/2})$ words between fast and slow memory. This is attained by well-known 'blocking' algorithms that partition A, B and C into square sub-blocks of dimension $(M/3)^{1/2}$ or a little less, so that one sub-block each of A, B and C fit in fast memory, and that multiply sub-block by sub-block.

This was generalized to the parallel case by Irony, Toledo and Tiskin (2004). When each of the P processors stores the minimal $M = O(n^2/P)$ words of data and does an equal fraction $2n^3/P$ of the arithmetic, their lower bound is $W = \Omega(\#\text{flops}/M^{1/2}) = \Omega(n^3/P/(n^2/P)^{1/2}) = \Omega(n^3/P^{1/2})$, which

is attained by Cannon's algorithm (Cannon 1969) and SUMMA (van de Geijn and Watts 1997). The paper by Irony *et al.* (2004) also considers the so-called '3D' case, which does less communication by replicating the matrices $P^{1/3}$ times. This requires $M = n^2/P^{2/3}$ words of fast memory per processor. The lower bound becomes $W = \Omega(n^3/P/M^{1/2}) = \Omega(n^2/P^{2/3})$ and is a factor $P^{1/6}$ smaller than before, and it is attainable (Aggarwal, Chandra and Snir 1990, Johnsson 1992, Agarwal *et al.* 1995).

This was eventually generalized by Ballard, Demmel, Holtz and Schwartz (2011*d*) to *any* classical algorithm, that is, one that sufficiently resembles the three nested loops of matrix multiplication, to $W = \Omega(\#\text{flops}/M^{1/2})$ (this will be formalized below). This applies to (1) matmul, all the BLAS, Cholesky, LU decomposition, LDL^T factorization, algorithms that perform factorizations with orthogonal matrices (under certain technical conditions), and some graph-theoretic algorithms such as Floyd–Warshall (Floyd 1962, Warshall 1962), (2) dense or sparse matrices, where #flops may be much less than n^3, (3) some whole programs consisting of sequences of such operations, such as computing A^k by repeated matrix multiplication, no matter how the operations are interleaved, and (4) sequential, parallel and other architectures mentioned above. This lower bound applies for M larger than needed to store all the data once, up to a limit (Ballard *et al.* 2012*d*).

Furthermore, this lower bound on the bandwidth cost W yields a simple lower bound on the latency cost S. If the largest message allowed by the architecture is $m_{\max}$, then clearly $S \geq W/m_{\max}$. Since no message can be larger than than the memory, that is, $m_{\max} \leq M$, we get $S = \Omega(\#\text{flops}/M^{3/2})$. Combining these lower bounds on W and W with the number of arithmetic operations yields a lower bound on the overall run time, and this in turn yields a lower bound on the energy required to solve the problem (Demmel, Gearhart, Lipshitz and Schwartz 2013*b*).

Comparing these bounds to the costs of conventional algorithms, we see that they are frequently not attained, even for dense linear algebra. Many new algorithms have been invented that do attain these lower bounds, which will be summarized in Sections 3 (classical algorithms) and 5 (fast Strassen-like algorithms).

1.4. Summary of results for iterative linear algebra

Krylov subspace methods are widely used, such as GMRES (Saad and Schultz 1986) and conjugate gradient (CG) (Hestenes and Stiefel 1952) for linear systems, or Lanczos (Lanczos 1950) and Arnoldi (Arnoldi 1951) for eigenvalue problems. In their unpreconditioned variants, each iteration performs 1 (or a few) sparse matrix–vector multiplications (SpMV for short) with the input matrix A, as well as some dense linear algebra operations such as dot products. After s iterations, the resulting vectors span an $s+1$ dimensional Krylov subspace, and the 'best' solution (depending on the

algorithm) is chosen from this space. This means that the communication costs grow in proportion to s. In the sequential case, when A is too large to fit in the small fast memory, it is read from the large slow memory s times. If A has nnz nonzero entries that are stored in an explicit data structure, this means that $W \geq s \cdot \text{nnz}$. In the parallel case, with A and the vectors distributed across processors, communication is required at each iteration. Unless A has a simple block diagonal structure with separate blocks (and corresponding subvectors) assigned to separate processors, at least one message per processor is required for the SpMV. And if a dot product is needed, at least $\log P$ messages are needed to compute the sum. Altogether, this means $S \geq s \log P$.

To avoid communication, we make certain assumptions on the matrix A: it must be 'well-partitioned' in a sense to be made more formal in Section 7, but for now think of a matrix resulting from a mesh or other spatial discretization, partitioned into roughly equally large submeshes with as few edges as possible connecting one submesh to another. In other words, the partitioning should be load-balanced and have a low 'surface-to-volume ratio'.

In this case, it is possible to take s steps of many Krylov subspace methods for the communication cost of one step. In the sequential case this means $W = O(\text{nnz})$ instead of $s \cdot \text{nnz}$; clearly reading the matrix once from slow memory for a cost of $W = \text{nnz}$ is a lower bound. In the parallel case this means $S = O(\log p)$ instead of $O(s \log p)$; clearly the latency cost of one dot product is also a lower bound.

The idea behind these new algorithms originally appeared in the literature as *s-step* methods (see Section 8 for references). One first computed a different basis of the same Krylov subspace, for example using the *matrix-powers kernel* $[b, Ab, A^2b, \ldots, A^sb]$, and then reformulated the rest of the algorithm to compute the same 'best' solution in this subspace. The original motivation was exposing more parallelism, not avoiding communication, which requires different ways of implementing the matrix-powers kernel. But this research encountered a numerical stability obstacle: the matrix-powers kernel is basically running the power method, so that the vectors A^ib are becoming more nearly parallel to the dominant eigenvector, resulting in a very ill-conditioned basis and failure to converge. Later research partly alleviated this by using different polynomial bases $[b, p_1(A)b, p_2(A)b, \ldots, p_s(A)b]$ where $p_i(A)$ is a degree-i polynomial in A, chosen to make the vectors more linearly independent (*e.g.*, Philippe and Reichel 2012). But choosing a good polynomial basis (still a challenge to do automatically in general) was not enough to guarantee convergence in all cases, because two recurrences in the algorithm independently updating the approximate solution and residual could become 'decoupled', with the residual falsely indicating continued convergence of the approximate solution. This was finally

overcome by a generalization (Carson and Demmel 2014) of the residual replacement technique introduced by Van der Vorst and Ye (1999). Reliable and communication-avoiding s-step methods have now been developed for many Krylov methods, which offer large speed-ups in theory and practice, though many open problems remain. These will be discussed in Section 8.

There are two directions in which this work has been extended. First, many but not all sparse matrices are stored with explicit nonzero entries and their explicit indices: the nonzero entries may be implicit (for example, -1 on the offdiagonal of a graph Laplacian matrix), or the indices may be implicit (common in matrices arising in computer vision applications, where the locations of nonzeros are determined by the associated pixel locations), or both may be implicit, in which case the matrix is commonly called a *stencil*. In these cases intrinsically less communication is necessary. In particular, for stencils, only the vectors need to be communicated; we discuss this further in Section 7.

Second, one often uses *preconditioned* Krylov methods, *e.g.*, $MAx = Mb$ instead of $Ax = b$, where M somehow approximates A^{-1}. It is possible to derive corresponding s-step methods for many such methods; see, for instance, Hoemmen (2010). If M has a similar sparsity structure to A (or is even sparser), then previous communication-avoiding techniques may be used. But since A^{-1} is generically dense, M would also often be dense if written out explicitly, even if it is applied using sparse techniques (*e.g.*, solving sparse triangular systems arising from an incomplete factorization). This means that many common preconditioners cannot be used in a straightforward way. One class that can be used is that of *hierarchically semiseparable matrices*, which are represented by low-rank blocks; all of these extensions are discussed in Section 8.

1.5. Outline of the rest of the paper

The rest of this paper is organized as follows. The first half (Sections 2–5) is devoted to direct (mostly dense, some sparse) linear algebra, and the second (Sections 6–8) to iterative linear algebra (mostly for sparse matrices).

We begin in Sections 2 and 3 with communication costs of classical direct algorithms. We present lower bounds for classical computations in Section 2, starting with the basic case of classical matrix multiplication (§ 2.1), extensions using reductions (§ 2.2), generalization to three-nested-loops computations (§§ 2.3, 2.4), orthogonal transformations (§ 2.5), and further extensions and impact of the lower bounds (§ 2.6).

In Section 3 we discuss communication costs of classical algorithms, both conventional and communication-optimal. We summarize sequential (§ 3.1) and parallel ones (§ 3.2), then provide some details (§ 3.3) and point to remaining gaps and future work (§ 3.4).

In Sections 4 and 5 we discuss communication costs of fast (Strassen-like) linear algebra. We present lower bounds for Strassen-like computations in Section 4, starting with the expansion analysis of computation graphs (§§ 4.1, 4.2) applied to Strassen's matrix multiplication (§ 4.3), and Strassen-like multiplication (§ 4.4), and other algorithms (§ 4.5). In Section 5 we discuss communication-optimal Strassen-like algorithms that attain the lower bounds, both sequential (§§ 5.1, 5.2), and parallel (§§ 5.3, 5.4).

In Sections 6, 7 and 8 we discuss communication costs of iterative linear algebra. We begin with sparse matrix–vector multiplication (SpMV) in Section 6, starting with sequential lower bounds and optimal algorithms (§ 6.1), followed by parallel lower bounds and optimal algorithms (§ 6.2). The main conclusion of Section 6 is that any conventional Krylov subspace (or similar) method that performs a sequence of SpMVs will most likely be communication-bound. This motivates Section 7, which presents communication-avoiding Krylov basis computations, starting with lower bounds on communication costs (§ 7.1), then Akx algorithms (§ 7.2), and blocking covers (§ 7.3). We then point to related work and future research (§ 7.4).

Based on the kernels introduced in Section 7, in Section 8 we present communication-avoiding Krylov subspace methods for eigenvalue problems (§ 8.2), and for linear systems (§ 8.3). We demonstrate speed-ups (§ 8.4), and discuss numerical issues with finite precision (§ 8.5), and how to apply preconditioning in communication-avoiding ways (§ 8.6). We conclude with remaining gaps and future research (§ 8.7).

2. Lower bounds for classical computations

In this section we consider lower bounds for *classical* direct linear algebra computations. These computations can be specified by algorithms that are basically composed of three nested loops; see further details in Section 2.3. For some special cases, such as dense matrix multiplication, 'classical' means that the algorithm performs all n^3 scalar multiplications in the definition of $n \times n$ matrix multiplication, though the order in which the scalar multiplications are performed is arbitrary. We thus exclude from the discussion in this section, for example, Strassen's fast matrix multiplication (Strassen 1969). See Sections 4 and 5 for lower bounds and upper bounds of Strassen-like methods.

2.1. Matrix multiplication

Hong and Kung (1981) proved a lower bound on the bandwidth cost required to perform dense matrix multiplication in the sequential two-level memory model using a classical algorithm, where the input matrices are too large to fit in fast memory. They obtained the following result, using what they

called a 'red–blue pebble game' analysis of the computation graph of the algorithm.

Theorem 2.1 (Hong and Kung 1981, Corollary 6.2). For classical matrix multiplication of dense $m \times k$ and $k \times n$ matrices implemented on a machine with fast memory of size M, the number of words transferred between fast and slow memory is

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

This result was proved using a different technique by Irony *et al.* (2004) and generalized to the distributed-memory parallel case. They state the following parallel bandwidth cost lower bound using an argument based on the Loomis–Whitney inequality (Loomis and Whitney 1949), given as Lemma 2.5.

Theorem 2.2 (Irony *et al.* 2004, Theorem 3.1). For classical matrix multiplication of dense $m \times k$ and $k \times n$ matrices implemented on a distributed-memory machine with P processors, each with a local memory of size M, the number of words communicated by at least one processor is

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

In the case where $m = k = n$ and each processor stores the minimal $M = O(n^2/P)$ words of data, the lower bound on bandwidth cost becomes $\Omega(n^2/P^{1/2})$. The authors also consider the case where the local memory size is much larger, $M = \Theta(n^2/P^{2/3})$, in which case $O(P^{1/3})$ times as much memory is used (compared to the minimum possible) and less communication is necessary. In this case the bandwidth cost lower bound becomes $\Omega(n^2/P^{2/3})$. See Section 2.6.2 for a discussion of limits on reducing communication by using extra memory, and Section 3.3.1 for further algorithmic discussion on utilizing extra memory for matrix multiplication.

For simplicity we will assume real matrices throughout the rest of this section; all the results generalize to the complex case.

2.2. Extending lower bounds with reductions

It is natural to try to extend the lower bounds for matrix multiplication to other linear algebra operation by means of reductions. Given a lower bound for one algorithm, we can make a reduction argument to extend that bound to another algorithm. In our case, given the matrix multiplication bounds, if we can show how to perform matrix multiplication using another algorithm (assuming the transformation requires no extra communication in an asymptotic sense), then the same bound must apply to the other algorithm, under the same assumptions.

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

$$\begin{pmatrix} I & 0 & -B \\ A & I & 0 \\ 0 & 0 & I \end{pmatrix} = \begin{pmatrix} I & & \\ A & I & \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} I & 0 & -B \\ & I & A \cdot B \\ & & I \end{pmatrix}. \tag{2.1}$$

That is, given two input matrices A and B, we can compute $A \cdot B$ by constructing the matrix on the left-hand side of the identity above, performing an LU decomposition, and then extracting the $(2,3)$ block of the upper triangular output matrix. Thus, given an algorithm for LU decomposition that communicates less than the lower bound for multiplication, we have an algorithm for matrix multiplication that communicates less than the lower bound, a contradiction. Note that although the dimension of the LU decomposition is three times that of the original multiplication, the same communication bound holds in an asymptotic sense. This reduction appears in Grigori, Demmel and Xiang (2011) and is stated formally as follows.

Theorem 2.3. Given a fast/local memory of size M, the bandwidth cost lower bound for classical LU decomposition of a dense $n \times n$ matrix is

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

A similar identity to equation (2.1) holds for Cholesky decomposition:

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

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

However, the reduction is not as straightforward as in the case of LU because the matrix-multiplication-by-Cholesky algorithm would include the computation of $A \cdot A^T$, which requires as much communication as general matrix multiplication.[1] We show in Ballard, Demmel, Holtz and Schwartz (2010) how to change the computation so that we can avoid constructing the $I + A \cdot A^T$ term and still perform Cholesky decomposition to obtain the product $A \cdot B$.

While a reduction argument is possible for LU and Cholesky decomposition, it does not seem to work for more general linear algebra operations, which motivated the approach in the next section.

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

2.3. Three-nested-loops computations

In this section we address classical direct linear algebra numerical methods. Recall that by 'classical' we mean algorithms that compute results using the definitions of computations, performing $O(n^3)$ scalar multiplications for dense matrices. These computations can be specified by algorithms that are basically composed of three nested loops.

The key observation, and the basis for the arguments in this section, is that the proof technique of Irony *et al.* (2004) can be applied more generally than just to dense matrix multiplication. The geometric argument is based on the lattice of indices (i, j, k), which corresponds to the updates $C_{ij} = C_{ij} + A_{ik} \cdot B_{kj}$ in the innermost loop. However, the proof does not depend on, for example, the scalar operations being multiplication and addition, the matrices being dense, or the input and output matrices being distinct. The important property is the relationship among the indices (i, j, k), which allows for the embedding of the computation in three dimensions. These observations let us state and prove a more general set of theorems and corollaries that provide a lower bound on the number of words moved into or out of a fast or local memory of size M: for a large class of classical linear algebra algorithms,

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

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

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

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

- all-pairs shortest paths computation based on the Floyd–Warshall approach.

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

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

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

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

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

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

2.3.1. Lower bound argument

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

Definition 2.4 (3NL computation). A computation is considered to be three-nested-loops (3NL) if it includes computing, for all $(i, j) \in S_c$ with $S_{ij} \subseteq \{1, 2, \ldots, n\}$,

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

where

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

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

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

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

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

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

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

Now we illustrate the notation in Definition 2.4 for the case of sequential dense $n \times n$ matrix multiplication $C = C + A \cdot B$, where A, B and C are stored in column-major layout in slow memory. We take S_c as all pairs (i,j) with $1 \le i,j \le n$ with output entry C_{ij} stored in location $\mathfrak{c}(i,j) = \mathcal{C} + (i-1) + (j-1) \cdot n$, where $\mathcal{C}$ is some memory location. Input matrix entry A_{ik} is analogously stored at location $\mathfrak{a}(i,k) = \mathcal{A} + (i-1) + (k-1) \cdot n$ and B_{kj} is stored at location $\mathfrak{b}(k,j) = \mathcal{B} + (k-1) + (j-1) \cdot n$, where $\mathcal{A}$ and $\mathcal{B}$ are offsets chosen so that none of the matrices overlap. The set $S_{ij} = \{1, 2, \ldots, n\}$ for all (i,j). Operation g_{ijk} is scalar multiplication, and f_{ij} computes the sum of its n arguments. Thus, $G = n^3$. In the case of parallel matrix multiplication, a single processor will perform only a subset of the computation. In this case, for a given processor, the sizes of the sets S_c and S_{ij} may be smaller than n^2 and n, respectively, and G will become n^3/P if the computation is load-balanced.

Loomis and Whitney (1949) proved a geometrical result that provides a surface-to-volume relationship in general high-dimensional space. We need

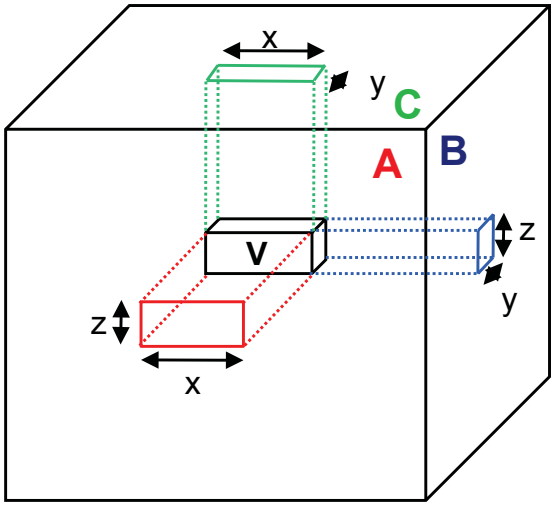


Figure 2.1. Intuition for Lemma 2.5: the volume of a box is equal to the square root of the product of the areas of its projections.

only the simplest version of their result, which will prove instrumental in our lower bound arguments.

Lemma 2.5 (Loomis and Whitney 1949). Let V be a finite set of lattice points in $\mathbb{R}^3$, that is, points (i, j, k) with integer coordinates. Let V_i be the projection of V in the i-direction, that is, all points (j, k) such that there exists an x so that $(x, j, k) \in V$. Define V_j and V_k similarly. Let $|\cdot|$ denote the cardinality of a set. Then $|V| \leq \sqrt{|V_i| \times |V_j| \times |V_k|}$.

An intuition for the correctness of this lemma is as follows. Think of a box of dimensions $x \times y \times z$. Then its (rectangular) projections on the three planes have areas $x \cdot y$, $y \cdot z$ and $x \cdot z$, and we have that its volume $x \cdot y \cdot z$ is equal to the square root of the product of the three areas. See Figure 2.1 for a graphical representation of this idea. In this instance equality is achieved; only the inequality applies in general.

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

Theorem 2.6 (Ballard *et al.* 2011*d*). The bandwidth cost lower bound of a 3NL computation (Definition 2.4) is

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

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

Proof. Following Irony *et al.* (2004), we consider any implementation of the computation as a stream of instructions involving computations and

memory operations: loads and stores from and to slow/global memory. The argument is as follows.

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

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

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

Now we compute the upper bound F using the geometric result of Loomis and Whitney (1949), a simpler form of which is given as Lemma 2.5. Let the set of lattice points (i,j,k) represent each function evaluation

$$g_{ijk}(\mathrm{Mem}(\mathfrak{a}(i,k)), \mathrm{Mem}(\mathfrak{b}(k,j))).$$

For a given segment, let V be the set of indices (i,j,k) of the g_{ijk} operations

performed during the segment, let V_k be the set of indices (i,j) of their destinations $\mathfrak{c}(i,j)$, let V_j be the set of indices (i,k) of their arguments $\mathfrak{a}(i,k)$, and let V_i be the set of indices (j,k) of their arguments $\mathfrak{b}(j,k)$. Then by Lemma 2.5,

$$|V| \le \sqrt{|V_i| \cdot |V_j| \cdot |V_k|} \le \sqrt{(4M)^3} \equiv F.$$

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

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

Note that the proof of Theorem 2.6 applies to any ordering of the g_{ijk} operations. In the case of matrix multiplication, there are no dependences between g_{ijk} operations, so every ordering will compute the correct answer. However, for most other computations, there are many dependences that must be respected for correct computation. This lower bound argument thus applies not only to correct reorderings of the algorithm but also to incorrect ones, as long as the computation satisfies the conditions of Definition 2.4.

In the parallel case, since some processor must compute at least $1/P$th of the 3NL operations, we can lower bound the number of words that processor communicates and achieve the following corollary.

Corollary 2.7 (Ballard *et al.* 2011*d*). The bandwidth cost lower bound of a 3NL computation (Definition 2.4) on a parallel machine is

$$W \ge \frac{G}{8P\sqrt{M}} - M,$$

where G is the total number of 3NL operations (performed globally), P is the number of processors, and M is the size of the local memory.

Note that Corollary 2.7 may not be the tightest lower bound for a given work distribution of G 3NL operations to processors. For example, if G_i is the number of 3NL operations performed by processor i (with $\sum_i G_i = G$), then the tightest lower bound is given by $\max_i G_i/(8\sqrt{M}) - M$. In the following section we will state corollaries of Theorem 2.6 in terms of G, the number of 3NL operations performed locally by a given processor; that is, the corollaries can be applied in either the sequential or parallel case.

In the case of dense matrix multiplication and other computations where $G = \Theta(n^3)$, we can state simplified versions of Theorem 2.6. In particular, we can write the lower bounds for dense 3NL computations in a way that is easily comparable with Strassen-like and other fast computations (see Section 4). In the sequential case, assuming $n \gg \sqrt{M}$, the lower bound

becomes

$$W = \Omega\left(\left(\frac{n}{\sqrt{M}}\right)^3 \cdot M\right). \tag{2.2}$$

In the parallel case, assuming $n \gg M^{1/2}P^{1/3}$, the lower bound becomes

$$W = \Omega\left(\left(\frac{n}{\sqrt{M}}\right)^3 \cdot \frac{M}{P}\right). \tag{2.3}$$

2.3.2. Applications of the lower bound

We now show how Theorem 2.6 applies to a variety of classical computations for numerical linear algebra.

2.3.2.1. BLAS. We begin with matrix multiplication, on which we base Definition 2.4. The proof is implicit in the illustration of the definition with matrix multiplication in Section 2.3.1.

Corollary 2.8. The bandwidth cost lower bound for classical (dense or sparse) matrix multiplication is $G/(8\sqrt{M}) - M$, where G is the number of scalar multiplications performed. In the special case of sequentially multiplying a dense $m \times k$ matrix times a dense $k \times n$ matrix, this lower bound is $mkn/(8\sqrt{M}) - M$.

This reproduces Theorem 2.2 from Irony *et al.* (2004) (with a different constant) for the case of two distinct, dense matrices, though we need no such assumptions. We note that this result could have been stated for sparse A and B by Hong and Kung (1981): combine their Theorem 6.1 (their $\Omega(|V|)$ is the number of scalar multiplications) with their Lemma 6.1 (whose proof does not require A and B to be dense). For algorithms attaining this bound in the dense case, see Section 3.3.1. For further discussion of this bound in the sparse case, see Ballard *et al.* (2013*c*).

We next extend Theorem 2.6 beyond matrix multiplication. The simplest extension is to the so-called level-3 BLAS (Basic Linear Algebra Subroutines: Blackford *et al.* 2002), which include related operations such as multiplication by (conjugate) transposed matrices, by triangular matrices and by symmetric (or Hermitian) matrices. Corollary 2.8 applies to these operations without change (in the case of $A^T \cdot A$ we use the fact that Theorem 2.6 makes no assumptions about the matrices being multiplied not overlapping).

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

upper triangular) is specified by

$$C_{ij} = \left(B_{ij} - \sum_{k=i+1}^{n} A_{ik} \cdot C_{kj} \right) \Big/ A_{ii}, \tag{2.4}$$

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

Corollary 2.9. The bandwidth cost lower bound for classical (dense or sparse) TRSM is $G/(8\sqrt{M})-M$, where G is the number of scalar multiplications performed. In the special case of sequentially solving a dense triangular $n \times n$ system with m right-hand sides, this lower bound is $\Omega(mn^2/\sqrt{M})$.

Proof. We need only verify that TRSM is a 3NL computation. We let f_{ij} be the function defined in equation (2.4) (or similarly for lower triangular matrices or other variants). Then we make the correspondences that C_{ij} is stored at location $\mathfrak{c}(i,j) = \mathfrak{b}(i,j)$, A_{ik} is stored at location $\mathfrak{a}(i,k)$, and g_{ijk} multiplies $A_{ik} \cdot C_{kj}$. Since A is an input stored in slow/global memory and C is the output of the operation and must be written to slow/global memory, the mappings $\mathfrak{a}$, $\mathfrak{b}$, and $\mathfrak{c}$ are all one-to-one into slow/global memory. Note that $\mathfrak{c} = \mathfrak{b}$ does not prevent the computation from being 3NL. Further, functions f_{ij} (involving a summation of g_{ijk} outputs) and g_{ijk} (scalar multiplication) depend nontrivially on their arguments. Thus, the computation is 3NL.

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

See Section 3.3.1 for discussions of algorithms attaining this bound for dense matrices.

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

We note that Theorem 2.6 also applies to the level-2 BLAS (*e.g.*, matrix–vector multiplication) and level-1 BLAS (*e.g.*, dot products), though the lower bound is not attainable. In those cases, the number of words required to access each of the input entries once already exceeds the lower bound of Theorem 2.6.

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

$$\begin{aligned} L_{ij} &= \left(A_{ij} - \sum_{k<j} L_{ik} \cdot U_{kj}\right) \Big/ U_{jj}, \qquad \text{for } i > j, \\ U_{ij} &= A_{ij} - \sum_{k<i} L_{ik} \cdot U_{kj}, \qquad \text{for } i \leq j. \end{aligned} \tag{2.5}$$

In the sparse case, the equations may be evaluated for some subset of indices (i, j) and the summations may be over some subset of the indices k. Equation (2.5) also assumes pivoting has already been incorporated in the interpretation of the indices i, j, and k. Note that since the set of input and output operands overlap, there are data dependences which must be respected for correct computation.

Corollary 2.10. The bandwidth cost lower bound for classical (dense or sparse) LU factorization is $G/(8\sqrt{M}) - M$, where G is the number of scalar multiplications performed. In the special case of sequentially factoring a dense $m \times n$ matrix with $m \geq n$, this lower bound is $\Omega(mn^2/\sqrt{M})$.

We omit the proof as it is similar to that of Corollary 2.9: see Ballard (2013, §4.1.2.2) for full details. Note that Corollary 2.10 reproduces the result from the reduction argument in Section 2.2 for dense and square factorization. However, this corollary is a strict generalization, as it also applies to sparse and rectangular factorizations. For a discussion of algorithms attaining this bound for dense matrices, see Section 3.3.3.

Consider incomplete LU (ILU) factorization (Saad 2003), where some entries of L and U are omitted in order to speed up the computation. In the case of *level-based* incomplete factorizations (*i.e.*, ILU(p)), Corollary 2.10 applies with G corresponding to the scalar multiplications performed. However, consider *threshold-based* ILU, which computes a possible nonzero entry L_{ij} or U_{ij} and compares it to a threshold, storing it only if it is larger than the threshold and discarding it otherwise. Does Corollary 2.10 apply to this computation?

Because a computed entry L_{ij} may be discarded, the assumption that f_{ij} depends nontrivially on its arguments is violated. However, if we restrict the count of scalar multiplications to the subset of S_c for which output entries are *not* discarded, then all the assumptions of 3NL are met, and the lower bound applies (with G computed based on the subset). This count may underestimate the computation by more than a constant factor (if nearly all computed values fall beneath the threshold), but the lower bound will be valid nonetheless. We consider another technique to arrive at a lower bound for threshold-based incomplete factorizations in Section 2.4.2.2.

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

$$\begin{aligned} L_{jj} &= \left(A_{jj} - \sum_{k<j} L_{jk}^2 \right)^{1/2}, \\ L_{ij} &= \left(A_{ij} - \sum_{k<j} L_{ik} \cdot L_{jk} \right) \Big/ L_{jj}, \quad \text{for } i > j. \end{aligned} \tag{2.6}$$

In the sparse case, the equations may be evaluated for some subset of indices (i, j) and the summations may be over some subset of the indices k. Equation (2.6) also assumes pivoting has already been incorporated in the interpretation of the indices i, j, and k. As in the case of LU factorization, there are data dependences which must be respected for correct computation.

Corollary 2.11. The bandwidth cost lower bound for classical (dense or sparse) Cholesky factorization is $G/(8\sqrt{M}) - M$, where G is the number of scalar multiplications performed. In the special case of sequentially factoring a dense $n \times n$ matrix, this lower bound is $\Omega(n^3/\sqrt{M})$.

We omit the proof as it is similar to that of Corollary 2.9: see Ballard (2013, §4.1.2.3) for full details. Note that Corollary 2.11 reproduces the result from the reduction argument in Section 2.2 for dense factorization. However, this corollary is a strict generalization, as it also applies to sparse factorizations. For algorithms attaining this bound in the dense case, see Section 3.3.2. As in the case of LU (Section 2.3.2.2), Corollary 2.11 is general enough to accommodate incomplete Cholesky (IC) factorizations (Saad 2003).

We now consider Cholesky factorization on a particular class of sparse matrices for which computational lower bounds are known. Since these computational bounds apply to G, Corollary 2.11 leads directly to a concrete communication lower bound. Hoffman, Martin and Rose (1973) and George (1973) prove that a lower bound on the number of multiplications required to compute the sparse Cholesky factorization of an $n^2 \times n^2$ matrix representing a five-point stencil on a 2D grid of n^2 nodes is $\Omega(n^3)$. This lower bound applies to any matrix containing the structure of the five-point stencil. This yields the following.

Corollary 2.12. In the case of the sparse Cholesky factorization on a sequential machine of a matrix which includes the sparsity structure of the matrix representing a five-point stencil on a two-dimensional grid of n^2 nodes, the bandwidth cost lower bound is $\Omega(n^3/\sqrt{M})$.

George (1973) shows that this arithmetic lower bound is attainable with a nested dissection algorithm in the case of the five-point stencil. Gilbert

and Tarjan (1987) show that the upper bound also applies to a larger class of structured matrices, including matrices associated with planar graphs. Recently, Grigori, David, Demmel and Peyronnet (2010) have obtained new algorithms for sparse cases of Cholesky decomposition that are proved to be communication-optimal using this lower bound.

2.3.2.4. Computing eigenvectors from Schur form. The Schur decomposition of a matrix A is the decomposition $A = QTQ^T$, where Q is unitary and T is upper triangular. Note that in the real-valued case, Q is orthogonal and T is quasi-triangular. The eigenvalues of a triangular matrix are given by the diagonal entries. Assuming all the eigenvalues are distinct, we can solve the equation $TX = XD$ for the upper triangular eigenvector matrix X, where D is a diagonal matrix whose entries are the diagonal of T. This implies that for $i < j$,

$$X_{ij} = \left(T_{ij}X_{jj} + \sum_{k=i+1}^{j-1} T_{ik}X_{kj}\right) \Big/ \left(T_{jj} - T_{ii}\right) \tag{2.7}$$

where $X_{jj} \neq 0$ can be arbitrarily chosen for each j. Note that in the sparse case, the equations may be evaluated for some subset of indices (i, j) and the summations may be over some subset of the indices k. After computing X, the eigenvectors of A are given by QX.

Corollary 2.13. The bandwidth cost lower bound for computing the eigenvectors of a (dense or sparse) triangular matrix with distinct eigenvalues is $G/(8\sqrt{M}) - M$, where G is the number of scalar multiplications performed. In the special case of a dense triangular $n \times n$ matrix on a sequential machine, this lower bound is $\Omega(n^3/\sqrt{M})$.

We omit the proof as it is similar to that of Corollary 2.9: see Ballard (2013, §4.1.2.4) for full details.

2.3.2.5. Floyd–Warshall all-pairs shortest paths. Theorem 2.6 applies to more general computations than strictly linear algebraic ones, where g_{ijk} are scalar multiplications and f_{ij} are based on summations. We consider the Floyd–Warshall method (Floyd 1962, Warshall 1962) for computing the shortest paths between all pairs of vertices in a graph. If we define $d_{ij}^{(k)}$ to be the shortest distance between vertex i and vertex j using the first k vertices, then executing the following computation for all k, i, and j determines in $D_{ij}^{(n)}$ the shortest path between vertex i and vertex j using the entire graph:

$$D_{ij}^{(k)} = \min\left(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}\right). \tag{2.8}$$

Taking care to respect the data dependences, the computation can be done in place, with $D^{(0)}$ being the original adjacency graph and each $D^{(k)}$ overwriting $D^{(k-1)}$. The original formulation of the algorithm consists of three

nested loops with k as the outermost loop index, but there are many other orderings which maintain correctness.

Corollary 2.14. The bandwidth cost lower bound for computing all-pairs shortest paths using the Floyd–Warshall method is $G/(8\sqrt{M}) - M$, where G is the number of scalar additions performed. In the special case of sequentially computing all-pairs shortest paths on a dense graph with n vertices, this lower bound is $\Omega(n^3/\sqrt{M})$.

We omit the proof as it is similar to that of Corollary 2.9: see Ballard (2013, §4.1.2.5) for full details. This result is also claimed (without proof) as Lemma 1 of Park, Penner and Prasanna (2004); the authors also provide a sequential algorithm attaining the bound. For a parallel algorithm attaining this bound, see Section 3.3.8 and further details in Solomonik, Buluç and Demmel (2013).

2.4. Three-nested-loop computation with temporary operands

2.4.1. Lower bound argument

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

Consider computing the Frobenius norm of a product of matrices:

$$\|A \cdot B\|_F^2 = \sum_{ij} (A \cdot B)_{ij}^2.$$

If we define f_{ij} as the square of the dot product of row i of A and column j of B, the output of f_{ij} does not necessarily map to a location in slow memory because entries of the product $A \cdot B$ are only temporary values, not outputs of the computation (the norm is the only output). However, in order to compute the norm correctly, every entry of $A \cdot B$ must be computed, so the matrix might as well be an output of the computation (in which case Theorem 2.6 would apply). In the proof of Theorem 2.15 below, we show using a technique of *imposing writes* that the amount of communication required for computations with temporary values like these is asymptotically the same as those forced to output the temporary values.

While the example above illustrates temporary output operands of f_{ij} functions, a computation may also have temporary input operands to g_{ijk} functions. For example, if we want to compute the Frobenius norm of a product of matrices where the entries of the input matrices are given by formulas (*e.g.*, $A_{ij} = i^2 + j$), then the computation may require very little communication since the entries can be recomputed on the fly as needed.

However, if we require that each temporary input operand be computed only once, and we map each operand to a location in slow/global memory, then if the operand has already been computed and does not reside in fast memory, it must be read from slow/global memory. The assumption that each temporary operand be computed only once seems overly restrictive in the case of matrix entries given by simple formulas of their indices, but for many other common computations (see Section 2.4.2), the temporary values are more expensive to compute and recomputation on the fly is more difficult.

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

We now state the result more formally.

Theorem 2.15. Suppose a computation is 3NL except that some of its operands (*i.e.*, inputs to g_{ijk} operations or outputs of f_{ij} functions) are temporary and are not necessarily mapped to slow/global memory. Then if the number of temporary operands is t, and if each (input or output) temporary operand is computed exactly once, then the bandwidth cost lower bound is given by

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

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

Proof. Let $\mathcal{C}$ be such a computation, and let $\mathcal{C}'$ be the same computation with the exception that for each of the three types of operands (defined by mappings $\mathfrak{a}$, $\mathfrak{b}$, and $\mathfrak{c}$), every temporary operand is mapped to a distinct

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

location in slow/global memory and must be written to that location by the end of the computation. This enforces that the mappings $\mathfrak{a}$, $\mathfrak{b}$, and $\mathfrak{c}$ are all one-to-one into slow/global memory, and so $\mathcal{C}'$ is a 3NL computation and Theorem 2.6 applies. Consider an algorithm that correctly performs computation $\mathcal{C}$. Modify the algorithm by imposing writes: every time a temporary operand is computed, we impose a write to the corresponding location in slow/global memory (a copy of the value may remain in fast memory). After this modification, the algorithm will correctly perform the computation $\mathcal{C}'$. Thus, since every temporary operand is computed exactly once, the bandwidth cost of the algorithm differs by at most t words from an algorithm to which the lower bound $W \geq G/(8\sqrt{M}) - M$ applies, and the result follows. □

2.4.2. Applications of the lower bound

2.4.2.1. Solving the normal equations. In Section 2.3.2.1 we proved a communication lower bound for the Cholesky–QR computation by applying Theorem 2.6 to the TRSM required to compute the orthogonal matrix Q. In some related situations, such as solving the normal equations $A^TAx = A^Tb$ (by forming A^TA and performing a Cholesky decomposition), the computation does not form Q explicitly. Here, we apply Theorem 2.15 to the computation A^TA, the output of which is a temporary matrix.

Corollary 2.16. The bandwidth cost lower bound for solving the normal equations to find the solution to a least-squares problem with matrix A is $G/(8\sqrt{M}) - M - t$, where G is the number of scalar multiplications involved in the computation of A^TA, t is the number of nonzeros in A^TA, and we assume the entries of A^TA are computed only once. In the special case of a dense $m \times n$ matrix A with $m \geq n$ on a sequential machine, this lower bound is $\Omega(mn^2/\sqrt{M})$.

Proof. As argued in Section 2.3.2.1, computing A^TA is a 3NL computation (we ignore the Cholesky factorization and triangular solves in this proof). That is, $\mathfrak{a}(i,j) = \mathfrak{b}(j,i)$ and f_{ij} is the summation function defined for either the lower ($i \geq j$) or upper ($i \leq j$) triangle because the output is symmetric. Since A is an input to the normal equations, it must be stored in slow memory. However, the output of A^TA need not be stored in slow memory (its Cholesky factor will be used to solve for the final output of the computation). Thus, the number of temporary operands is the number of nonzeros in the output of A^TA, which are all outputs of f_{ij} functions. In the case of a dense $m \times n$ matrix A with $m \geq n$, the output A^TA is $n \times n$ (symmetric, so $n^2/2$ terms). When $m, n \geq \sqrt{M}$, the $mn^2/\sqrt{M}$ term asymptotically dominates the (negative) M and $n^2/2$ terms. □

Note that if A is sparse, R may be denser, in which case a stronger lower bound can be derived from the Cholesky part.

2.4.2.2. Incomplete factorizations. Section 2.3.2.2 considered lower bounds for threshold-based incomplete LU factorizations. Because Theorem 2.6 requires that the f_{ij} functions depend nontrivially on their arguments, we must ignore all scalar multiplications that lead to discarded outputs (due to their values falling below the threshold). However, because the output values must be fully computed before comparing them to the threshold value, we may be ignoring a significant amount of the computation. Using Theorem 2.15, we can state another (possibly tighter) lower bound which counts all the scalar multiplications performed by imposing reads and writes on the discarded values.

Corollary 2.17. Consider a threshold-based incomplete LU or Cholesky factorization, and let t be the number of output values discarded due to thresholding. Assuming each discarded value is computed exactly once, the bandwidth cost lower bound for the computation is $G/(8\sqrt{M}) - M - t$, where G is the number of scalar multiplications.

We omit the proof: see Ballard (2013, §4.2.2.2) for full details.

2.4.2.3. LDL^T factorization. For the (symmetric) factorization of symmetric indefinite matrices, symmetric pivoting is required for numerical stability. Independent of sparsity and pivot order, the classical Bunch–Kaufman LDL^T factorization (Bunch and Kaufman 1977), where L is unit lower triangular and D is block diagonal with 1×1 and 2×2 blocks, is specified in the case of positive or negative definite matrices (*i.e.*, all diagonal blocks of D are 1×1) by

$$\begin{aligned} D_{jj} &= A_{jj} - \sum_{k<j} L_{jk}^2 D_{kk}, \\ L_{ij} &= \left(A_{ij} - \sum_{k<j} L_{ik} \cdot (D_{kk} L_{jk}) \right) \Big/ D_{jj}, \qquad \text{for } i > j. \end{aligned} \tag{2.9}$$

In the sparse case, the equations may be evaluated for some subset of indices (i, j) and the summations may be over some subset of the indices k. Equation (2.9) also assumes pivoting has already been incorporated in the interpretation of the indices i, j and k.

Note that in this case, the operand $(D_{kk}L_{jk})$ is a temporary operand. This complication is overlooked in an earlier paper (Ballard *et al.* 2011*d*), where it is claimed that the argument for Cholesky also applies to LDL^T factorization. In the terminology used here, it is assumed that LDL^T is 3NL. Here, we obtain the lower bound as a corollary of Theorem 2.15.

To specify the more general computation where D includes both 1×1 and 2×2 blocks, we define the matrix $W = DL^T$ and let S be the set of

rows/columns corresponding to a 1×1 block of D. Then, for $j \in S$, the computation of column j of L can be written similarly to equation (2.9):

$$L_{ij} = \left(A_{ij} - \sum_{k<j} L_{ik} \cdot W_{kj} \right) \Big/ D_{jj}, \quad \text{for } i > j. \tag{2.10}$$

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

$$L_{i,j:j+1} = \left(A_{i,j:j+1} - \sum_{k<j} L_{i,k} \cdot W_{k,j:j+1} \right) D^{-1}_{j:j+1,j:j+1}, \quad \text{for } i > j+1. \tag{2.11}$$

Corollary 2.18. The bandwidth cost lower bound for classical (dense or sparse) LDL^T factorization is $G/(\sqrt{8M}) - M - t$, where G is the number of scalar multiplications performed in computing L, t is the number of nonzeros in the matrix DL^T, and we assume the entries of DL^T are computed only once. In the special case of sequentially factoring a dense $n \times n$ matrix, this lower bound is $\Omega(n^3/\sqrt{M})$.

We omit the proof: see Ballard (2013, §4.2.2.3) for full details. See Section 3.3.4 for a discussion of algorithms for this computation. No known sequential algorithm attains this bound for all matrix dimensions and performs a numerically stable pivoting scheme.

2.4.2.4. LTL^T factorization. This symmetric indefinite factorization computes a lower triangular matrix L and a symmetric tridiagonal matrix T such that $A = LTL^T$. Symmetric pivoting is required for numerical stability. Parlett and Reid (1970) developed an algorithm for computing this factorization requiring approximately $(2/3)n^3$ flops, the same cost as LU factorization and twice the computational cost of Cholesky. Aasen (1971) improved the algorithm and reduced the computational cost to $(1/3)n^3$, making use of a temporary upper Hessenberg matrix $H = TL^T$. Aasen's algorithm works by alternately solving for unknown values in the equations $A = LH$ and $H = TL^T$. Because the matrix H is integral to the computation but is a temporary matrix, we use Theorem 2.15 to obtain a communication lower bound.

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

we specify a classical LTL^T computation in terms of the lower triangular matrix L:

$$L_{ij} = \left(A_{i,j-b} - \sum_{k=b+1}^{j-b} L_{ik} H_{k,j-b} \right) \Big/ H_{j,j-b}, \quad \text{for } b < j < i \leq n. \tag{2.12}$$

In the sparse case, the equations may be evaluated for some subset of indices (i, j) and the summations may be over some subset of the indices k. Equation (2.12) also assumes pivoting has already been incorporated in the interpretation of the indices i, j, and k.

Corollary 2.19. The bandwidth cost lower bound for classical (dense or sparse) LTL^T factorization is $G/(8\sqrt{M}) - M - t$, where G is the number of scalar multiplications performed in computing L, t is the number of nonzeros in the matrix TL^T, and we assume the entries of TL^T are computed only once. In the special case of sequentially factoring a dense $n \times n$ matrix (with T having bandwidth $b \ll n$), this lower bound is $\Omega(n^3/\sqrt{M})$.

We omit the proof: see Ballard (2013, §4.2.2.4) for full details. This bound is attainable in the sequential case by the algorithm presented in Ballard *et al.* (2013*c*). See Section 3.3.4 for further discussion of symmetric indefinite algorithms.

2.4.2.5. Gram–Schmidt orthogonalization. We consider the Gram–Schmidt process (both classical and modified versions) for orthogonalization of a set of vectors (see, *e.g.*, Algorithm 3.1 of Demmel 1997). Given a set of vectors stored as columns of an $m \times n$ matrix A, the Gram–Schmidt process computes a QR decomposition, though the R matrix is sometimes not considered part of the output. For generality, we assume the R matrix is not a final output and use Theorem 2.15. Letting the columns of Q be the computed orthonormal basis, we specify the Gram–Schmidt computation in terms of the equation for computing entries of R. In the case of Classical Gram–Schmidt, we have

$$R_{ij} = \sum_{k=1}^{m} Q_{ki} A_{kj}, \tag{2.13}$$

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

$$R_{ij} = \sum_{k=1}^{m} Q_{ki} Q_{kj}, \tag{2.14}$$

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

Corollary 2.20. The bandwidth cost lower bound for (dense or sparse) QR factorization using Classical or Modified Gram–Schmidt orthogonalization is $G/(8\sqrt{M}) - M - t$, where G is the number of scalar multiplications performed in computing R, t is the number of nonzeros in R, and we assume the entries of R are computed only once. In the special case of orthogonalizing a dense $m \times n$ matrix with $m \geq n$ on a sequential machine, this lower bound is $\Omega(mn^2/\sqrt{M})$.

We omit the proof: see Ballard (2013, §4.2.2.5) for full details.

2.5. Applying orthogonal transformations

The most stable and highest-performing algorithms for QR decomposition are based on applying orthogonal transformations. Two-sided orthogonal transformations are used in the reduction step of the most commonly used approaches for solving eigenvalue and singular value problems: transforming a matrix to Hessenberg form for the nonsymmetric eigenproblem, tridiagonal form for the symmetric eigenproblem, and bidiagonal form for the SVD. In this section, we state two lower bounds, first in the context of one-sided orthogonal transformations (as in QR decomposition), and then generalized for two-sided transformations. Each of the lower bounds requires certain assumptions on the algorithms. We discuss in Section 2.5.3 to which algorithms each of the lower bounds apply.

The case of applying orthogonal transformations is more subtle to analyse for several reasons: (1) there is more than one way to represent the orthogonal factor (*e.g.*, Householder reflections and Givens rotations), (2) the standard ways to reorganize or 'block' transformations to reduce communication involve using the distributive law, not just summing terms in a different order (Bischof and Van Loan 1987, Schreiber and Van Loan 1989, Puglisi 1992), and (3) there may be many temporary operands that are not mapped to slow/global memory.

2.5.1. One-sided orthogonal transformations

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

Furthermore, we model the standard way of blocking Householder vectors, writing

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

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

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

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

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

$$A_{ij} = A_{ij} - \sum_{k=1}^{h} U_{ik} Z_{kj}, \tag{2.15}$$

where z_k (the kth row of Z) is a temporary quantity computed from A, u_k, and possibly other columns of U, depending on how Householder vectors are blocked. If A is $m \times n$, then U is $m \times h$ and Z is $h \times n$. Note that in the case of QR decomposition, it may be that $h \gg n$ (if one Householder vector is used to annihilate one entry below the diagonal, for example). The equations may be evaluated for some subset of indices (i, j) and the summations are over some subset of the indices k (even in the dense case). Equation (2.15) also assumes pivoting has already been incorporated in the interpretation of the indices i and j.

We state here two lower bounds that apply to separate sets of algorithms involving one-sided orthogonal transformations. The first lower bound assumes only that the number of Householder vectors per column is a constant (independent of the number of rows). In this case, because the computation is 3NL with temporary operands, we can apply Theorem 2.15 and bound the number of temporary operands. We state this result without proof: see Ballard (2013, §4.3.1) for full details.

Corollary 2.21. The bandwidth cost lower bound for applying orthogonal updates as specified in equation (2.15) is $G/(8\sqrt{M}) - M - t$, where G is the number of scalar multiplications involved in the computation of UZ and t is the number of nonzeros in Z. In the special case of sequentially computing

a QR decomposition of an $m \times n$ matrix using only a constant number of Householder vectors per column, this lower bound is $\Omega(mn^2/\sqrt{M})$.

Many efficient algorithms do not satisfy the assumptions of Corollary 2.21, but a second, more involved lower bound argument applies to algorithms that use many Householder vectors per column. However, two other assumptions are necessary for the lower bound to apply. First, we assume that the algorithm does not block Householder updates (*i.e.*, all T matrices are 1×1). Second, we assume the algorithm makes 'forward progress'. Informally, forward progress means that an entry which is deliberately zeroed out is not filled in by a later transformation: see Ballard *et al.* (2011*d*, Definition 4.3) or Ballard (2013, Definition 4.18) for a formal definition. Again, we omit the proof: see Ballard *et al.* (2011*d*) or Ballard (2013) for full details.

Theorem 2.22. An algorithm that applies orthogonal transformations to annihilate matrix entries, does not compute T matrices of dimension two or greater for blocked updates, maintains forward progress, and performs G flops of the form $U \cdot Z$ (as defined in equation (2.15)) has a bandwidth cost of at least $\Omega(G/\sqrt{M}) - M$ words. In the special case of a dense $m \times n$ matrix with $m \geq n$ on a sequential machine, this lower bound is $\Omega(mn^2/\sqrt{M})$.

2.5.2. Two-sided orthogonal transformations

Standard algorithms for computing eigenvalues and eigenvectors, or singular values and singular vectors (the SVD), start by applying orthogonal transformations to both sides of A to reduce it to a 'condensed form' (Hessenberg, tridiagonal or bidiagonal) with the same eigenvalues or singular values, and simply related eigenvectors or singular vectors (Demmel 1997). We can extend our argument for one-sided orthogonal transformations to these computations. We can have some arbitrary interleaving of (block) Householder transformations applied on the left,

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

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

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

where we define $Z_R = A \cdot U_R \cdot T_R$. Combining these, we can index the computation by Householder vector number similarly to equation (2.15):

$$A(i,j) = A(i,j) - \sum_{k_L} U_L(i,k_L) \cdot Z_L(k_L,j) - \sum_{k_R} Z_R(i,k_R) \cdot U_R(j,k_R). \quad (2.16)$$

Of course, many possible dependences are ignored here, much as when we stated a similar formula for one-sided transformations. At this point we can apply either of the two lower bound arguments from before: we can

either assume that (1) the number of Householder vectors is small, so that the number of temporary Z_L and Z_R values are bounded, applying Theorem 2.15, or (2) all T matrices are 1×1, and we make 'forward progress', using the argument in the proof of Theorem 2.22. In case (1) we obtain a similar result to Corollary 2.21, as follows.

Corollary 2.23. The bandwidth cost lower bound for applying two-sided orthogonal updates is $G/(8\sqrt{M}) - M - t$, where G is the number of scalar multiplications involved in the computation of $U_L Z_L$ and $Z_R U_R$ (as specified in equation (2.16)) and t is the number of nonzeros in Z_L and Z_R. In the special case of sequentially reducing an $m \times n$ matrix to bidiagonal form using only a constant number of Householder vectors per row and column, this lower bound is $\Omega(mn^2/\sqrt{M})$. In the special case of reducing an $n \times n$ matrix to tridiagonal or Hessenberg form using only a constant number of Householder vectors per column, this lower bound is $\Omega(n^3/\sqrt{M})$.

In case (2), we have a similar result to Theorem 2.22.

Theorem 2.24. An algorithm that applies two-sided orthogonal transformations to annihilate matrix entries, does not compute T matrices of dimension two or greater for blocked updates, maintains forward progress, and performs G flops of the form $U \cdot Z$ (as defined in equation (2.16)) has a bandwidth cost of at least $\Omega(G/\sqrt{M}) - M$ words. In the special case of sequentially reducing a dense $m \times n$ matrix to bidiagonal form with $m \geq n$, this lower bound is $\Omega(mn^2/\sqrt{M})$. In the special case of reducing an $n \times n$ matrix to tridiagonal or Hessenberg form, this lower bound is $\Omega(n^3/\sqrt{M})$.

2.5.3. Applicability of the lower bounds

While we conjecture that all classical algorithms for applying one- or two-sided orthogonal transformations are subject to a lower bound in the form of Theorem 2.6, not all of those algorithms meet the assumptions of either of the two lower bound arguments presented in this section. However, many standard and efficient algorithms do meet the criteria.

For example, algorithms for QR decomposition that satisfy this assumption of Corollary 2.21 include the blocked, right-looking algorithm (currently implemented in (Sca)LAPACK: Anderson *et al.* 1992, Blackford *et al.* 1997) and the recursive algorithm of Elmroth and Gustavson (1998). The simplest version of Communication-Avoiding QR (*i.e.*, one that does not block transformations: see the last paragraph in Section 6.4 of Demmel, Grigori, Hoemmen and Langou 2012) satisfies the assumptions of Theorem 2.22. However, most practical implementations of CAQR do block transformations to increase efficiency in other levels of the memory hierarchy, and neither proof applies to these algorithms. The recursive QR decomposition algorithm of Frens and Wise (2003) is also communication-efficient, but again our proofs do not apply.

Further, Corollary 2.23 applies to the conventional blocked, right-looking algorithms in LAPACK (Anderson *et al.* 1992) and ScaLAPACK (Blackford *et al.* 1997) for reduction to Hessenberg, tridiagonal and bidiagonal forms. Our lower bound also applies to the first phase of the successive band reduction algorithm of Bischof, Lang and Sun (2000*a*, 2000*b*), namely reduction to band form, because this satisfies our requirement of forward progress. However, the second phase of successive band reduction does not satisfy our requirement of forward progress because it involves *bulge chasing*, which repeatedly creates nonzero entries outside the band and zeroes them out again (indeed, finding a tight lower bound for this part is an open question). But since the first phase does asymptotically more arithmetic than the second phase, our lower bound based on just the first phase cannot be much improved.

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

2.6. Extensions and impact of the lower bounds

2.6.1. Tensor contractions

The lower bounds extend to tensor contractions, defined below, which can be thought of as matrix multiplications in disguise. A mode m tensor is an m-way array (*e.g.*, matrices have two modes). See Kolda and Bader (2009) for a full discussion of tensor terminology and computations.

Suppose A and B are mode m_A and m_B tensors, respectively, with each mode of dimension n (so that A and B are $n \times n \times \cdots \times n$ tensors, though with different numbers of modes), and we wish to contract over the last c indices of A and the first c indices of B. Let $a = m_A - c$ and $b = m_B - c$; the contraction is defined componentwise as

$$C(j_1, \ldots, j_{a+b}) = \sum_{i_1=1}^{n} \cdots \sum_{i_c=1}^{n} A(j_1, \ldots, j_a, i_1, \ldots, i_c) \cdot B(i_1, \ldots, i_c, j_{a+1}, \ldots, j_{a+b}).$$

We reinterpret A, B, C as matrices, in order to recast this as matrix multiplication. Consider the lexicographical order on $\{1, \ldots, n\}^d$ and the bijection with $\{1, \ldots, n^d\}$ it induces, for some positive integer d. Reindexing in this manner, A becomes $n^a \times n^c$, B becomes $n^c \times n^b$, and C becomes $n^a \times n^b$, and the c summations collapse into a single summation from 1 to n^c. This is a 3NL computation, and Theorem 2.6 applies with $G = n^{a+b+c}$. This argument extends to the case when the modes of A and B have variable dimensions (as opposed to a fixed n), provided the contracted dimensions match (so that the contraction is well defined).

2.6.2. Memory-independent lower bounds

Note that the lower bounds of Theorem 2.6 include a factor of $M^{1/2}$ in the denominator of the expression. As pointed out in Irony *et al.* (2004), this indicates that having and using more local memory (larger M) in the parallel case can possibly reduce the communication costs of the corresponding algorithms. Indeed, many algorithms for matrix multiplication exploit this property (Dekel, Nassimi and Sahni 1981, McColl and Tiskin 1999, Solomonik and Demmel 2011); see Section 3.3.1.2 for more details. However, there are limits to how much extra local memory an algorithm can trade off to reduce communication; these limits come from a second set of lower bounds which are independent of the local memory size M.

The lower bound arguments of Irony *et al.* (2004) and Theorem 2.6 for classical algorithms and those proved in Section 4 for Strassen-like algorithms can be extended to prove memory-independent lower bounds (Ballard *et al.* 2012*d*). Theorem 2.25 states the result for classical, dense matrix multiplication, but it can be extended to 3NL computations and Strassen-like algorithms in a straightforward way.

Theorem 2.25. Suppose a parallel algorithm performing classical dense matrix multiplication begins with one copy of the input matrices and has computational cost $\Theta(n^3/P)$. Then, for sufficiently large P,[3] some processor must send or receive at least $\Omega\big(\frac{n^2}{P^{2/3}}\big)$ words.

These bounds dominate the previous memory-dependent bounds once the local memory size is sufficiently large. In particular, the memory-dependent and memory-independent bounds coincide when $M = \Theta\big(\frac{n^2}{P^{2/3}}\big)$. Further, the memory-independent bounds imply that there are strict limits on the perfect strong scaling range of matrix multiplication algorithms (both classical and Strassen-like). That is, within the *perfect strong-scaling range*, for a fixed problem size, by doubling the number of processors (and therefore doubling the total memory available) both the computational and communication

[3] The theorem applies to any $P \geq 2$ with a strict enough assumption on the load balance.

costs are halved. Beyond the perfect strong-scaling range, the reduction in computational cost is linear, but the reduction in communication cost is sublinear (Ballard *et al.* 2012*d*).

2.6.3. Energy bounds

In a recent work (Demmel *et al.* 2013*b*), we have modelled an algorithm's execution energy E via a small set of architectural parameters, to derive lower and upper bounds on the amount of energy that is consumed during run time. From these bounds, we prove that a realm of perfect strong scaling in energy and time (*i.e.*, for a given problem size n, the energy consumption remains constant as the number of processors P increases and the run time decreases in proportion to P) exists for matrix multiplication (classical and Strassen) and the direct ($O(n^2)$) N-body problem.

We model the total energy cost E of executing a parallel algorithm by

$$E = P(\gamma_e F + \beta_e W + \alpha_e S + \delta_e MT + \epsilon_e T). \tag{2.17}$$

Here γ_e, β_e and α_e are the energy costs (in joules) per flop, per word transferred and per message, respectively; δ_e is the energy cost per stored word in memory per second. The term $\delta_e MT$ assumes that we only pay for energy of memory that we are utilizing for the duration of the algorithm (a strong architectural assumption, but appropriate for a lower bound). Here ϵ_e is the remaining energy used per second in the system outside the memory. Note that ϵ_e encompasses the static leakage energy from circuits as well as the energy of other devices not defined within the model, such as disk behaviour, cooling, or transfers within the local memory hierarchy.

2.6.3.1. Example: energy costs of classical matrix multiplication. Recall that the asymptotic run time of a communication-optimal classical matrix multiplication is

$$T(n, P, M) = \frac{\gamma_t n^3}{P} + \frac{\beta_t n^3}{M^{1/2} P} + \frac{\alpha_t n^3}{M^{3/2} P} \tag{2.18}$$

within the perfect strong scaling range of $P_{\min} = n^2/M \leq P \leq n^3/M^{3/2}$ (recall Section 2.6.2), where γ_t is the seconds per flop, n^3/P is the number of flops, $n^3/(M^{1/2}P)$ is the total number of words sent, β_t is the seconds per word sent, $n^3/(M^{3/2}P)$ is the total number of messages sent, and α_t is the seconds per message sent. It scales perfectly because all terms of $T(n, P, M)$ are inversely proportional to P. Its energy cost is then

$$\begin{aligned} E(n, P, M) = (\gamma_e + \gamma_t \epsilon_e) n^3 + \left((\beta_e + \beta_t \epsilon_e) + \frac{(\alpha_e + \alpha_t \epsilon_e)}{M} \right) \frac{n^3}{M^{1/2}} \\ + \delta_e \gamma_t M n^3 + \left(\delta_e \beta_t + \frac{\delta_e \alpha_t}{M} \right) M^{1/2} n^3. \end{aligned} \tag{2.19}$$

Note that $E(n, P, M)$ is algebraically independent of P. Thus perfect scaling costs no additional energy. If P exceeds the range of perfect strong scaling, energy costs does depend on P, and may increase. See Demmel *et al.* (2013*b*) for further details on scaling, and for application of energy and power bounds to further algorithms.

We can use these formulas to ask various questions, such as: How do we choose P and M to minimize energy needed for a computation? Given the maximum allowed run time T, how much energy do we need to achieve it? Given the maximum allowed energy E, what is the minimum run time T that we can attain? Given a target energy efficiency (Gflops/Watt), what architectural parameters (γ_t, γ_e, *etc.*) are needed to achieve it? See Demmel *et al.* (2013*b*) for further details.

2.6.4. Heterogeneous machines

We can also extend these lower bounds to heterogeneous machine models. Note that for the distributed-memory parallel machine models (where computational and communication costs are homogenous among processors and pairs of processors, respectively) the lower bound applies to each processor individually. That is, the communication required by each processor is a function of the amount of computation that processor performs. Thus, we can also apply the analysis to heterogenous systems, where processors perform flops and access data at different rates. In Ballard, Demmel and Gearhart (2011*b*) we present a shared-memory heterogenous parallel model, extend the lower bounds to this model, and present optimal algorithms for matrix multiplication. Using the lower bounds, one can formulate and solve an optimization problem for the optimal amount of work to assign to each processor to achieve load balance, and the work can be assigned in a way that also minimizes communication.

2.6.5. Sparse matrix–matrix multiplication

For many sparse computations, the lower bounds for classical computations are unattainable. While these lower bounds suggest a maximum re-use of data of $O(\sqrt{M})$, many computations involving sparse matrices have inherent limitations on re-use which are much smaller. For example, applying a dense or sparse matrix to a dense vector yields an arithmetic intensity of only two flops per sparse matrix entry. Similarly, the multiplication of two sparse matrices also suffers from low arithmetic intensity. However, similar proof techniques can be employed to establish lower bounds that are tighter than those appearing in this paper and actually attainable. The bounds require extra assumptions on the set of algorithms to which they apply and the sparsity structure of the input matrices (*i.e.*, corresponding to Erdős–Rényi random graphs), but we establish matching lower and upper bounds in Ballard *et al.* (2013*f*).

2.6.6. Bandwidth–latency trade-offs

Our latency lower bounds have so far been derived in a very simple manner from the bandwidth lower bounds: the number of messages is at least as large as the number of words sent divided by the largest possible message size. In fact, as we will see in Section 3, these latency lower bounds are attainable for the parallel algorithms we have considered, when using the least memory possible ($M = n^2/P$); they are also attainable for matmul when using more memory than this (see Sections 2.6.2 and 3.3.1.2). However, for LU, QR and other algorithms with more complicated dependency graphs than matmul, the latency lower bounds, for the case of extra memory, were unattainable. This led to new lower bounds showing that there is a trade-off between latency and bandwidth along the critical path that must be respected, such that both cannot be simultaneously minimized when using extra memory; indeed, latency alone is minimized by algorithms that use the least memory. For example, in the case of LU and Cholesky factorizations, the product of the bandwidth and latency costs must be $\Omega(n^2)$. The work by Solomonik, Carson, Knight and Demmel (2014) extends these results to a variety of direct and iterative matrix computations.

2.6.7. Computations that access arrays

The lower bound approach developed by Irony *et al.* (2004) for matrix multiplication, and generalized above in Theorem 2.6 for 3NL computations (Definition 2.4), has recently been extended by Christ *et al.* (2013) to a larger class of computations, of the form

$$\text{for } i \in S, \quad \text{inner\_loop}(i, \text{Mem}(\mathfrak{a}_1(\phi_1(i))), \ldots, \text{Mem}(\mathfrak{a}_m(\phi_m(i)))),$$

where $S \subset \mathbb{Z}^d$ is the non-empty and finite iteration space, indexed by d-tuples of integers $i = (i_1, \ldots, i_d)$, and for $j \in \{1, \ldots, m\}$, $\phi_j \colon \mathbb{Z}^d \to \mathbb{Z}^{d_j}$ is an affine map. As in 3NL computations, the functions $\mathfrak{a}_j \colon \phi_j(S) \to \mathcal{M}$ are injections into slow/global memory, and the subroutines inner_loop$(i, \cdot)$ depend nontrivially on their arguments.

Recall that the proof of Theorem 2.6 hinges on obtaining an upper bound $F = F(M)$ on the number of 3NL operations (here, inner_loop calls). The Loomis–Whitney inequality (Loomis and Whitney 1949) can be used to show $F = O(M^{3/2})$ for 3NL computations. The main result of Christ *et al.* (2013) applied a recent generalization of Loomis and Whitney (1949) in Bennett, Carbery, Christ and Tao (2010) to show that $F = O(M^\sigma)$, where $\sigma = \min 1^T s$ subject to $\Delta s \geq 1$, where $\{H_i\}$ is an enumeration of all nontrivial subgroups $H_i \leq \mathbb{Z}^d$ and $\Delta_{ij} = \text{rank}(\phi_j(H_i))/\text{rank}(H_i)$. (The minimum σ can be computed decidably.) Applying reasoning similar to that of Theorem 2.6, this leads to a lower bound $\Omega(|S|/M^{\sigma-1})$ on the number of words moved between slow and fast memory, and this result can be extended to bound communication in a distributed-memory parallel machine, or to

bound the number of messages (latency cost) rather than the number of words (bandwidth cost).

There are many open questions regarding attainability of these bounds in Christ *et al.* (2013), but in the absence of inter-iteration data dependences (*i.e.*, no imposed order on S), and for sufficiently large and regular S, it is attainable in the special case where each ϕ_j is a projection onto a subset of the coordinates (*e.g.*, classical matrix multiplication).

3. Communication-optimal classical algorithms

In this section we focus on classical direct algorithms for sequential and parallel machines and discuss the current state of the art in terms of communication costs. The main goal of this section is to provide a comprehensive (though certainly not exhaustive) summary of communication-optimal algorithms for dense linear algebra computations and provide references to papers with algorithmic details, communication cost analyses, and demonstrated performance improvements. We consider all the fundamental direct numerical linear algebra computations – BLAS, Cholesky and symmetric indefinite factorizations, LU and QR factorizations, eigenvalue and singular value factorizations – and compare the best algorithms with the lower bounds presented in Section 2.

It is also natural to compare the lower bounds of Section 2 with the costs of standard implementations in widely used libraries such as LAPACK and ScaLAPACK. As we will see, these standard libraries often do not attain the lower bounds, even for the well-studied problem of parallel matrix multiplication. This observation has motivated a great deal of recent work by the community to invent new algorithms that do attain the lower bounds (or at least do asymptotically less communication than the standard libraries). In fact, some of these communication-optimal algorithms have been in the literature for a while, but their advantages from the communication cost perspective have only recently been recognized.

We consider here both sequential and parallel algorithms. Recall the sequential two-level memory model presented in Section 1.2: we consider communication between a fast memory of size M and a slow memory of unbounded size, and we track both the number of words and messages that an algorithm moves. Because a message requires its words to be stored contiguously in slow memory, we must specify the matrix data layout in determining latency costs. We also consider the multiple-level memory hierarchy sequential model in this section, as it more accurately reflects many of today's machines.

Recall also the distributed-memory parallel memory model presented in Section 1.2: in this case we consider communication among a set of P processors with a link between every pair of processors, and we track both words

and messages communicated by the parallel machine along the critical path of the algorithm. All of the algorithms discussed in this section assume a block-cyclic data distribution, where a block size of 1 gives a cyclic distribution and a block size of $n/\sqrt{P}$ gives a blocked distribution (Blackford *et al.* 1997).

In the following subsections we summarize the state of the art in communication-optimal sequential (Section 3.1) and parallel algorithms (Section 3.2) (providing tables of references to communication-optimal algorithms in each case), and then discuss in Section 3.3 each of the fundamental computations in more detail. First, we highlight three key ideas that have been instrumental in improving the communication efficiency of algorithms for several different computations.

- *Two-dimensional blocking in the sequential case.* A principal innovation of the LAPACK libraries was the implementation of 1D blocking: dividing the matrix into column panels and implementing algorithms based on the pattern of panel factorizations and efficiently blocked trailing matrix updates. While this approach yields high performance in many situations, it is not always sufficient for communication optimality. More recent communication-optimal algorithms use a 2D form of blocking that yields better efficiency in the panel factorizations in particular. However, these 2D innovations include fundamental changes in the computations: in the case of LU (Grigori *et al.* 2011), they involve a form of pivoting that is different but as stable in practice (see Algorithm 3.1), and in the case of QR (Demmel, Grigori, Hoemmen and Langou 2012), they involve a different representation of the orthogonal factor (see Algorithm 3.2).
- *Recursion.* Communication efficiency of blocked algorithms often involves judicious choice of block sizes, based on the size of fast or local memory. For many linear algebra computations, recursive algorithms 'automatically' choose the right block sizes and involve simpler and more elegant code. In the sequential case, such algorithms are often *cache-oblivious*, for which there are many examples (Frigo, Leiserson, Prokop and Ramachandran 1999, Gustavson 1997, Toledo 1997, Frens and Wise 2003, Ballard *et al.* 2013*f*). Such algorithms minimize communication on two-level and hierarchical memory models. Recursion is also an effective tool in the parallel case, and several communication-optimal algorithms can be expressed succinctly in this way (McColl and Tiskin 1999, Tiskin 2007, Demmel *et al.* 2013*a*).
- *Trading off extra memory for reduced communication in the parallel case.* Conventional parallel algorithms (*e.g.*, those in ScaLAPACK) are designed to use little extra memory: no more than a constant factor times the memory required to store the input and output. However,

the communication lower bounds presented in Section 2 decrease as the size of the local memory increases, suggesting the possibility of trading off extra memory for reduced communication: that is, if the local memory $M = cn^2/P$, then the communication cost is expected to be a decreasing function of c. Indeed, this is possible for matrix multiplication (Berntsen 1989, Irony *et al.* 2004, Demmel *et al.* 2013*a*) and other linear algebra computations (Tiskin 2007, Solomonik and Demmel 2011). See Table 3.3 for a summary of known algorithms that navigate this trade-off optimally.

3.1. Summary of classical sequential algorithms

In this section, we summarize the current state of the art for algorithms that perform the classical $O(n^3)$ computations on sequential machines. For each of the computations considered here, we compare the communication costs of the algorithms to the lower bounds presented in Section 2. Table 3.1 summarizes the communication-optimal classical algorithms for the most fundamental dense linear algebra computations. We differentiate between algorithms that minimize communication only in the two-level model and those that are optimal in a multiple-level memory hierarchy too. We also differentiate between algorithms that minimize only bandwidth costs and those that minimize both bandwidth and latency costs.

We say that an algorithm is communication-optimal in the sequential model if its communication costs are within a constant factor of the corresponding lower bound (we sometimes relax it to a logarithmic factor) and that it performs no more than a constant factor more computation than alternative algorithms. Most of the algorithms have the same leading constant in computational cost as the standard algorithms, though we note where constant factor increases occur. In some cases, there exists a small range of matrix dimensions where the algorithm is sub-optimal: the communication cost includes a term that is typically a low-order term, but sometimes exceeds the lower bound. For example, the rectangular recursive algorithms of Elmroth and Gustavson (1998), Gustavson (1997) and Toledo (1997) are sub-optimal with respect to bandwidth cost only in the rare case when n satisfies $n/\log n \ll \sqrt{M} \ll n$ but optimal in all other cases (Ballard *et al.* 2013*f*). We omit these details for sufficiently small ranges.

One may imagine that sequential algorithms minimizing communication for any number of levels of a memory hierarchy might be very complex, possibly depending on not just the number of levels but also their sizes. In this context, it is worth distinguishing a class of algorithms, called *cache-oblivious* (Frigo *et al.* 1999), that can minimize communication between all levels (at least asymptotically) independent of the number of levels and their sizes, assuming a nested memory hierarchy (*i.e.*, faster memory is

Table 3.1. Sequential classical algorithms attaining communication lower bounds. We list algorithms that attain the lower bounds for two levels and multiple levels of memory hierarchy.

Computation	Two-level memory		Multiple-level memory	
	Minimizes words	Minimizes messages	Minimizes words	Minimizes messages
BLAS-3	[1, 2]		[1, 2]	
Cholesky	[3, 4, 5, 6]	[3, 5, 6]	[3, 5, 6]	
LU	[6, 7, 8, 9]	[7, 8]	[6, 7, 9]	[7]
Symmetric indefinite	[10]		[10]	
QR	[7, 11, 12, 13]	[7, 11, 13]	[7, 12, 13]	[7, 13]
Symmetric eigenproblem	[14, 15]		[14]	
SVD	[14, 15, 16]		[14, 16]	
Nonsymm. eigenproblem	[14]		[14]	

[1] Ballard *et al.* (2013*f*)
[2] Frigo *et al.* (1999)
[3] Ahmed and Pingali (2000)
[4] Anderson *et al.* (1992)
[5] Ballard *et al.* (2010)
[6] Gustavson (1997)
[7] Ballard *et al.* (2013*f*)
[8] Grigori *et al.* (2011)
[9] Toledo (1997)
[10] Ballard *et al.* (2013*a*)
[11] Demmel *et al.* (2012)
[12] Elmroth and Gustavson (1998)
[13] Frens and Wise (2003)
[14] Ballard *et al.* (2011*a*)
[15] Ballard *et al.* (2013*d*)
[16] Haidar *et al.* (2013)

smaller). These algorithms are recursive, and provided that a matching recursive layout is used, these algorithms may also minimize the number of messages independent of the number of levels of memory hierarchy. Not only do cache-oblivious algorithms perform well in theory but they can also be adapted to perform well in practice; see Yotov *et al.* (2007), for example.

We emphasize that only a few of the communication-optimal algorithms referenced here are included in standard libraries such as LAPACK. While this section focuses on asymptotic complexity rather than measured performance on current architectures, many of the papers referenced for algorithms here also include performance data and demonstrate significant speed-ups over asymptotically sub-optimal alternatives. Our communal goal is to eventually make all of these algorithms available via widely used libraries.

Table 3.2. Parallel classical algorithms attaining communication lower bounds assuming minimal memory is used. That is, these algorithms have computational cost $\Theta(n^3/P)$ and use local memory of size $O(n^2/P)$.

Computation	Minimizes words	Minimizes messages
BLAS-3	[1, 2, 3, 4]	[1, 2, 3, 4]
Cholesky	[2]	[2]
LU	[2, 5, 10, 11]	[5, 10, 11]
Symmetric indefinite	[2, 6, 9]	[6, 9]
QR	[2, 7]	[7]
Symmetric eigenproblem and SVD	[2, 8, 9]	[8, 9]
Nonsymmetric eigenproblem	[8]	[8]

[1] Agarwal *et al.* (1994)
[2] Blackford *et al.* (1997)
[3] Cannon (1969)
[4] van de Geijn and Watts (1997)
[5] Grigori *et al.* (2011)
[6] Ballard *et al.* (2013*a*)
[7] Demmel *et al.* (2012)
[8] Ballard *et al.* (2011*a*)
[9] Ballard *et al.* (2013*d*)
[10] Solomonik and Demmel (2011)
[11] Tiskin (2002)

3.2. Summary for the parallel algorithms

In this section we summarize the current state of the art for algorithms that perform the classical $O(n^3)$ computations for dense matrices on parallel machines. We first assume that no more than a constant factor of extra local memory is used. For each of the computations considered here, we can compare the communication complexity of the algorithms to the lower bounds presented in Section 2, where we fix the local memory size to $M = \Theta(n^2/P)$. Table 3.2 summarizes the communication-optimal algorithms in this case for the most fundamental dense linear algebra computations. Another term for these minimal memory algorithms is '2D', which was first used to distinguish minimal memory matrix multiplication algorithms from so-called '3D' algorithms that do use more than a constant factor of extra memory. In these 2D algorithms, the processors are organized in a two-dimensional grid, with most communication occurring within processor rows or columns.

Recall the lower bounds that apply to these dense computations, where the number of g_{ijk} operations is $G = \Theta(n^3/P)$. For these values of G and M, the lower bound on the number of words communicated by any processor is $\Omega(n^2/\sqrt{P})$, and the lower bound on the number of messages is $\Omega(\sqrt{P})$. In order for an algorithm to be considered communication-optimal,

we require that its communication complexity be within a polylogarithmic (in P) factor of the two lower bounds and that it perform no more than a constant factor more computation than alternative algorithms (*i.e.*, the parallel computational cost is $\Theta(n^3/P)$).

The asymptotic communication costs for ScaLAPACK algorithms are given in Table 5.8 of Blackford *et al.* (1997). Note that the bandwidth costs for those algorithms include a $\log P$ factor due to the assumption that collective communication operations (*e.g.*, reductions and broadcasts) are performed with tree-based algorithms. Better algorithms exist for these collectives that do not incur the extra factor in bandwidth cost. For example, a broadcast can be performed with a scatter followed by an all-gather: see Thakur, Rabenseifner and Gropp (2005) or Chan, Heimlich, Purkayastha and van de Geijn (2007) for more details. Thus, the extra logarithmic factor is not inherent in the algorithm, only in the way collective operations were first implemented in ScaLAPACK.

However, we emphasize the fact that the optimality of algorithms listed in Table 3.2 holds only if no more than $O(n^2/P)$ local memory is used. If we remove the assumption and let M grow larger, then the communication lower bounds for these computations decreases, exposing a trade-off between local memory and communication costs.

There are only a few known algorithms for linear algebra that are able to navigate the memory–communication trade-off, but none as successfully as in the case of matrix multiplication. For matrix multiplication, both bandwidth and latency costs can be simultaneously reduced with the use of extra memory. That is, the bandwidth cost can be reduced from $\Theta(n^2/P^{1/2})$ to $\Theta(n^2/P^{2/3})$ and the latency cost can be reduced from $\Theta(\sqrt{P})$ down to $\Theta(\log P)$, if enough memory is available. In the case of linear algebra computations that involve more dependences than matrix multiplication, there exists a second trade-off between bandwidth and latency costs. By the bandwidth–latency trade-off in Solomonik *et al.* (2014) for various algorithms (presented in Section 2.6.6), bandwidth and latency are not simultaneously improved for $M \gg \Omega(n^2/P)$: decreasing the bandwidth cost below $\Theta(n^2/P^{1/2})$ increases the latency cost above $\Theta(\sqrt{P})$. The bandwidth–latency trade-off bound for LU and Cholesky is given by $W \cdot S = \Omega(n^2)$.

3.3. Fundamental numerical linear algebra computations

We now discuss the rows of each of Tables 3.1, 3.2, and 3.3 in more detail. For each fundamental computation, we discuss sequential and parallel cases separately.

3.3.1. BLAS computations

While the lower bounds given in Section 2.3.2.1 apply to all BLAS computations, only the BLAS-3 computations have algorithms that attain them.

Table 3.3. Parallel classical algorithms attaining communication lower bounds with no assumption on local memory use. These algorithms have computational cost $\Theta(n^3/P)$ and may use local memory of size more than $\Theta(n^2/P)$. We list algorithms that minimize communication relative to the tightest lower bounds (memory-dependent or memory-independent), including the bandwidth–latency lower bound trade-off described in Section 2.6.6, in the case of Cholesky and LU.

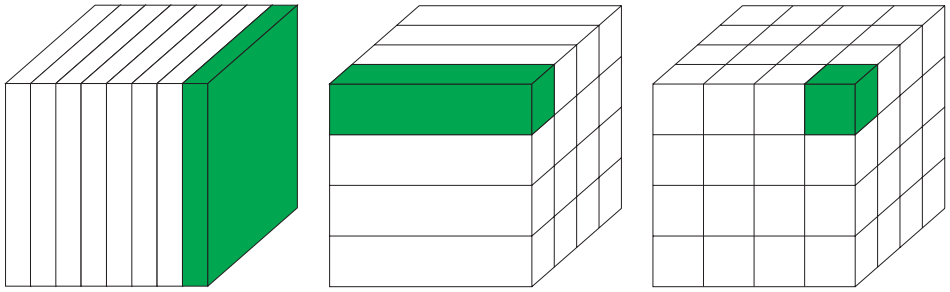
Computation	Minimizes communication
Matmul	[1, 2, 3]
Cholesky	[4, 5, 6]
LU	[2, 7]

[1] McColl and Tiskin (1999)
[2] Solomonik and Demmel (2011)
[3] Demmel *et al.* (2013*a*)
[4] Tiskin (2002)
[5] Georganas *et al.* (2012)
[6] Lipshitz (2013)
[7] Tiskin (2007)

In the case of BLAS-2 and BLAS-1 computations, the arithmetic intensity (*i.e.*, the ratio of the number of arithmetic operations to the number of inputs and outputs) is $O(1)$, so it is impossible for the bandwidth cost to be a factor of $O(\sqrt{M})$ smaller than the arithmetic cost, assuming the data has to be read from slow memory.

3.3.1.1. Sequential case. For BLAS-3 computations, blocked versions of the naive algorithms attain the lower bound in the two-level memory model when the block size is chosen to be $\Theta(\sqrt{M})$: see, for example, the BLOCK-MULT algorithm in Frigo *et al.* (1999) for matrix multiplication. In order to attain the corresponding latency cost lower bound, a block-contiguous data structure is necessary so that every block computation involves contiguous chunks of memory. Furthermore, the block computations can themselves be blocked. Using a nested level of blocking for each level of memory (and choosing the block sizes appropriately), these algorithms can minimize communication between every pair of successive levels in a memory hierarchy. Note that a matching hierarchical block-contiguous data structure is needed to minimize latency costs. We do not include a reference in Table 3.1, as these blocked algorithms are generally considered folklore.

In addition to the explicitly blocked algorithms, there are recursive algorithms for all of the BLAS-3 computations. As explained in Frigo *et al.* (1999) for rectangular matrix multiplication, these recursive algorithms also attain the lower bounds of Section 2.3.2.1. In order to minimize latency costs, we use a matching recursive data layout, such as the rectangular recursive layout of Ballard *et al.* (2013*f*) which matches the REC-MULT



(a) 1D (b) 2D (c) 3D

Figure 3.1. Illustrations of 1D, 2D and 3D algorithms for matrix multiplication. The entire cube represents all the scalar multiplications (work) associated with square matrix multiplication: (i, j, k) stands for the multiplication $A_{ik} \cdot B_{kj}$ to be accumulated into C_{ij}. Lines correspond to divisions of work to processors; the work of a particular processor is shaded. While 1D algorithms divide the cube in only one dimension, 2D algorithms divide the cube in two dimensions, and 3D algorithms divide the cube in all three dimensions. In particular, 3D algorithms divide the work associated with computing a single output matrix entry among multiple processors, which need not be the case for 1D and 2D algorithms.

algorithm of Frigo *et al.* (1999). For computations involving square matrices, data layouts based on Morton orderings and its variants help minimize latency costs. For recursive algorithms for triangular solve, see Algorithm 3 of Ballard *et al.* (2010), for example, where the right-hand sides form a square matrix, or Algorithm 5 of Ballard *et al.* (2013*f*) for the general rectangular case. Similar algorithms exist for symmetric and triangular matrix multiplications and symmetric rank-k updates. Because these algorithms are recursive and cache-oblivious, they minimize communication costs between every pair of memory levels in a hierarchy.

3.3.1.2. Parallel case. ScaLAPACK (Blackford *et al.* 1997) includes the Parallel BLAS library, or PBLAS, which has algorithms for matrix multiplication (and its variants) and triangular solve that minimize both words and messages, assuming minimal memory is used. The history of communication-optimal matrix multiplication goes back to Cannon (1969). While Cannon's algorithm is asymptotically optimal, a more robust and tunable algorithm known as SUMMA (Agarwal, Gustavson and Zubair 1994, van de Geijn and Watts 1997) is more commonly used in practice. For a more complete summary of minimal memory, or 2D, matrix multiplication algorithms, see Irony *et al.* (2004, Section 4).

The first algorithms to reduce the communication cost of parallel matrix multiplication by using extra memory were developed by Aggarwal *et al.* (1990) (in the LPRAM model) and Berntsen (1989) (on a hypercube network). Aside from using extra local memory, the main innovation in these

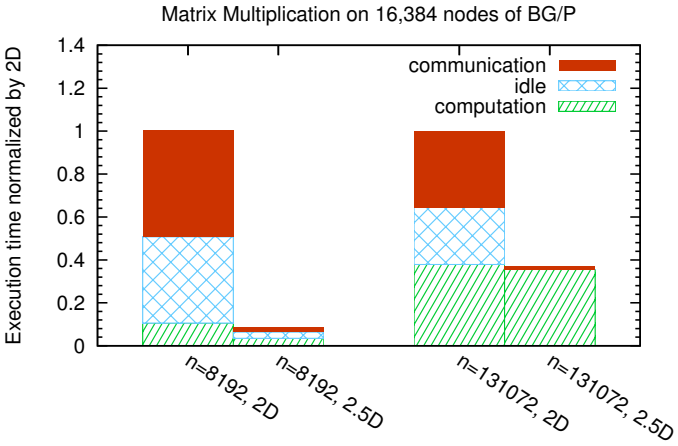


Figure 3.2. 2.5D matrix multiplication on BG/P, 16K nodes / 64K cores.

algorithms is to divide the work for computing single entries in the output matrix among multiple processors; in the case of Cannon's and other 2D algorithms, a single processor computes all of the scalar multiplications and additions for any given output entry. See Figure 3.1 for a visual classification of 1D, 2D, and 3D matrix multiplication algorithms based on the assignment of work to processors.

For a more complete summary of early 3D dense matrix multiplication algorithms, see Irony *et al.* (2004, Section 5). The extra memory required for the 3D algorithms is $O(n^2/P^{2/3})$, or $O(P^{1/3})$ times the minimal amount of memory required to store the input and output matrices. The communication savings, compared to 2D algorithms, is a factor of $O(P^{1/6})$ words and $\tilde{O}(P^{1/2})$ messages (where the latter notation hides away logarithmic factors in P). However, these algorithms provide only a binary alternative to 2D algorithms: only if enough memory is available can 3D algorithms be employed. McColl and Tiskin (1999) showed how to navigate the trade-off continuously (in the BSPRAM model): for example, given local memory of size $\Theta(n^2/P^{2\alpha+2/3})$, their algorithm achieves bandwidth cost of $O(n^2/P^{2/3-\alpha})$ for $0 \leq \alpha \leq 1/6$. Later, Solomonik and Demmel (2011) independently developed and implemented a practical version of the algorithm which generalizes the 2D SUMMA algorithm, demonstrating speed-ups of up to 12×. See Figure 3.2.

Because their approach fills the gap between 2D and 3D algorithms, the authors coined '2.5D' to describe their algorithm. Their algorithm is *topology-aware*, meaning that it addresses and takes advantage of the interprocessor network topology. It thus performs well on machines with torus of dimension three or higher (similar to Cannon's algorithm performing well on 2D torus machines). A similar approach is used in the communication-avoiding all-pairs shortest paths (APSP) algorithm, obtaining a speed-up

of 6.2× on 24K cores of a Cray XE6 machine compared to the best previous algorithm (Solomonik *et al.* 2013). This approach is also used in the communication-avoiding Cyclops Tensor Framework, gaining a speed-up of up to 3× on 3K cores of a Cray XE6 machine, compared to previous algorithms (Solomonik, Matthews, Hammond and Demmel 2013*c*). The 2.5D algorithm (Solomonik and Demmel 2011) is generalized to rectangular matrices in Schatz, Poulson and van de Geijn (2013), though it is not communication-optimal in all cases.

A recursive algorithm, which utilizes the parallelization approach in Ballard *et al.* (2012*e*) and Lipshitz, Ballard, Demmel and Schwartz (2012), also achieves the same asymptotic communication costs for square matrices and is generalized to rectangular matrices in Demmel *et al.* (2013*a*) and Lipshitz (2013), demonstrating speed-ups of up to 10× over MKL and 141× over ScaLAPACK for some rectangular matrix dimensions.

Note that the algorithms that are able to continuously navigate the trade-off between memory and communication can exhibit perfect strong scaling. That is, by doubling the number of processors (and therefore doubling the total amount of available memory), both computational and communication costs are reduced by a factor of two, thereby cutting the run time by a factor of two. This is possible only within a limited range, due to the existence of the lower bound discussed in Section 2.6.2 and Ballard *et al.* (2012*d*). For more details on perfect energy scaling, see Demmel *et al.* (2013*b*).

3.3.2. Cholesky factorization

Cholesky factorization is used primarily for solving symmetric, positive definite linear systems of equations. Since pivoting is not required, making the algorithm communication-optimal is much easier (*e.g.*, compared to LU). For a more complete discussion of sequential algorithms for Cholesky factorization and their communication properties, see Ballard *et al.* (2010).

3.3.2.1. Sequential case. The reference implementation in LAPACK (Anderson *et al.* 1992) (`potrf`) is a blocked algorithm, and by choosing the block size to be $\Theta(\sqrt{M})$, the algorithm attains the lower bound of Corollary 2.11. As in the case of the BLAS computations, a block-contiguous data structure can be used to obtain the latency cost lower bound. An algorithm with nested levels of blocking and a matching data layout can minimize communication for multiple levels of memory.

A recursive algorithm for Cholesky factorization was first proposed in Gustavson (1997) and later matched with a block-recursive data structure in Ahmed and Pingali (2000). We present the communication cost analysis in Ballard *et al.* (2010), where the algorithm is shown to be communication-optimal and cache-oblivious as long as cache-oblivious BLAS subroutines

are used. Thus, the recursive algorithm is optimal for both two-level and multiple-level memory models.

Note also that for the Cholesky factorization of sparse matrices whose sparsity structures satisfy certain graph-theoretic conditions (having 'good separators'), the lower bound of Corollary 2.11 can also be attained (Grigori *et al.* 2010). For general sparse matrices, the problem is open.

3.3.2.2. Parallel case. ScaLAPACK's parallel Cholesky routine (`pxposv`) minimizes both words and messages on distributed-memory machines with the right choice of block size. See Ballard *et al.* (2010) for a description of the algorithm and its communication analysis. Note that the bandwidth cost in that paper includes a $\log P$ factor that can be removed with a more efficient broadcast routine.

Tiskin presents a recursive algorithm in the BSP model (Tiskin 2002) for LU factorization without pivoting that uses extra memory to reduce communication, though it exhibits a trade-off between bandwidth and latency costs. This algorithm can be applied to symmetric positive definite matrices, though it uses explicit triangular matrix inversion and multiplication (ignoring stability issues) and also ignores symmetry. Georganas *et al.* (2012) extend the ideas of Solomonik and Demmel (2011) to the symmetric positive definite case, saving arithmetic by exploiting symmetry and maintaining stability by using triangular solves. Lipshitz (2013) provides a similar algorithm for Cholesky factorization, along with a recursive algorithm for triangular solve, that also maintains symmetry and stability. These Cholesky algorithms achieve a bandwidth cost of $O(n^2/P^\alpha)$ and latency cost of $O(P^\alpha)$ for $1/2 \leq \alpha \leq 2/3$, by using $P^{2\alpha-1}$ times as much memory as the minimum required. In other words, to minimize bandwidth costs, it pays to use more memory than the minimum necessary, but to minimize latency, the minimum memory algorithm is optimal. These communication costs match the trade-off lower bounds proved in Solomonik *et al.* (2014) for Cholesky factorization, that is, the product of bandwidth and latency costs is $\Omega(n^2)$.

3.3.3. LU factorization

For general, nonsymmetric linear systems, an LU factorization is the direct method of choice. In order to maintain numerical stability, algorithms must incorporate some form of pivoting. For performance reasons (and because it is generally sufficient in practice), we consider performing only row interchanges. The development of communication-optimal complete-pivoting algorithms (those that perform both row and column interchanges) is ongoing work; see Demmel, Grigori, Gu and Xiang (2013*c*, Section 5) for a possible approach.

3.3.3.1. Sequential case. There is a long history of algorithmic innovation to reduce communication costs for LU factorizations on sequential machines. Table 1 in Ballard *et al.* (2013*f*) highlights several of the innovations and compares the asymptotic communication costs of the algorithms discussed here.

The LU factorization algorithm in LAPACK (`getrf`) uses partial pivoting and is based on 'blocking' in order to cast much of the work in terms of matrix–matrix multiplication rather than working column by column and performing most of the work as matrix–vector operations. The algorithm is a right-looking, blocked algorithm, and by choosing the right block size, the algorithm asymptotically reduces the communication costs compared to the column-by-column algorithm. In fact, for very large matrices ($m \times n$, with $m, n > M$) it can attain the communication lower bound (see Corollary 2.10). However, for reasonably sized matrices ($\sqrt{M} < m, n < M$) the blocked algorithm is sub-optimal with respect to its communication costs.

In the late 1990s, both Toledo (1997) and Gustavson (1997) independently showed that using recursive algorithms can reduce communication costs. The analysis in Toledo (1997) shows that the recursive LU (RLU) algorithm, which also performs partial pivoting, moves asymptotically fewer words than the algorithm in LAPACK when $m < M$ (though latency cost is not considered in that work). In fact, the RLU algorithm attains the bandwidth cost lower bounds. Furthermore, RLU is cache-oblivious, so it minimizes bandwidth cost for any fast memory size and between any pair of successive levels of a memory hierarchy.

Motivated by the growing latency cost on both sequential and parallel machines, Grigori *et al.* (2011) considered bandwidth and latency cost metrics and presented an algorithm called Communication-Avoiding LU (CALU) that minimizes both. In order to attain the lower bound for latency cost (proved in that paper via reduction from matrix multiplication: see equation (2.1)), the authors used the block-contiguous layout and introduced tournament pivoting as a new and different scheme than partial pivoting.

The tournament pivoting scheme makes different pivoting choices to partial pivoting, though the two schemes are equivalent in a weak sense. Grigori *et al.* (2011) show that, in exact arithmetic, the Schur complement obtained after each step of performing tournament pivoting on a matrix A is the same as the Schur complement obtained after performing partial pivoting on a larger matrix whose entries are the same as the entries of A (sometimes slightly perturbed) and zeros. More generally, the entire CALU process is equivalent to LU with partial pivoting on a large, but very sparse matrix, formed by entries of A and zeros. Hence we expect that tournament pivoting will behave similarly to partial pivoting (and thus be stable) in practice. Indeed, Grigori *et al.* (2011) present extensive experiments on random matrices and a set of special matrices to support this claim. The upper bound

on the growth factor of CALU is worse than that of LU with partial pivoting. However, there are Wilkinson-like matrices for which partial pivoting leads to an exponential growth factor, but not tournament pivoting, and *vice versa*. We discuss the tournament pivoting algorithm in more detail for the parallel case in Section 3.3.3.2.

One drawback of CALU is that it requires knowledge of the fast memory size for both algorithm and data layout (*i.e.*, it is not cache-oblivious). Making the cache-oblivious RLU algorithm latency optimal had been an open problem for a few years. For example, arguments are made in Ballard *et al.* (2010) and Grigori *et al.* (2011) that RLU is not latency optimal for several different fixed data layouts. In Ballard *et al.* (2013*f*), using a technique called 'shape-morphing', we show that attaining communication optimality, being cache-oblivious, and using partial pivoting are all simultaneously achievable.

3.3.3.2. Parallel case. ScaLAPACK's LU routine (`pxgesv`) minimizes the number of words moved but not the number of messages. The parallel version of the communication-avoiding LU algorithm of Grigori *et al.* (2011) is able to reduce the number of messages. The use of tournament pivoting allows the overall algorithm to reach both bandwidth and latency cost lower bounds.

Tournament pivoting gets its name from the way it chooses pivot rows. In conventional Gaussian elimination, one pivot row is chosen at a time by selecting the maximum element (in absolute value) in the column. In tournament pivoting, some number b rows are chosen at a time by selecting the b most linearly independent rows of a column panel. The selected rows are the winners of a tournament, which can be thought of as a reduction with the reduction operator choosing the b most linearly independent rows from a set of $2b$ rows (this selection is done using LU with partial pivoting on a $2b \times b$ matrix). After the b most linearly independent rows are chosen, they are pivoted to the top of the column panel and the factorization is repeated via Gaussian elimination with no pivoting. We present pseudocode for parallel Tall Skinny LU (TSLU) in Algorithm 3.1.

In the sequential case, partial pivoting can be maintained while still minimizing both words and messages using a technique known as shape-morphing (Ballard *et al.* 2013*f*). Unfortunately, the idea of shape-morphing is unlikely to yield the same benefits in the parallel case. Choosing pivots for each of n columns lies on the critical path of the algorithm and therefore must be done in sequence. Each pivot choice either requires at least one message or for the whole column to reside on a single processor. This seems to require either $\Omega(n)$ messages or $\Omega(n^2)$ words moved, which both asymptotically exceed the respective lower bounds.

Algorithm 3.1 $[\mathcal{P}, L, U] = \text{TSLU}(A)$; 'Tall Skinny LU' with tournament pivoting

Require: Number of processors P is a power of 2 and i is the proc. index
Require: A is $m \times b$, distributed in block row layout; A_i is proc. i's block
Require: GEPP is local Gaussian elimination with partial pivoting
Require: GENP is local Gaussian elimination with no pivoting
Ensure: $\mathcal{P}A = LU$ with L_i is owned by proc. i and U is owned by proc. 0
Ensure: $\mathcal{P}$ is chosen so that $\mathcal{P}A(1:b,:) = B_{0,\log P}$

1: $[\mathcal{P}_{i,0}, L_{i,0}, U_{i,0}] = \text{GEPP}(A_i)$
2: $B_{i,0} = (\mathcal{P}_{i,0}A_i)(1:b,:)$ ▷ extract b 'most independent' rows from A_i
3: **for** $k = 1$ to $\log P$ **do** ▷ binary reduction tree for tournament
4: **if** $i \equiv 0 \mod 2^k$ **then**
5: $j = i + 2^{k-1}$
6: Receive $B_{j,k-1}$ from processor j
7: $C = \begin{bmatrix} B_{i,k-1} \\ B_{j,k-1} \end{bmatrix}$
8: $[\mathcal{P}_{i,k}, L_{i,k}, U_{i,k}] = \text{GEPP}\,(C)$
9: $B_{i,k} = (\mathcal{P}_{i,k}C)(1:b,:)$ ▷ extract b 'most independent' rows from left and right children in the tree
10: **else if** $i \equiv 2^{k-1} \mod 2^k$ **then**
11: Send $B_{i,k-1}$ to processor $i - 2^{k-1}$
12: **end if**
13: **end for**
14: **if** $i = 0$ **then**
15: Broadcast $A_0(1:b,:)$ (to origins of rows of $B_{0,\log P}$)
16: $A_0(1:b,:) = B_{0,\log P}$
17: $[L_0, U] = \text{GENP}(A_0)$
18: Broadcast U to all processors
19: **else**
20: If applicable, receive rows of A_0 and update A_i
21: Receive U
22: $L_i = \text{TRSM}(A_i, U)$
23: **end if**

There exist other parallel algorithms for LU factorization that use extra memory to reduce communication. Tiskin (2007) incorporates pairwise pivoting into a new algorithm with the same asymptotic costs as the algorithm in Tiskin (2002), which has no pivoting. While pairwise pivoting is not as stable as partial pivoting in theory or practice (Sorensen 1985, Trefethen and Schreiber 1990), the approach is generic enough to apply to QR factorization based on Givens rotations. However, the algorithm seems to be of only theoretical value: for example, the constants in the computational costs are much larger than minimal-memory algorithms. Solomonik and Demmel (2011) devise a stable LU factorization algorithm that uses tournament pivoting, is computation-efficient, and achieves the same asymptotic costs as the algorithm of Tiskin (2002, 2007); they demonstrate significant speed-ups compared to minimal-memory LU algorithms.

As in the case of Cholesky, an important characteristic of extra-memory algorithms for LU factorization is a trade-off between bandwidth and latency costs. That is, the product of the number of words and the number of messages sent is $\Theta(n^2)$; this trade-off is shown to be necessary in Solomonik *et al.* (2014). We mention here only one of many speed-ups: up to 2.1× of 2.5D LU on 64K of an IBM BG/P machine compared to previous parallel LU factorization (Georganas *et al.* 2012). A similar approach, applied to the direct N-body problem, leads to speed-ups of up to 11.8× on the 32K core IBM BG/P, compared to similarly tuned 2D algorithms (Driscoll *et al.* 2013).

3.3.4. Symmetric indefinite factorizations

If a linear system is symmetric but not positive definite, we can compute a factorization with half the arithmetic of the nonsymmetric case (as is the case with the Cholesky factorization), but pivoting is required to ensure numerical stability. The need for symmetric pivoting complicates the computation, and communication-efficient algorithms are not as straightforward.

3.3.4.1. Sequential case. A brief history of sequential symmetric indefinite factorizations and their communication costs is given by Ballard *et al.* (2013*b*). The most commonly used factorization (and the one implemented in LAPACK) is LDL^T, where D is block diagonal with 2×2 and 1×1 blocks, using either Bunch–Kaufman or rook pivoting. An alternative factorization, due to Aasen (1971), computes a tridiagonal matrix T instead of a block diagonal matrix D. Both factorizations use symmetric pivoting. The same lower bound for dense matrices, given in Corollaries 2.18 and 2.19, applies to both computations.

However, there exist no current communication-optimal algorithms that compute these factorizations directly. The implementations of LDL^T in

LAPACK (Anderson *et al.* 1992) (`sytrf`) and of LTL^T in Rozložník, Shklarski and Toledo (2011) can attain the lower bound for large matrices (where $n \geq M$) but fail for reasonably sized matrices (where $\sqrt{M} \leq n \leq M$). They also never attain the latency cost lower bound, and work only for the two-level memory model. It is an open problem whether there exists a communication-optimal algorithm that computes the factorizations directly.

The communication-optimal algorithm presented in Ballard *et al.* (2013*a*) first computes a factorization LTL^T, where T is a symmetric band matrix (with bandwidth $\Theta(\sqrt{M})$), and then decomposes T in a second step. This algorithm is a block algorithm, and with a block-contiguous data structure, it minimizes both words and messages in the two-level memory model.

Because the subroutines in the communication-avoiding symmetric indefinite factorization can all be performed with blocked or recursive algorithms themselves, it is possible to extend the algorithm (with matching data structure) to minimize communication costs in the multiple-level memory model. Note that the Shape-Morphing LU algorithm (Ballard *et al.* 2013*f*) is necessary to perform the panel factorization subroutine with optimal latency cost for all subsequent levels of memory.

3.3.4.2. Parallel case. While the two-stage blocked version of Aasen's algorithm in Ballard *et al.* (2013*a*) does not address the parallel case, the algorithm can be parallelized to minimize communication in the minimal memory case. This algorithm computes the factorization in two steps, first reducing the symmetric matrix to band form and then factoring the band matrix. The first step requires the use of a communication-efficient Tall Skinny LU factorization routine (as used in CALU in Section 3.3.3). The second step can be performed naively with a nonsymmetric band LU factorization (with no parallelism) with computational and communication costs that do not asymptotically exceed the costs of the first step. We leave the details of the parallelization of the reduction to band form and a more complete consideration of efficient parallel methods for factoring the band matrix to future work. No parallel algorithms are known that effectively use extra memory to reduce computation for symmetric indefinite factorizations. Using an approach similar to that of Ballard *et al.* (2013*b*) leads to a speed-up of up to 2.8× over MKL, while losing only one or two digits in the computed residual norms.

3.3.5. QR factorization

The QR factorization is commonly used for solving least-squares problems, but it has applications in many other computations such as eigenvalue and singular value factorizations and many iterative methods. While there are several approaches to computing a QR factorization, including Gram–Schmidt orthogonalization and Cholesky–QR, we focus in this section on

those approaches that use a sequence of orthogonal transformations (*i.e.*, Householder transformations or Givens rotations) because they are the most numerically stable.

Note that Cholesky–QR can be performed optimally with respect to communication costs. It involves computing A^TA, a Cholesky factorization, and a TRSM, all of which have communication-optimal algorithms (see Sections 3.3.1 and 3.3.2).

3.3.5.1. Sequential case. The history of reducing communication costs for QR factorization on sequential machines is very similar to that of LU. The algorithm in LAPACK (`geqrf`) is based on the Householder–QR algorithm (see Algorithm 3.2 of Demmel 1997, for example), computes one Householder vector per column, and also uses blocking to cast most of the computation in terms of matrix multiplication. However, this form of blocking is more complicated than in the case of LU, and is based on the ideas of Bischof and Van Loan (1987) and Schreiber and Van Loan (1989). While the blocking requires extra flops, the cost is only a lower-order term. Although the algorithm in LAPACK is much more communication-efficient than the column-by-column approach, it still does not minimize bandwidth cost for reasonably sized matrices ($\sqrt{M} < n < M$), and the column-major data layout prevents latency optimality. Note that the representation of the Q factor is compactly stored as the set of Householder vectors used to triangularize the matrix (*i.e.*, one Householder vector per column of the matrix).

Shortly after the rectangular recursive algorithms for LU were developed, a similar algorithm for QR was devised by Elmroth and Gustavson (1998). As in the case of LU, this algorithm is cache-oblivious and minimizes words moved (but not necessarily messages). It also computes one Householder vector per column. However, the algorithm performs a constant factor more flops than Householder–QR, requiring about 17% more arithmetic for tall and skinny matrices and about 30% more for square matrices. To address this issue, the authors present a hybrid algorithm which combines the ideas of the algorithm in LAPACK and the rectangular recursive one. The hybrid algorithm involves a parameter that must be chosen correctly (relative to the fast memory size) in order to minimize communication, so it is no longer cache-oblivious.

Later, another recursive algorithm for QR was developed by Frens and Wise (2003). The recursive structure of the algorithm involves splitting the matrix into quadrants instead of left and right halves, more similar to the recursive Cholesky algorithm than the previous rectangular recursive LU and QR algorithms. Because recursive calls always involve matrix quadrants, the algorithm maps perfectly to the block-recursive data layout. Indeed, with this data layout, the algorithm minimizes both words and messages and is

cache-oblivious. Unfortunately, the algorithm involves forming explicit orthogonal matrices rather than working with their compact representations, which ultimately results in a constant factor increase in the number of flops of about $3\times$. It is an open question whether this algorithm can be modified to reduce this computational overhead.

At nearly the same time as the development of the CALU algorithm and tournament pivoting, a similar blocked approach for QR factorization, called communication-avoiding QR (CAQR), was designed in Demmel *et al.* (2012). CAQR maps to the block-contiguous data layout and minimizes both words and messages in the two-level model, but because it requires the algorithmic and data layout block size to be chosen correctly, it is not cache-oblivious. Interestingly, it also requires a new representation of the Q factor arising from the Tall Skinny QR (TSQR) algorithm, which we discuss in more detail in Section 3.3.5.2. While just as compact as in the conventional Householder–QR, the new representation varies with an internal characteristic of the algorithm. The ideas behind TSQR and CAQR first appear in Golub, Plemmons and Sameh (1988), as well as Buttari, Langou, Kurzak and Dongarra (2007), Gunter and van de Geijn (2005) and Elmroth and Gustavson (1998); see Demmel *et al.* (2012) for a more complete list of references. Note that the CAQR algorithm satisfies the assumptions of Theorem 2.22 (it maintains forward progress and need not compute T matrices of dimension two or greater) and attains both the bandwidth cost lower bound stated in the theorem as well as the latency lower bound corollary.

As explained in Ballard *et al.* (2013*f*), the shape-morphing technique can be applied to the rectangular recursive QR algorithm of Elmroth and Gustavson (1998) to obtain similar results as in the case of LU factorization. Shape-morphing QR is both communication-optimal and cache-oblivious, though it suffers from the same increase in computational cost as the original rectangular recursive algorithm. Again, a hybrid version reduces the flops at the expense of losing cache-obliviousness.

We also note that rank-revealing QR is an important variant of QR factorization that is used for solving rank-deficient least-squares problems, rank determination, and other applications; see Gu and Eisenstat (1996) for more discussion of algorithms and applications. While the conventional QR with column pivoting approach suffers from high communication costs, there do exist communication-optimal algorithms for this computation. In the case of tall and skinny matrices, TSQR can be used as a preprocessing step, and conventional rank-revealing QR algorithms can be applied to the resulting triangular factor with little extra communication cost. See Demmel *et al.* (2013*c*) for an application of the tournament pivoting idea of CALU to column pivoting within rank-revealing QR of general (not so tall and skinny) matrices, and see Algorithm 1 of Ballard, Demmel and Dumitriu (2011*a*) for a randomized rank-revealing QR algorithm that requires efficient (non-rank-

revealing) QR factorization and matrix multiplication subroutines, and is needed for eigenvalue problems.

3.3.5.2. Parallel case. ScaLAPACK's QR routine (`pxgeqrf`) also minimizes bandwidth cost but not latency cost. It is a parallelization of the LAPACK algorithm, using one Householder vector per column. At nearly the same time as the development of the CALU algorithm, Demmel *et al.* (2012) developed the Communication-Avoiding QR (CAQR) algorithm, which minimizes both words and messages. The principal innovation of parallel CAQR is the factorization of a tall and skinny submatrix using only one reduction, for a cost of $O(\log P)$ messages rather than communicating once per column of the submatrix. The algorithm for tall and skinny matrices is called TSQR, and it is an important subroutine not only for general QR factorization but also many other computations: see Mohiyuddin, Hoemmen, Demmel and Yelick (2009), for example. In order to obtain this reduction, the authors abandoned the conventional scheme of computing one Householder vector per column and instead use a tree of Householder transformations. This results in a different representation of the orthogonal factor, though it has the same storage and computational requirements as the conventional scheme.

The TSQR operation is effectively a reduction operation, and in the distributed-memory parallel case, the optimal reduction tree is binary. We present TSQR in Algorithm 3.2, where we assume the number of processors is a power of 2. See also Figure 3.3, a sketch of the case of four processors. Applied to an $m \times n$ matrix such that $m/P > n$, the communication costs of the reduction are $O(n^2 \log P)$ words and $O(\log P)$ messages. In the context of CAQR, the implicit representation computed by TSQR is used to update the trailing matrix; this requires a different algorithm to the conventional approach, and is described in Demmel *et al.* (2012). It is possible to perform TSQR and recover the conventional Householder storage scheme without asymptotically increasing communication costs: see Ballard *et al.* (2014) for more details.

Tiskin (2002, 2007) has proposed generic pairwise reduction algorithms that use extra local memory to reduce communication compared to minimal-memory algorithms. These algorithms can be used for Givens-based QR factorizations. While no known lower bound exists for a trade-off between bandwidth and latency costs for QR factorization, his algorithms navigate the same trade-off as in the case of LU factorization. However, these algorithms are still considered only theoretical because they involve a (possibly large) constant factor increase in the computational cost compared to minimal-memory QR algorithms.

As in the sequential case, for communication-optimal algorithms performing rank-revealing QR factorizations, see Demmel *et al.* (2013*c*) and Ballard *et al.* (2011*a*, Algorithm 1). Again for tall and skinny matrices, a

Algorithm 3.2 $[\{Y_{i,k}\}, R] = \mathrm{TSQR}(A)$; Tall Skinny QR

Require: Number of processors P is a power of 2 and i is the proc. index
Require: A is $m \times b$, distributed in block row layout; A_i is proc. i's block
Ensure: $R = R_{0,\log p}$ is stored by proc. 0 and $Y_{i,k}$ is stored by proc. i
Ensure: $A = QR$ with Q implicitly represented by $\{Y_{i,k}\}$
1: $[Y_{i,0}, R_{i,0}] = \text{Householder–QR}(A_i)$
2: **for** $k = 1$ to $\log P$ **do** ▷ binary reduction tree for TSQR
3: **if** $i \equiv 0 \mod 2^k$ **then**
4: $j = i + 2^{k-1}$
5: Receive $R_{j,k-1}$ from processor j
6: $[Y_{i,k}, R_{i,k}] = \mathrm{QR}\left(\begin{bmatrix} R_{i,k-1} \\ R_{j,k-1} \end{bmatrix}\right)$
7: **else if** $i \equiv 2^{k-1} \mod 2^k$ **then**
8: Send $R_{i,k-1}$ to processor $i - 2^{k-1}$
9: **end if**
10: **end for**

communication-optimal rank-revealing TSQR is straightforward to derive, since a costly (conventional) rank-revealing QR can be applied after TSQR to the resulting triangular factor, with no communication cost.

We note one example of a speed-up of 13× for TSQR on a Tesla C2050 Ferma NVIDIA GPU, compared to CULA, a commercial library from EM Photonics (Anderson, Ballard, Demmel and Keutzer 2011).

3.3.6. Symmetric eigenfactorization and SVD

The processes for determining the eigenvalues and eigenvectors of a symmetric matrix and the singular values and singular vectors of a general matrix are computationally similar. In both cases, the standard approach is to reduce the matrix via two-sided orthogonal transformations (stably preserving the eigenvalues or singular values) to a condensed form. In the symmetric eigenproblem, this condensed form is a tridiagonal matrix; in the case of the SVD, the matrix is reduced to bidiagonal form. Computing the eigenvalues or singular values of these more structured matrices is much cheaper (both in terms of computation and communication) than reducing the full matrices to condensed form, so we do not consider this phase of computation. The most commonly used tridiagonal and bidiagonal solvers include MRRR, bisection/inverse iteration, divide-and-conquer, or QR iteration: see Demmel, Marques, Parlett and Vömel (2008*c*), for example. After both eigen- or singular values and vectors are computed for the condensed forms, the eigen- or singular vectors of the original matrix can be computed via a back-transformation, by applying the orthogonal matrices that transformed the dense matrix to tri- or bidiagonal.

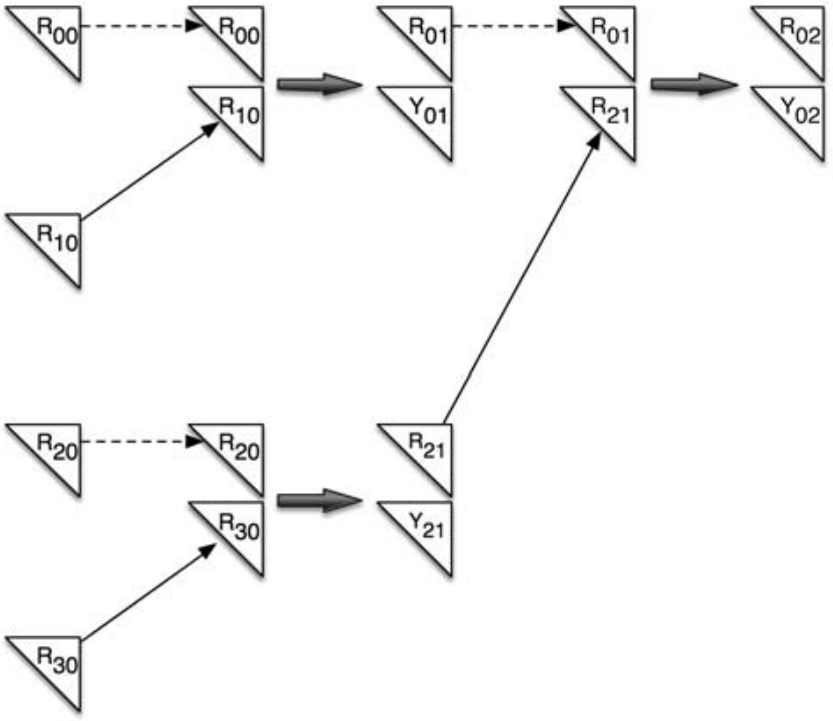


Figure 3.3. TSQR parallel algorithm with four processors, with notation matching Algorithm 3.2. Solid black arrows indicate a message, and dotted black arrows show which data do not need to be sent or received at a particular step. Grey thick arrows indicate a computation (in this case, the QR factorization of two vertically stacked R factors).

3.3.6.1. Sequential case. LAPACK's routines for computing the symmetric eigen-decomposition (`syev`) and SVD (`gesvd`) use a similar approach to the LU and QR routines, blocking the computations to cast work in terms of calls to matrix multiplication. However, because the transformations are two-sided, a constant fraction of the work is cast as BLAS-2 operations, such as matrix–vector multiplication, which are communication-inefficient. As a result, these algorithms do not minimize bandwidth or latency costs, for any matrix dimension; they require communicating $\Theta(n^3)$ words, meaning the data re-use achieved for a constant fraction of the work is as low as $O(1)$. Later work improved the constant factor in the bandwidth cost by $2\times$ (Howell *et al.* 2008), but it is still far from optimal.

Bischof, Lang and Sun (2000*a*, 2000*b*) proposed a two-step approach to reducing a symmetric matrix to tridiagonal form known as Successive Band Reduction (SBR): first reduce the dense matrix to band form and then reduce the band matrix to tridiagonal. The advantage of this approach is that the first step can be performed so that nearly all of the computation is

cast as matrix multiplication; that is, the data re-use can be $\Theta(\sqrt{M})$, which is communication-optimal. The drawback is that reducing the band matrix to tridiagonal form is a difficult task, requiring $O(n^2b)$ flops (as opposed to $O(nb^2)$ flops in the case of the symmetric indefinite linear solver) and complicated data dependences. The algorithms of Ballard, Demmel and Knight (2013*d*) perform this reduction in a communication-efficient manner.

If only eigenvalues are desired, then the two-step approach applied to a dense symmetric matrix performs the same number of flops (modulo lower-order terms) as the standard approach used in LAPACK. If eigenvectors are also desired, then the computational cost of the back-transformation phase is higher for the two-step approach by a constant factor. In terms of the communication costs, the two-step approach can be much more efficient, matching the lower bound of Corollary 2.23 and Theorem 2.24: see Ballard *et al.* (2011*a*, Section 6) for the bandwidth cost analysis. In order to minimize communication across the entire computation, the two-step approach also requires a communication-efficient Tall Skinny QR factorization (during the first step) and one of the algorithms proposed in Ballard *et al.* (2013*d*) (for the second step). All of the SBR algorithms designed for the symmetric eigenproblem can be adapted for computing the SVD.

Another communication-optimal approach is to use the spectral divide-and-conquer algorithms described in Section 3.3.7, adapted for symmetric matrices or computing the SVD. In the symmetric case, a more efficient iterative scheme is presented by Nakatsukasa and Higham (2012). These approaches require efficient QR factorization and matrix multiplication algorithms and perform a constant factor more computation than the reduction approaches.

3.3.6.2. Parallel case. As in the case of one-sided factorizations, ScaLAPACK's routines for the two-sided factorizations for the symmetric eigen-decomposition (`pxsyev`) and SVD (`pxgesvd`) minimize bandwidth cost but fail to attain the latency cost lower bound. However, by using the two-step SBR approach, we can minimize both words and messages. In the two-step approach, the first step requires an efficient parallel Tall Skinny QR factorization, such as TSQR. For the second step, the use of the parallel algorithm given in Ballard *et al.* (2013*d*) ensures that the overall algorithm achieves the lower bounds. As in the sequential case, the two-step approach requires extra computational cost (up to a logarithmic factor in P) when eigenvectors are desired. Formulating optimal, extra-memory parallel algorithms for the symmetric eigenproblem and SVD is ongoing research.

We note a speed-up of up to 6.7× for symmetric eigen-decomposition with banded A, on a ten-core Intel Westmere platform compared to PLASMA version 2.4.1 (Ballard, Demmel and Knight 2012*a*).

3.3.7. Nonsymmetric eigen-decomposition

The standard approach for computing the eigen-decomposition of a nonsymmetric matrix is similar to the symmetric case: orthogonal similarity transformations are used to reduce the dense matrix to a condensed form, from which the Schur form is computed. In the nonsymmetric case, the condensed form is an upper Hessenberg matrix (an upper triangular matrix with one nonzero subdiagonal), and QR iteration (with some variations) is typically used to annihilate the subdiagonal. In this case, the amount of data in the condensed form is asymptotically the same as the original matrix (about $n^2/2$ versus n^2), and $\Theta(n^3)$ computation is required to obtain Schur form from a Hessenberg matrix (as opposed to the symmetric case, where the data and computation involved in solving the tridiagonal eigenproblem are lower-order terms). Thus, in determining the communication cost of the overall algorithm, we cannot ignore the second phase of computation as in Section 3.3.6.

3.3.7.1. Sequential case. LAPACK's routine for the nonsymmetric eigenproblem (`geev`) takes the reduction approach and moves $O(n^3)$ words in the reduction phase and $O(n^3)$ words in the QR iteration, so it is sub-optimal both in terms of words and messages. While there have been approaches to reduce communication costs in the reduction phase in a manner similar to SBR, reducing first to band-Hessenberg and then to Hessenberg form (see Karlsson and Kågström (2011)), it is an open question whether an asymptotic reduction is possible and the lower bound of Corollary 2.23 and Theorem 2.24 is attainable. Even if the reduction phase can be done optimally, it is also an open question whether QR iteration can be done in an equally efficient manner. Some work on reducing communication for multishift QR iteration in the sequential case, based on the work of Braman, Byers and Mathias (2002*a*, 2002*b*), appears in Mohanty and Gopalan (2012). See also Granat, Kågström, Kressner and Shao (2012) and Kågström, Kressner and Shao (2012).

Because of the difficulties of the reduction approach, we consider a different approach for computing the nonsymmetric eigen-decomposition, called spectral divide-and-conquer. In this approach, the goal is to compute an orthogonal similarity transformation which transforms the original matrix into a block upper triangular matrix, thereby generating two smaller subproblems whose Schur form can be combined to compute the Schur form of the original matrix. While there are a variety of spectral divide-and-conquer methods, we focus on the one proposed in Bai, Demmel and Gu (1997*b*), adapted in Demmel, Dumitriu and Holtz (2007*a*), and further developed in Ballard *et al.* (2011*a*). This approach relies on a randomized rank-revealing QR factorization and communication-optimal algorithms for QR factorization and matrix multiplication. Under mild assumptions, the algorithm

asymptotically minimizes communication (and is cache-oblivious if the QR factorization and matrix multiplication algorithms are) but involves a (possibly large) constant factor more computation than the reduction and QR iteration approach. The algorithm can be applied to the generalized eigenproblem as well as symmetric variants and the SVD. See Ballard *et al.* (2011*a*) for full details of the algorithm and its communication costs.

Note also that to form the eigen-decomposition from Schur form requires computing the eigenvectors of a (quasi-)triangular matrix. The LAPACK routine for this computation (`trevc`) computes one eigenvector at a time and does not minimize communication. A communication-optimal, blocked algorithm is presented in Ballard *et al.* (2011*a*, Section 5).

3.3.7.2. Parallel case. For the nonsymmetric eigenproblem, ScaLAPACK's routine (`pxgeev`) minimizes neither words nor messages. As in the sequential case, it is an open problem to minimize communication using the standard approach of reduction to Hessenberg form followed by Hessenberg QR iteration. Ongoing algorithmic and implementation development on the ScaLAPACK code has been improving the communication costs and speed of convergence: see Granat *et al.* (2012) for details. These improvements, while important, do not achieve the communication lower bound.

For the purposes of minimizing communication, we consider a different approach based on spectral divide-and-conquer. As in the sequential case, by using the method of Bai *et al.* (1997*b*), Ballard *et al.* (2011*a*) and Demmel *et al.* (2007*a*), we can minimize both words and messages with the use of optimal parallel QR factorization and matrix multiplication subroutines.

Formulating parallel algorithms for the nonsymmetric eigenproblem that utilize extra memory optimally is an open problem.

3.3.8. Direct computations with sparse matrices

All of the algorithms for direct matrix computations discussed thus far in Section 3.3 have been designed for dense matrices. While the number of communication-optimal, direct algorithms for sparse matrices is very limited in comparison, we highlight three examples of algorithms that are optimal for specific sets of sparse matrices.

The first example is of an algorithm for computing all-pairs shortest paths between vertices in graph via the Floyd–Warshall method (Floyd 1962, Warshall 1962). For graphs of sufficiently low diameter, even if the original graph is sparse, the associated distance matrix fills in quickly and dense lower bounds apply. The communication-optimal parallel algorithm for this computation appears in Solomonik *et al.* (2013).

In other sparse computations, the majority of the computational work is performed on dense submatrices. This is the case for computation of

the Cholesky factorization of matrices corresponding to meshes and planar graphs. The extension of the lower bounds and the identification of a communication-optimal algorithm are given in Grigori *et al.* (2010).

The last example involves matrices and computations which are truly sparse: multiplying two sparse matrices whose sparsity structures correspond to Erdős–Rényi random graphs with expected vertex degrees constant with respect to the matrix dimension. In this case, the lower bounds of Section 2 are unattainable; however, similar proof techniques can obtain tighter lower bounds in the parallel case that match optimal algorithms. See Ballard *et al.* (2013*c*) for details.

3.4. Conclusions and future work

3.4.1. Sequential case

Sequential algorithms that are communication-optimal exist for all of the dense matrix computations discussed in this section (see Table 3.1), though much work remains to be done. In some cases, implementations of theoretically optimal algorithms have demonstrated performance improvements over previous algorithms; in others, implementations are still in progress. There are several possible future directions of algorithmic improvement: finding ways to reduce costs by constant factors or develop other theoretically optimal algorithms that might perform more efficiently in practice.

For example, it may be possible to minimize both words and messages for one-sided factorizations without relying on tournament pivoting, Householder reduction trees, or the block-Aasen algorithm by using more standard blocked algorithms (as in LAPACK) and adding a second level of blocking. This could also produce a communication-optimal LDL^T algorithm. Other open problems with respect to QR factorization include reconstructing Householder vectors from the tree representation of TSQR and modifying the algorithm of Frens and Wise (2003) to be more computationally efficient.

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

In particular, this approach relies on randomization in two ways: first, the division of the spectrum is done with a randomly chosen curve through the complex plane, and second, a randomized rank-revealing QR factorization is used to implicitly compute a factorization of a matrix product of the form $A^{-1}B$ (Ballard *et al.* 2011*a*). Also, computing the eigenvectors from Schur form requires an optimal algorithm (assuming minimal memory) which is

presented in Ballard *et al.* (2011*a*, Section 5). For sparse matrices, most cases are open, and only a few optimal algorithms exist.

3.4.2. Parallel case

Parallel algorithms that are communication-optimal exist for many, but not all, dense matrix problems. As in the sequential case, there are many constant-factor improvements possible for the algorithms discussed in this section (see Table 3.2). In particular, many implementations are in progress for demonstrating performance benefits of the new algorithms for symmetric indefinite factorizations and computing the symmetric eigen-decomposition and SVD.

Furthermore, the algorithms presented in Table 3.2 assume limitations on local memory that are often not necessary, especially in strong-scaling scenarios. Only a few of the computations listed in Table 3.2 appear in Table 3.3: many open algorithmic problems remain. Developing and optimizing extra-memory algorithms is an important area of research for developing scalable algorithms. For sparse matrices, most cases are open, and only a few optimal algorithms exist.

4. Lower bounds for Strassen-like computations

In this section we consider the communication costs of Strassen's algorithm (Strassen 1969) and similar computations. Recall that Strassen's key idea is to multiply 2×2 matrices using seven scalar multiplications instead of eight. Because $n \times n$ matrices can be divided into quadrants, Strassen's idea applies recursively. Each of the seven quadrant multiplications is computed recursively, and the computational cost of additions and subtractions of quadrants is $\Theta(n^2)$. Thus, the recurrence for the flop count is $F(n) = 7F(n/2) + \Theta(n^2)$ with base case $F(1) = 1$, which yields $F(n) = \Theta(n^{\log_2 7})$, which is asymptotically less computation than the classical algorithm.

The main results in the following section expose a wonderful fact: not only does Strassen's algorithm require less computation than the classical algorithm, but it also requires less communication. The lower bounds are lower than the bounds for the classical algorithm (Hong and Kung 1981, Irony *et al.* 2004). In both sequential and parallel cases, there now exist communication-optimal algorithms that achieve the lower bounds: see Section 5.

4.1. Expansion and communication

The computation performed by an algorithm on a given input can be modelled as a computation directed acyclic graph (CDAG): we have a vertex for each input, intermediate, and output argument, and edges according to direct dependences; for example, for the binary arithmetic operation $x = y + z$

we have directed edges from vertices corresponding to operands y and z to the vertex corresponding to x.

In the sequential case, an implementation (or schedule) determines the order of execution of the arithmetic operations, which respects the partial ordering of the CDAG. In the parallel case, an implementation determines which arithmetic operations are performed by which of the P processors as well as the ordering of local operations. This corresponds to partitioning the CDAG into P parts. Edges crossing between the various parts correspond to arguments that are in the possession of one processor but are needed by another processor and therefore relate to communication.

4.2. General framework

The proof technique described here has two main ingredients. One is a partition argument, similar to that used in Section 2 for classical algorithms in numerical linear algebra. However, we replace the use of Loomis–Whitney geometric inequality there (Lemma 2.5) with edge expansion analysis of the corresponding computation graph. Various analyses of computation graphs appear in Hong and Kung (1981) and Savage (1995) for communication cost bounds and in Lev and Valiant (1983) for bounding the arithmetic cost.

We base our analysis on the *edge expansion* of Strassen's CDAG. The edge expansion $h(G)$ of a d-regular (*i.e.*, d neighbours for each vertex) undirected graph $G = (V, E)$ is

$$h(G) \equiv \min_{U \subseteq V, |U| \leq |V|/2} \frac{|E_G(U, V \setminus U)|}{d \cdot |U|}, \tag{4.1}$$

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

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

$$h_s(G) \equiv \min_{U \subseteq V, |U| \leq s} \frac{|E_G(U, V \setminus U)|}{d \cdot |U|}. \tag{4.2}$$

For many interesting graph families (including Strassen's CDAG), $h_s(G)$ does not depend on $|V(G)|$ when s is fixed, although it may decrease when s increases.

Intuitively, the edge expansion relates the amount of data that must be accessed in order to perform a given set of computations. For a given set S of vertices (a set of arithmetic operations, of size $|S| \leq s$), the neighbours of S are data items that are necessary to perform the computation; if this data is not in fast or local memory, communication is required. Hence the edge expansion of a set S can be used to bound the communication costs.

Algorithm 4.1 Matrix multiplication: Strassen's algorithm

Require: Two $n \times n$ matrices, A and B

procedure C = MATRIX-MULTIPLICATION(A, B, n)
  if $n = 1$ **then**
    $C = A \cdot B$
  else
    Decompose A, B, C into quadrants: $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$
    $S_1 = A_{11} + A_{22}$; $T_1 = B_{11} + B_{22}$
    $S_2 = A_{21} + A_{22}$; $T_2 = B_{11}$
    $S_3 = A_{11}$; $T_3 = B_{12} - B_{22}$
    $S_4 = A_{22}$; $T_4 = B_{21} - B_{11}$
    $S_5 = A_{11} + A_{12}$; $T_5 = B_{22}$
    $S_6 = A_{21} - A_{11}$; $T_6 = B_{11} + B_{12}$
    $S_7 = A_{12} - A_{22}$; $T_7 = B_{21} + B_{22}$
    for $i = 1$ to 7 **do**
      $Q_i =$ MATRIX-MULTIPLICATION$(S_i, T_i, n/2)$
    end for
    $C_{11} = Q_1 + Q_4 - Q_5 + Q_7$
    $C_{12} = Q_3 + Q_5$
    $C_{21} = Q_2 + Q_4$
    $C_{22} = Q_1 - Q_2 + Q_3 + Q_6$
  end if
  return C
end procedure

4.3. Strassen's algorithm

The CDAG of Strassen's algorithm can be decomposed into three subgraphs: an 'encoding' of the input matrix A, an 'encoding' of the input matrix B, and 'decoding' of the $n^{\log_2 7}$ scalar multiplications to produce the output matrix C. We choose to perform an expansion analysis of the last of these subgraphs, which we denote by $\mathrm{Dec}_{\lg n} C$ for matrices of dimension n. See Algorithm 4.1 and Figure 4.1.

In Ballard, Demmel, Holtz and Schwartz (2011*c*, 2012*b*) we show the following.

Lemma 4.1. The edge expansion of $\mathrm{Dec}_k C$ is

$$h(\mathrm{Dec}_k C) = \Omega\left(\left(\frac{4}{7}\right)^k\right).$$

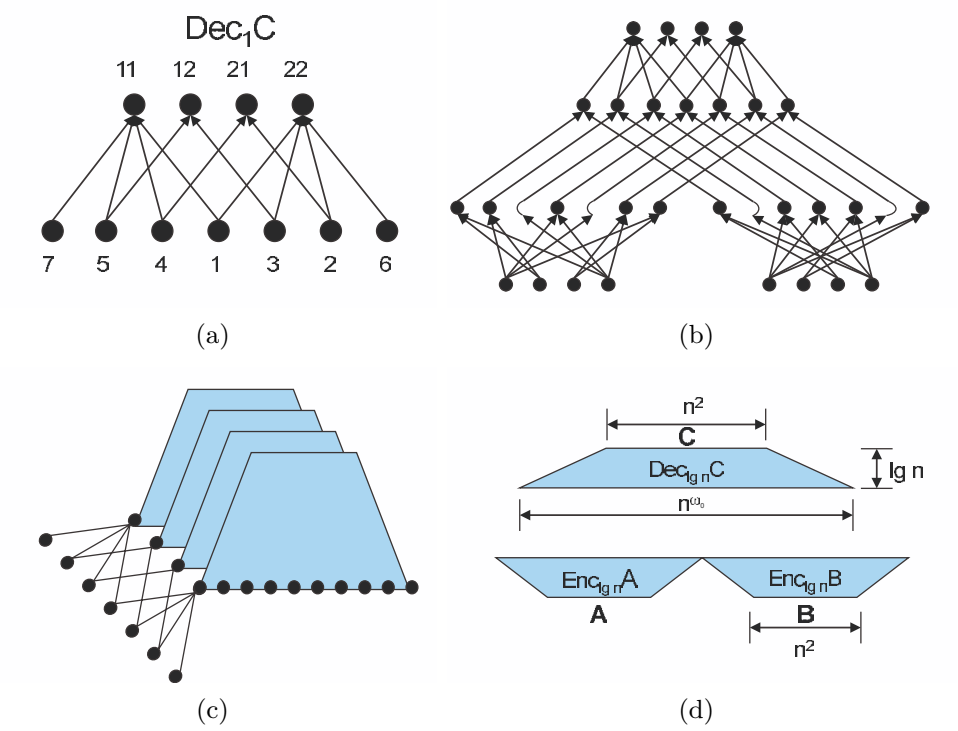


Figure 4.1. The computation graph of Strassen's algorithm (see Algorithm 4.1): (a) $\mathrm{Dec}_1 C$, (b) H_1, (c) $\mathrm{Dec}_{\lg n} C$, (d) $H_{\lg n}$.

By another argument (proof in Ballard, Demmel, Holtz and Schwartz 2012*b*) we obtain that

$$h_s(\mathrm{Dec}_{\log_2 n} C) \geq h(\mathrm{Dec}_k C),$$

where $s = \Theta(7^k)$. Combining with the partition argument we obtain the following lower bound.

Theorem 4.2 (Ballard *et al.* 2012*b*). Consider Strassen's algorithm implemented on a sequential machine with fast memory of size M. Then, for $M \leq n^2$, the bandwidth cost of Strassen's algorithm is

$$W(n, M) = \Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{\log_2 7} \cdot M\right).$$

It holds for any implementation and any known variant of Strassen's algorithm that is based on performing 2×2 matrix multiplication with seven scalar multiplications. This includes Winograd's $O(n^{\log_2 7})$ variant using 15 additions instead of 18, which is the most commonly used fast matrix multiplication algorithm in practice. This lower bound is tight, in that it is attained by the standard recursive sequential implementation of Strassen's algorithm.

The proof technique of Theorem 4.2 extends to parallel machines, yielding the following result.

Corollary 4.3 (Ballard *et al.* 2012*b*). Consider Strassen's algorithm implemented on a parallel machine with P processors, each with a local memory of size M. Then, for

$$\frac{3n^2}{P} \leq M \leq \frac{n^2}{P^{2/\log_2 7}},$$

the bandwidth cost of Strassen's algorithm is

$$W(n, P, M) = \Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{\log_2 7} \cdot \frac{M}{P}\right).$$

While Corollary 4.3 does not hold for all sizes of local memory (relative to the problem size and number of processors), the following memory-independent lower bound, proved using similar techniques (Ballard *et al.* 2012*d*), holds for all local memory sizes, though it requires separate assumptions.

Theorem 4.4 (Ballard *et al.* 2012*d*). Suppose a parallel algorithm performing Strassen's matrix multiplication load balances the computation. Then, the bandwidth cost is

$$W(n, P) = \Omega\left(\frac{n^2}{P^{2/\log_2 7}}\right).$$

Note that the bound in Corollary 4.3 dominates the one in Theorem 4.4 for $M = O(n^2/P^{2/\log_2 7})$ (which coincides with the lower bound). Thus, the tightest lower bound for parallel implementations of Strassen is the maximum of these two bounds. Similar bounds exist for classical computations: see Section 2.6.2. As in the classical case, the bound in Theorem 4.4 implies that there is a limit to how far a parallel algorithm for Strassen's matrix multiplication can strongly scale perfectly. See Table 4.1 and Ballard *et al.* (2012*d*) for more details.

4.4. Strassen-like algorithms

A Strassen-like algorithm is a recursive matrix multiplication algorithm based on a scheme for multiplying $k \times k$ matrices using q scalar multiplications for some k and $q < k^3$ (so that the algorithm performs $O(n^{\omega_0})$ flops where $\omega_0 = \log_k q$). For the latest bounds on the arithmetic complexity of matrix multiplication and references to previous bounds, see Williams (2012). For our lower bound proof to apply, we require another technical criterion for Strassen-like algorithms: the decoding graph must be connected. This class of algorithms includes many (but not all) fast matrix multiplications. For details and examples, see Ballard *et al.* (2012*b*, 2012*f*).

Table 4.1. Communication-cost lower bounds for parallel matrix multiplication and perfect strong scaling ranges; n is the matrix dimension, M is the local memory size, P is the number of processors, and ω_0 is the exponent of the arithmetic cost.

	Classical	Strassen	Strassen-like
Memory-dependent lower bound	$\Omega\left(\frac{n^3}{PM^{1/2}}\right)$	$\Omega\left(\frac{n^{\log_2 7}}{PM^{\log_2 7/2-1}}\right)$	$\Omega\left(\frac{n^{\omega_0}}{PM^{\omega_0/2-1}}\right)$
Memory-independent lower bound	$\Omega\left(\frac{n^2}{P^{2/3}}\right)$	$\Omega\left(\frac{n^2}{P^{2/\log_2 7}}\right)$	$\Omega\left(\frac{n^2}{P^{2/\omega_0}}\right)$
Perfect strong scaling range	$P = O\left(\frac{n^3}{M^{3/2}}\right)$	$P = O\left(\frac{n^{\log_2 7}}{M^{\log_2 7/2}}\right)$	$P = O\left(\frac{n^{\omega_0}}{M^{\omega_0/2}}\right)$

For Strassen-like algorithms, the statements of the communication lower bounds have the same form as Theorem 4.2, Corollary 4.3, and Theorem 4.4: replace $\log_2 7$ with ω_0 everywhere it appears! (See Table 4.1.) The proof technique follows that for Strassen's algorithm. While the bounds for the classical algorithm have the same form, replacing $\log_2 7$ with 3, the proof techniques are quite different (Hong and Kung 1981, Irony *et al.* 2004). In particular, the proof here does not apply to the classical matrix multiplication (and other three-nested-loops algorithms; recall Section 2) because the decoding graph there is not connected: for example, $C_{11} = A_{11}\cdot B_{11}+A_{12}\cdot B_{21}$ and $C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$ do not share anything.

4.5. Fast rectangular matrix multiplication

Many fast algorithms have been devised for multiplication of rectangular matrices: see Ballard *et al.* (2012*f*) for a detailed list. A fast algorithm for multiplying $m \times k$ and $k \times r$ matrices in $q < mkr$ scalar multiplications can be applied recursively to multiply $m^t \times k^t$ and $k^t \times r^t$ matrices in $O(q^t)$ flops. For such algorithms, the CDAG has a very similar structure to Strassen and Strassen-like algorithms for square multiplication in that it is composed of two encoding graphs and one decoding graph. Assuming that the decoding graph is connected, the proofs of Theorem 4.2 and Lemma 4.1 apply, where we plug in mr and q for 4 and 7, respectively. In this case, we obtain a result analogous to Theorem 4.2 which states that the communication cost of such an algorithm is given by $\Omega(q^t/M^{\log_{mr} q-1})$. If the output matrix is the largest of the three matrices (*i.e.*, $k < m$ and $k < r$), then this lower bound is attained by the natural recursive algorithm and is therefore tight. The lower bound extends to the parallel case as well, analogous to Corollary 4.3, and can be attained using the algorithmic technique of McColl and Tiskin (1999) and Ballard *et al.* (2012*e*); see Section 5.

4.6. Fast linear algebra

Fast matrix multiplication algorithms are basic building blocks in many fast algorithms in linear algebra, such as algorithms for LU, QR, and eigenvalue and singular value decompositions (Demmel *et al.* 2007*a*). Therefore, communication cost lower bounds for these algorithms can be derived from our lower bounds for fast matrix multiplication algorithms. For example, a lower bound on LU (or QR, *etc.*) follows when the fast matrix multiplication algorithm is called by the LU algorithm on sufficiently large submatrices. This is the case in the algorithms of Demmel *et al.* (2007*a*), and we can then deduce matching lower and upper bounds (Ballard, Demmel, Holtz and Schwartz 2012*b*); see Section 5.

5. Communication-optimal Strassen-like algorithms

The results of Section 4 suggest that the communication costs of fast algorithms can be substantially less than classical algorithms. While a smaller lower bound does not imply that this is the case, we discuss in this section fast algorithms that attain the lower bounds, showing that the bounds in Section 4 are tight.

5.1. Fast sequential matrix multiplication

The bandwidth cost of Strassen's algorithm where the recursion tree is traversed in the usual depth-first order, can be bounded above with the following argument. Let $W(n, M)$ be the bandwidth cost of the algorithm applied to $n \times n$ matrices on a machine with fast memory of size M. The recursion consists of computing seven subproblems and performing matrix additions, where the base case occurs when the problem fits entirely in fast memory ($cn^2 \leq M$ for a small constant $c > 3$). In the base case, we read the two input submatrices into fast memory, perform the matrix multiplication inside fast memory, and write the result into the slow memory. We thus have

$$W(n, M) \leq 7 \cdot W\left(\frac{n}{2}, M\right) + O(n^2) \quad \text{and} \quad W\left(\frac{\sqrt{M}}{c}, M\right) = O(M).$$

Thus

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

Note that this matches the lower bound stated in Theorem 4.2. In order to attain the latency lower bound as well, a careful choice of matrix layout is necessary. Morton ordering (also known as bit-interleaved layout) enables the recursive algorithm to attain the latency lower bound; see Frigo *et al.* (1999) and Wise (2000) for more details.

5.2. Fast sequential linear algebra

Demmel *et al.* (2007*a*) showed that nearly all of the fundamental problems in dense linear algebra can be solved with algorithms using asymptotically the same number of flops as matrix multiplication. Although the stability properties of fast matrix multiplication are slightly weaker than those of classical matrix multiplication, all fast matrix multiplication algorithms are stable or can be made stable, with asymptotically negligible arithmetic cost (Demmel, Dumitriu, Holtz and Kleinberg 2007*b*). Further, Demmel *et al.* (2007*a*) showed that fast linear algebra can be made stable at the expense of only a polylogarithmic (*i.e.*, polynomial in $\log n$) factor increase in cost. That is, to maintain stability, one can use polylogarithmically more bits to represent each floating point number and to compute each flop. While this increases the time to perform one flop or move one word, it does not change the number of flops computed or floating point numbers moved by the algorithm.

The bandwidth cost analysis for the algorithms presented in Demmel *et al.* (2007*a*) is given in Ballard, Demmel, Holtz and Schwartz (2012*c*). While stability and computational complexity were the main concerns in Demmel *et al.* (2007*a*), in Ballard *et al.* (2012*c*) the bandwidth cost of the linear algebra algorithms is shown to match the corresponding lower bound. To minimize latency costs, the analysis in Ballard *et al.* (2012*c*) must be combined with the shape-morphing technique of Ballard *et al.* (2013*f*).

5.3. Fast parallel matrix multiplication

Compared to classical linear algebra, much less work has been done on the parallelization of fast linear algebra algorithms (matrix multiplication or otherwise). Because there is not as rich a history of minimal-memory fast algorithms, we do not differentiate between minimal-memory and extra-memory algorithms in this section. Note that the fast algorithms that do exist are analogous to the classical extra-memory algorithms of Section 3: they can be executed with $M = O(n^2/P)$ if necessary but can also exploit extra memory at reduced communication cost if possible. While Strassen's matrix multiplication has been efficiently parallelized, both in theory (*e.g.*, McColl and Tiskin 1999, Ballard *et al.* 2012*e*) and in practice (*e.g.*, our CAPS algorithm: see Figure 5.1), there are only a few theoretical results for other fast matrix multiplication algorithms and for other linear algebra computations.

McColl and Tiskin (1999) present a parallelization of any Strassen-like matrix multiplication algorithm in the BSPRAM model that achieves a bandwidth cost of $W(n,P) = O(n^2/P^{2/\omega_0-\alpha(\omega_0-2)})$ words using local memory of size $M = O(n^2/P^{2/\omega_0+2\alpha})$, where ω_0 is the exponent of the computational cost of the algorithm and $0 \le \alpha \le 1/2 - 1/\omega_0$. This algorithm is

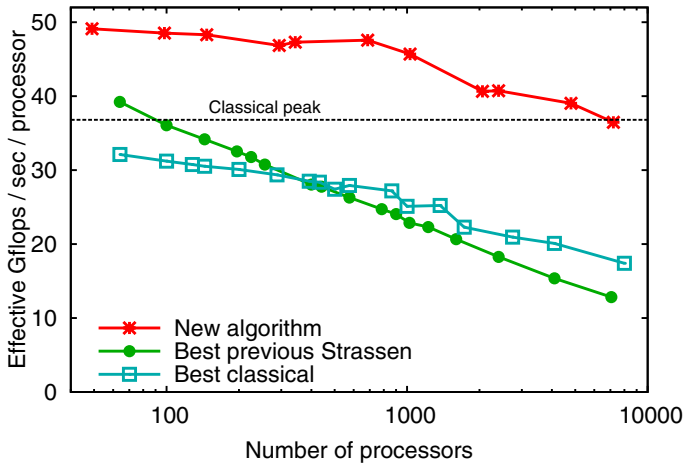


Figure 5.1. Strong-scaling performance comparison of parallel matrix multiplication algorithms on a Cray XT4. All data correspond to a fixed dimension $n = 94\,080$. The x-axis represents the number of processors P on a log scale, and the y-axis measures effective performance, or $2n^3/(P \cdot \text{time})$. The new algorithm CAPS outperforms all other known algorithms and exceeds the peak performance of the machine with respect to the classical flop count. CAPS runs 24–184% faster than the best previous Strassen-based algorithm and 51–84% faster than the best classical algorithm for this problem size.

communication-optimal for any local memory size; in particular, choosing maximum α achieves a minimal-memory algorithm, and choosing minimum α achieves the memory-independent lower bound (given by Theorem 4.4 for Strassen's algorithm). As in the classical algorithm, this occurs when M is as large as the total communication required by one processor, and the latency lower bound shrinks to one message.

A more practical version of the algorithm, CAPS (for Communication-Avoiding Parallel Strassen) with communication analysis in the distributed-memory model, as well as an implementation with performance results, is presented by Ballard *et al.* (2012*e*). For more detailed implementation description and performance results, see Lipshitz *et al.* (2012) and Lipshitz (2013). We show that the new algorithm is more efficient than any other parallel matrix multiplication algorithm of which we are aware, including those that are based on the classical algorithm and those that are based on previous parallelizations of Strassen's algorithm.

Figure 5.1 shows performance on a Cray XT4. For results on other machines, see Lipshitz *et al.* (2012). For example, running on a Cray XE6 with up to 10 000 cores, for a problem of dimension $n = 131\,712$, our new algorithm attains performance as high as 30% above the peak for classical matrix multiplication, 83% above the best classical implementation, and

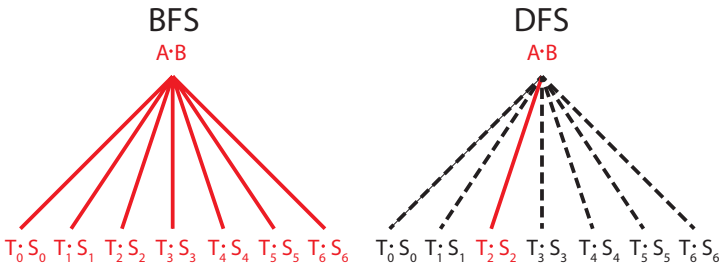


Figure 5.2. Representation of BFS and DFS. In BFS, all seven subproblems are computed at once, each on 1/7 of the processors. In DFS, the seven subproblems are computed in sequence, each using all the processors.

75% above the best previous implementation of Strassen's algorithm. Even for a small problem of dimension $n = 4704$, it attains performance 66% higher than the best classical implementation.

In order to understand the main idea behind CAPS, consider the recursion tree of Strassen's sequential algorithm. CAPS traverses it in parallel as follows. At each level of the tree, the algorithm proceeds in one of two ways. A 'breadth-first step' (BFS) divides the seven subproblems among the processors, so that 1/7 of the processors work on each subproblem independently and in parallel. A 'depth-first step' (DFS) uses all the processors on each subproblem, solving each one in sequence. See Figure 5.2 and Algorithm 5.1.

In short, BFS requires more memory but reduces communication costs while DFS requires little extra memory but is less communication-efficient. In order to minimize communication costs, the algorithm must choose an ordering of BFS and DFS that uses as much memory as possible. See Ballard *et al.* (2012*e*), Lipshitz *et al.* (2012), Demmel *et al.* (2013*a*) and Lipshitz (2013) for further details on optimizing BFS–DFS interleaving.

5.4. Fast parallel linear algebra

We note that the recursive algorithms in Section 3 that use square matrix multiplication as a subroutine can benefit from a fast matrix multiplication algorithm. In particular, the triangular solve and Cholesky decomposition algorithms of Lipshitz (2013, Section 5) and the algorithms of Tiskin (2007) attain the same computational costs as the matrix multiplication algorithm used and similarly navigate the communication–memory trade-off. However, these algorithms have only been analysed theoretically: no implementations exist yet. We leave the implementation of these known algorithms and the development of new algorithms for the rest of linear algebra to future work.

Algorithm 5.1 CAPS, in brief

Require: A, B, n, where A and B are $n \times n$ matrices
P = number of processors
Ensure: $C = A \cdot B$

▷ The dependence of the S_i on A, the T_i on B and C on the Q_i follows the Strassen or Strassen–Winograd algorithm. See Ballard *et al.* (2012*e*).

```
1: procedure C = CAPS(A, B, n, P)
2:     if enough memory then                          ▷ Do a BFS step
3:         locally compute the S_i and T_i from A and B
4:         while i = 1...7 do
5:             redistribute S_i and T_i
6:             Q_i = CAPS(S_i, T_i, n/2, P/7)
7:             redistribute Q_i
8:         end while
9:         locally compute C from all the Q_i
10:    else                                           ▷ Do a DFS step
11:        for i = 1...7 do
12:            locally compute S_i and T_i from A and B
13:            Q_i = CAPS(S_i, T_i, n/2, P)
14:            locally compute contribution of Q_i to C
15:        end for
16:    end if
17: end procedure
```

6. Sparse matrix–vector multiplication (SpMV)

We now turn our attention to *iterative* methods, as opposed to the *direct* methods that were the focus of the first half of this work. We are concerned in particular with Krylov subspace methods (KSMs), a widely used class of iterative methods for solving linear systems and eigenproblems, and the focus of Section 8. At its core, a KSM constructs a basis of a Krylov subspace (to be defined in Section 7), conventionally by performing a sequence of matrix–vector multiplications, and typically the matrix is sparse. In this section, we discuss the communication costs of sparse matrix–vector multiplication, and then in Section 7, we demonstrate communication-avoiding approaches for the more general computation of a Krylov basis. Lastly, in Section 8, we show how to apply the communication-avoiding strategies developed in the previous section to KSMs; besides matrix–vector operations, KSMs also perform vector–vector operations, for instance Gram–Schmidt orthogonalization, and we discuss opportunities to reduce these communication costs in Section 8.2.1.2.

A *matrix–vector multiplication* (MV) algorithm computes $y = Ax$ for an $m \times n$ matrix A; the 'vectors' x and y are sometimes matrices, typically with a small number of columns. The *classical MV* algorithm computes the sums-of-products $y_i = \sum_{j=1}^{n} A_{ij}x_j$ for $i \in \{1, \ldots, m\}$. A *classical sparse MV* (SpMV) algorithm is only required to compute a scalar product $A_{ij}x_j$, or to add two partial sums of products, when both operands are nonzero.

In this section we will restrict our attention to classical SpMV, but remark briefly on the myriad of 'sparse' MV algorithms that exploit more general algebraic properties of (A, x) besides zero elements. For example, a matrix with constant entries $A_{ij} = c$ has rank one, and can be applied by the algorithm $y = c(1^T x)1$, where 1 is an n-vector of all ones, with $\Theta(n)$ operations, rather than $\Theta(n^2)$. Lower bounds can be derived on the computational cost of applying a linear operator A, in terms of the number of degrees of freedom in its representation (Demmel 2013); however, general lower bounds on communication cost have so far only addressed the case of classical MV. As well as avoiding open theoretical questions, one practical justification for restricting to classical SpMV is that this is the most efficient algorithm, in terms of '+' and '·' operations, that is correct when the entries of (A, x) vary over a general semiring (Bender *et al.* 2010); see Kepner and Gilbert (2011) for applications of SpMV over semirings. Our practical justification, however, is that the communication-avoiding optimization proposed in the next section is easier to apply to classical SpMV, because the computation graph of the algorithm is encoded by the graph of A, defined next.

There are many correspondences between matrices and graphs; see, for example, Kepner and Gilbert (2011). Motivated by our our KSM applications in Section 8, we assume square, complex-valued matrices, although this discussion generalizes to rectangular matrices over a semiring. Given an $n \times n$ matrix A over $\mathbb{C}$, we say that the *graph of* A, $G(A) = (V, E)$, is the directed graph with vertices $V = \{1, \ldots, n\}$ and edges $(i, j) \in E \subseteq V \times V$ corresponding to the entries of A which are interpreted[4] as nonzero. We sometimes refer to E as the (*nonzero*) *structure* of A, and define the *number of nonzeros* by $\text{nnz} = |E|$.

While we restrict our attention to classical SpMV algorithms, we do not restrict the allowable *sparse matrix representations*, that is, the data structures used to represent the matrix A in memory. We informally categorize these representations based on whether they store the nonzero values A_{ij}, and their positions (i, j), either *explicitly* or *implicitly* (see Figure 6.1). An explicitly stored value A_{ij} or position (i, j) is represented in memory independently from other values/positions in the data structure, thus incurring

[4] Practical optimizations such as register blocking demonstrate that it can be more efficient to explicitly store and compute with some zero values (Vuduc, Demmel and Yelick 2005).

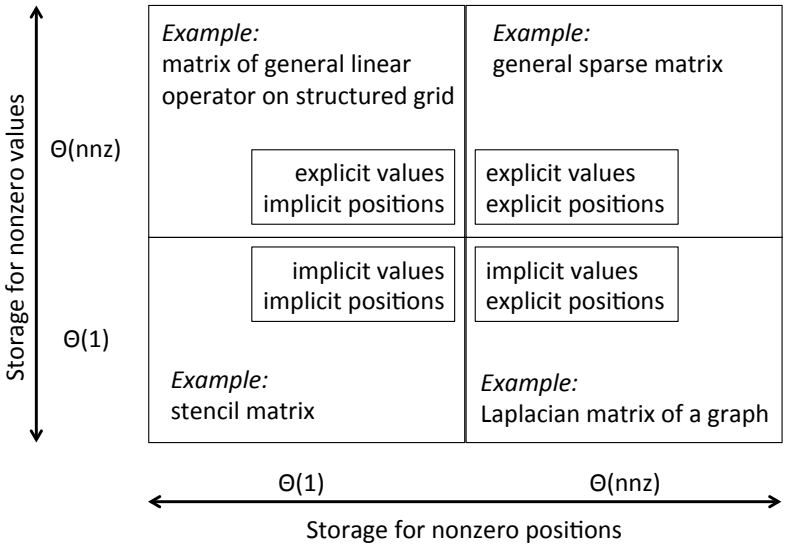


Figure 6.1. A characterization of sparse matrix representations, shown on a Cartesian plane.

a distinct memory footprint, and so we require $\Omega(\text{nnz})$ memory for the whole matrix; implicit storage means that the representation exploits additional assumptions about the matrix, which may eliminate much or all of the memory needed to store it. For example, *stencil matrices*, *e.g.*, the discrete Laplacian on a regular grid, have entries $A_{ij} = f(i, j)$, where the space required to store and compute the function f does not grow with the matrix dimension; thus stencil matrices have implicit values and positions. In some cases we can query whether $A_{ij} = 0$ in constant time/space, but A_{ij} must be stored explicitly, as for the matrix of a general linear operator on a structured mesh, where the values are independent and must be stored explicitly, but their positions can be represented implicitly due to the regular connectivity structure. In other cases the values are implicit but the structure is general and stored explicitly, such as the Laplacian matrix A of a simple (*i.e.*, undirected and without loops) graph G: for each pair of distinct vertices v_i, v_j in G, $A_{ij} = -1$ if v_i and v_j are adjacent in G, otherwise $A_{ij} = 0$, and the diagonal $A_{ii} = \deg(v_i)$, the degree of v_i. A general sparse matrix has no assumptions on its nonzero values and positions, so they must be stored explicitly. As depicted in Figure 6.1, the distinction between 'implicit' and 'explicit' is fuzzy: for example, a general Toeplitz matrix requires $\Theta(\sqrt{\text{nnz}})$ storage for its nonzero values. As well as the storage complexity, we are also interested in the run time cost of computing the SpMV; as mentioned above, we restrict our attention to classical SpMV, so the number of flops is $\Theta(\text{nnz})$, regardless of whether the storage is implicit or explicit.

To confirm our practical intuition that SpMV performance is bounded by communication costs rather than computation costs ($\Theta(\text{nnz})$), we look for communication lower bounds. We will not exploit zero entries in the input vector x, and we assume that $\text{nnz} \geq n$. We will distinguish between two cases: the 'implicit case' corresponds to the case of implicit values and implicit positions, and the 'explicit case' corresponds to the three cases where values and/or positions are explicit.

6.1. Lower bounds and optimal algorithms (sequential)

In the sequential model, a bandwidth lower bound for the explicit case follows from the fact that $W = \Omega(\text{nnz})$ words must be moved between slow and fast memory (of size M), since this many nonzero values and/or positions must be read to apply A. This assumes that the matrix storage exceeds the fast memory size by a constant factor, that is, $\text{nnz} \geq cM$ for some $c > 1$, to avoid the case when most or all A could reside in fast memory before the computation. Since we allow messages of size between 1 and M, the latency lower bounds are a factor of M smaller. This bandwidth lower bound is tight: the naive algorithm reads $O(\text{nnz})$ words, scanning A and performing the scalar multiplications and summations in the natural order. However, the latency lower bound $\Omega(\text{nnz}/M)$ is not trivially attainable, because in block transfer models such as ours (introduced in Section 2 and similar to Aggarwal and Vitter 1988), messages must consist of values stored in contiguous memory addresses. The naive algorithm will attain the latency lower bound if, for example, in each row of A the nonzeros lie in sets of M contiguous columns, or if the input vector can fit in fast memory, but attainability of $\Omega(\text{nnz}/M)$ messages is open for general sparsity structures.

In the implicit case, we can apply the results of Christ *et al.* (2013), applied there to the case of a direct N-body simulation, to obtain

$$W = \Omega\left(\max\left\{\frac{\text{nnz}}{M}, \#\text{inputs}, \#\text{outputs}\right\}\right) = \Omega\left(\max\left\{\frac{\text{nnz}}{M}, n\right\}\right).$$

We assume that the arguments of $\Omega(\cdot)$ are $\omega(M)$ to hide $O(M)$ subtractive terms due to the possibility of operands residing in fast memory before or after the execution. This bandwidth lower bound is only known to be tight in certain special cases, such as when the nonzeros of A are concentrated in dense $\Omega(M) \times \Omega(M)$ submatrices; if these submatrices are contiguous, then the latency lower bound is also attainable. However, for random sparsity patterns, it seems unlikely that $\Theta(M)$ data re-use is attainable, as suggested by results in Bender *et al.* (2010), discussed next.

Asymptotically tighter sequential latency lower bounds in the implicit case were established by Bender *et al.* (2010), under a variant of the sequential model of Hong and Kung (1981) augmented with block transfers of size $1 \leq B \leq M$ (see Aggarwal and Vitter 1988). Whereas our model is

more general, since it does not fix B and counts words and messages separately, our lower bounds for latency are weaker, since we simply divide the bandwidth lower bound by the maximum B (*i.e.*, M). Bender *et al.* derived the logarithmically larger lower bound of

$$\Omega(\min\{kn/B(1+\log_{M/B} n/(kM)), kn\})$$

messages of fixed size B for matrices with nnz $= kn$, for some $k \leq n^{1/3}$, and demonstrated a sorting-based SpMV algorithm that attains this lower bound for these matrices. This bound assumes the optimal memory layout of matrix and vectors; Bender *et al.* also give tighter attainable latency lower bounds for a wider range of k for the explicit cases where the entries of A, represented as tuples (i, j, A_{ij}), are stored in column-major layout, or in an arbitrary (*e.g.*, pessimal) layout. These results demonstrate that in many explicit cases, the latency lower bound $\Omega(\text{nnz}\,/M)$ in the preceding paragraph is attainable, provided, in the case of optimal memory layout, that the ratio $n/(kM)$ remains bounded above by a constant. These results also demonstrate a class of problems where implicit storage can give no asymptotic benefit, suggesting that attaining $\Theta(M)$ data re-use requires problems with irregular sparsity that violate their assumptions, especially $k \leq n^{1/3}$. The results of Bender *et al.* (2010) were subsequently extended to computing a sequence $y_1^T Ax_1, \ldots, y_m^T Ax_m$ of sparse bilinear forms for rectangular A (Greiner and Jacob 2010*a*), and to computing AX where A, X are both $n \times n$ and X is dense, assuming $B \leq \sqrt{M}$ (Greiner and Jacob 2010*b*). Tight lower bounds in a similar model were derived in Hupp and Jacob (2013) for a class of stencils (see, *e.g.*, Section 7.2.5).

In the explicit case, it is the $\Omega(\text{nnz})$ bandwidth lower bound that leads to the conventional wisdom that SpMV performance is communication-bound. Each nonzero A_{ij}, or its position, is only needed once, so there is no re-use of these values. Thus, if the nonzeros, or their positions, do not fit in cache, then they can be accessed at no faster rate than main memory bandwidth. More importantly, at most two floating-point operations – a multiply and, perhaps, an add – are performed for each A_{ij} read from memory. Thus the *computational intensity* – the ratio of floating-point operations to memory operations – is no greater than two, and is in fact often less. Furthermore, typical sparse matrix data structures require indirect loads and/or stores for SpMV. These indirect accesses often interfere with compiler and hardware optimizations, so that the memory operations may not even run at full memory bandwidth. These issues are discussed by Vuduc *et al.* (2005) and Williams *et al.* (2009), for example. Authors such as these have found experimentally that typical sequential SpMV implementations generally achieve no more than 10% of the machine's peak floating-point rate on commodity microprocessors. In the best case, memory bandwidth usually does not

suffice to keep all the floating-point units busy, so performance is generally bounded above by peak memory bandwidth.

Some optimizations can improve the computational intensity of SpMV for certain classes of matrices. For example, considering small, dense blocks of A as 'nonzeros' rather than the nonzero elements themselves – a technique called register blocking – helps exploit re-use of vector entries, and also reduces the number of indices needing to be read from memory. For details, see Vuduc (2003), Vuduc *et al.* (2005) and Williams *et al.* (2009), for example. For a small class of non-square matrices, a technique called cache-blocking may increase re-use in the source vector: see Nishtala, Vuduc, Demmel and Yelick (2007) for details. Reordering the sparse matrix to concentrate elements around the diagonal (*e.g.*, reverse Cuthill–McKee ordering) can improve spatial locality of the vector accesses, potentially reducing the latency cost. Nevertheless, none of these optimizations can make the performance of sequential SpMV (explicit case) comparable with the performance of dense matrix–matrix operations, whose computational intensity is $\Theta(\sqrt{M})$ (see Section 3.1).

We have raised the possibility of much greater computational intensity in the implicit case, for instance, $\Theta(M)$ when A is dense. While the results of Bender *et al.* (2010) suggest that this is rare for general (unstructured) sparse matrices, there are many practical examples of dense matrices that admit compressed representations: for instance, matrices arising from discrete Fourier transforms, or V-cycles of multigrid and fast multipole methods. Moreover, there are algorithms that apply these matrices in $o(\text{nnz})$ operations, by exploiting the recursive and/or low-rank structures in the linear operator. While special cases like the FFT are well studied (Hong and Kung 1981), general communication lower bounds for such algorithms, which do not satisfy the constraints of classical SpMV algorithms (as defined above), are open.

The sequential communication-avoiding approaches introduced in Section 7 focus primarily on the explicit case, although they also apply to the implicit case. However, in the implicit case, this approach may or may not demonstrate an asymptotic reduction in communication; in our KSM application, we will see there is still an opportunity, as explained further in Section 7.2.

6.2. Lower bounds and optimal algorithms (parallel)

Parallel communication lower bounds for SpMV require some notion of initial data layout, load balance and/or local memory capacity, in order to avoid the (communication-optimal) situation where one of the the P processors computes $y = Ax$ locally, by storing all the data, and doing no communication at all.

We define the parallel classical SpMV algorithm as one where each processor j owns a matrix $A^{(j)}$ and computes $y^{(j)} = A^{(j)}x$, where

$$A = \sum_{j=1}^{P} A^{(j)}$$

is a sum of matrices with disjoint nonzero structures. (Allowing the nonzero structures to overlap does not invalidate the lower bounds.) The vectors x, y are distributed across the P processors, and their layout, along with the splitting of A, determines the communication cost: first, zero or more entries of x are communicated, and then zero or more entries of y are computed by a reduction over the (sparse) vectors $y^{(j)}$. We assume a load-balanced parallelization among $P \geq 2$ processors, where at least two processors perform at least nnz/P flops.

If we do not constrain the layout of x and y, zero communication is always possible in the explicit case. Consider a P-way 1D block row-wise partition of A, that is, each processor is assigned a subset of the rows of A, and replicate the source vector entries on each processor whose block row has nonzeros in the corresponding columns. (This replication no more than doubles the memory requirements, since either the matrix nonzeros or their positions are explicit.) There is no communication required to accumulate y, so all communication is hidden in the replication phase. In our KSM application, we will see that such replication would incur a run time communication cost, and so is not actually free; in this application, zero communication is only attainable in special cases, such as when A has block diagonal structure. When x and y are to be partitioned across the P processors with no replication, a hypergraph model is more appropriate for modelling the communication cost, as explained shortly.

In the implicit case, the lower bound from Christ *et al.* (2013) (again for the direct N-body problem, as in the sequential case) gives

$$W = \Omega\left(\max\left\{\frac{\mathrm{nnz}/P}{M}, \#\text{inputs}, \#\text{outputs}\right\}\right) = \Omega\left(\frac{\mathrm{nnz}}{PM}\right),$$

provided $P = o(\mathrm{nnz}/M^2)$. When more processors are used, zero communication may be possible, for the same reasons as in the explicit case. This is related to the notion of perfect strong scaling (see Section 2.6.2). The latency lower bound is smaller by a factor of M. This is a natural generalization of the sequential bandwidth and latency lower bounds (implicit case), and the algorithm that can attain these lower bounds, nonzero structure permitting, is similar: tile a sufficiently large square of the iteration space $\{(i, j)\}$ into $\Theta(M) \times \Theta(M)$ subsquares, and distribute subsquares to processors; see, for instance, the communication-optimal N-body algorithms in Christ *et al.* (2013) and Driscoll *et al.* (2013).

It turns out that the communication costs for parallel SpMV without

data replication (implicit or explicit storage) can be exactly modelled by a hypergraph constructed from the computation's DAG. (Note that the matrix nonzeros never need to be communicated between processors, so we do not expect a distinction between implicit and explicit storage in this context.) In the fine-grained SpMV hypergraph model (Çatalyürek and Aykanat 2001), vertices represent matrix nonzeros (scalar multiplications) and the hyperedges are the union of the predecessor and successor sets, that is, the vertices adjacent to incoming (resp. outgoing) edges, of each vertex in the graph of A. Each vertex partition corresponds to a parallelization of the classical SpMV computations, and the induced hyperedge cut corresponds to interprocessor communication for that parallelization. By varying the metric applied to the cut, one can exactly measure communication volume (number of words moved) or synchronization (number of messages between processors) on a distributed-memory machine. Additional constraints may be applied to enforce load balance requirements. Various heuristics are applied to find approximate solutions to these NP-hard partitioning problems in practice and mature software packages are available: see, for example, Devine *et al.* (2006).

It is open whether the parallel hypergraph model can also exactly model communication in the sequential case; a comparison of the hypergraph model with the combinatorial notion of an s-partition (Hong and Kung 1981) may be one way forward. It is worth noting that hypergraph models have inspired successful approaches to reducing cache traffic in sequential SpMV (Yzelman and Bisseling 2011). It is also open how to allow for replication of the vectors in the hypergraph model; especially in the implicit case, the minimum obtained from hypergraph partitioning could be far from the optimum.

An important component of parallel SpMV is sequential SpMV. It is hard to characterize the proportion of time that parallel SpMV spends in sequential SpMV (done on each processor), relative to how much time it spends in communication between processors, because there are many different kinds of parallel processors and sparse matrices. Trade-offs between sequential and parallel communication costs in a DAG-based computation model were developed in Bilardi and Preparata (1999), and it would be valuable to extend the hypergraph model to explore these trade-offs. Nevertheless, the common experience is that even if both the computational and communication load are balanced evenly among the processors, the cost of communication is enough, in distributed-memory computations at least, to justify the expense of a complicated reordering scheme that reduces communication volume and/or the number of messages; see, for example, Wolf, Boman and Hendrickson (2008). This suggests that interprocessor communication is an important part of parallel SpMV, and that avoiding that communication could improve performance significantly.

7. Krylov basis computations

Despite all the optimizations described in Section 6, the fact remains that the computational intensity of classical SpMV, when the nonzeros or their positions are stored explicitly, is $O(1)$. So any algorithm that uses classical SpMV as a 'black box', such as conventional KSMs for solving $Ax = b$, will necessarily be communication-bound. This means that we need a different kernel than SpMV, with higher computational intensity, to have a hope of reducing communication. This section presents such a kernel, and Section 8 shows how to use it to restructure KSMs to avoid communication.

We turn our attention to computing a length-$(k+1)$ basis of the Krylov subspace,

$$\mathcal{K}_k(A, x) = \text{span}\{x, Ax, \ldots, A^k x\},$$

assembled as a matrix $K_k = [x_0, \ldots, x_k]$. More precisely, we consider the computation $x_j = Ax_{j-1}$ for $j \in \{1, \ldots, k\}$, that is, $x_j = A^j x_0$, which can also be expressed as a matrix equation,

$$A[x_0, \ldots, x_{k-1}] = [x_1, \ldots, x_k] = [x_0, \ldots, x_k] \begin{bmatrix} 0_{1,k} \\ I_{k,k} \end{bmatrix},$$

and the more general computation $x_j = p_j(A)x_0$, where p_j is a degree-j polynomial, defined by a recurrence

$$p_j(z) = \left(zp_{j-1}(z) - \sum_{i=1}^{j} h_{i,j} p_{i-1}(z) \right) \Big/ h_{j+1,j},$$

with $p_0(z) = 1$, or

$$A[x_0, \ldots, x_{k-1}] = [x_0, \ldots, x_k] H_k, \tag{7.1}$$

where the upper Hessenberg matrix H_k has entries $h_{i,j}$ and is nonzero on its subdiagonal.[5] We note that we cannot straightforwardly replace the SpMV by computing K_k in conventional KSMs, because A is multiplied by different vectors when $k > 1$. In Section 8 we exploit the fact that KSMs compute bases of the same underlying spaces $\mathcal{K}_k$, and can be reformulated by a change of basis to use K_k. We also note that numerical stability will play a role in which basis vectors of $\mathcal{K}_k$ we can accurately compute. This will be addressed in Section 8.5.1, and is the motivation for allowing the freedom of choosing the polynomials p_j.

Conventionally, K_k may be computed by a sequence of k SpMV operations, and $O(k^2)$ vector–vector operations, depending on the polynomial recurrence. On a distributed-memory parallel machine, this involves k rounds

[5] We ignore the possibility that $x_0, \ldots, x_k$ may not be linearly independent, either in exact arithmetic, or to machine precision.

of messages, one for each vector computed, in order to distribute remote vector entries, called *ghost zones*. On a sequential machine with a two-level memory hierarchy, when A is explicit[6] (see Section 6), and its storage cost sufficiently exceeds the fast memory capacity (M), the matrix A must be read $\Theta(k)$ times from slow memory to fast memory. Our goal is to reduce the parallel latency cost, and the components of the sequential bandwidth and latency cost associated with reading A, ideally by factors of $\Theta(k)$. Further, we will attempt to minimize any resulting increases in computation, memory requirements, parallel bandwidth cost, and sequential communication of the vectors, ideally keeping each bounded by small constant factors; matrices where this is possible (for the approaches in this section) are called *well-partitioned*. We will restrict our attention to 'classical' algorithms for computing K_k, called *Akx algorithms*.

The rest of this section is organized as follows. In Section 7.1 we review what is known about communication lower bounds for computing Krylov bases. Then, in Section 7.2, we define Akx algorithms, which naturally generalize classical SpMV (Section 6) to Krylov basis computations. Lastly, in Section 7.3, we will demonstrate a 'non-classical' algorithm that also computes K_k but is not realizable as a simple reorganization of k classical SpMV computations, and may have much better performance for matrices that are not well-partitioned in the sense described above but are more general.

7.1. Lower bounds

To our knowledge, tight and general communication lower bounds for computing Krylov bases (*e.g.*, K_k) are open. However, progress has been made for special classes of structured matrices. For example, if A is a stencil on an $N \times \cdots \times N$ d-dimensional mesh (see Section 7.2.5 for a precise definition), the results of Hong and Kung (1981, Corollary 7.1) can be used to show that $\Omega(kN^d/M^{1/d})$ words must be moved between slow and fast memory sequentially. The results of Christ *et al.* (2013) extend this lower bound to a class of DAGs that admit a more general type of geometric embedding. Parallel extensions of this result have been discovered more recently, for example in Scquizzato and Silvestri (2014) and Solomonik *et al.* (2014).

By ignoring data dependences between the columns of K_k, we see that a lower bound for the simpler *SpMM* computation $Y = AX$, where X is a dense matrix with a small number k of columns and Y is computed in a row-wise manner, also applies to computing K_k. Although this lower bound

[6] Sequentially, when A is implicit, the communication cost is dominated by moving vector entries, and $\Theta(k)$ communication savings may not be possible. However, sometimes only the last vector x_k, or a set of linear functionals $\{y(x_1), \ldots, y(x_k)\}$, is needed, in which case there is $O(k)$ potential communication savings.

seems quite weak, it is asymptotically attainable in some special cases, for instance when A is block diagonal and each diagonal block plus the corresponding block row of K_k fits in fast/local memory. Indeed, this block row-wise computation of K_k underpins a more general communication-avoiding approach (Akx algorithms) that we discuss in Section 7.2. We will show that for certain problems and ranges of parameters, the arithmetic, communication, and storage costs of computing K_k with certain Akx algorithms are within constant factors of those for computing AX via classical SpMM, indicating asymptotic optimality. Since these algorithms exploit redundant copies of inputs and intermediate quantities, it does not seem that the lower bound approaches in Bender *et al.* (2010) and Greiner and Jacob (2010*a*) apply; however, it is open whether such redundancy is necessary for communication savings. We hope that these approaches, as well as those of Scquizzato and Silvestri (2014) and Solomonik *et al.* (2014), will generalize and tighten the lower bounds for this problem, and motivate a more thorough exploration of the Akx design space.

7.2. Akx algorithms

In this section we summarize some of the Akx algorithms derived by Demmel, Hoemmen, Mohiyuddin and Yelick (2007*c*) (see also Demmel, Hoemmen, Mohiyuddin and Yelick 2008*b*); we refer to the subsequent works by Mohiyuddin *et al.* (2009) and Mohiyuddin (2012) for details on the implementation (we summarize their speed-up results in Figure 7.1). We will distinguish two Akx approaches: Akx, the conventional approach of repeated calls to SpMV, and CA-Akx, a potentially communication-avoiding approach.

We will first consider the parallel case and, for simplicity, a tridiagonal $n \times n$ matrix A with explicit nonzeros and positions, and using the monomial basis $x_j = p_j(A)x_0 = A^j x_0$. (Of course, KSMs, our motivating application, are typically not used for tridiagonal A.) This example suffices to illustrate the more general concept of iteration space tiling (see Section 7.4). We then extend the Akx approaches to arbitrary (square) sparse matrices and polynomial recurrences, and introduce related sequential versions. We compare Akx and CA-Akx when A is an explicitly stored *stencil on a mesh* (defined below); in these special cases, we obtain upper bounds that demonstrate asymptotic communication optimality for ranges of parameters.

While CA-Akx is a general approach, communication savings depend on nonzero structure, especially the graph of A, $G(A)$, having vertex partitions with good (small) surface-to-volume ratios, in a sense to be formalized below. In Section 7.3 we discuss an algebraic manipulation that extends the applicability of CA-Akx to matrices with dense blocks (possibly large surface-to-volume ratio), provided these blocks have low rank.

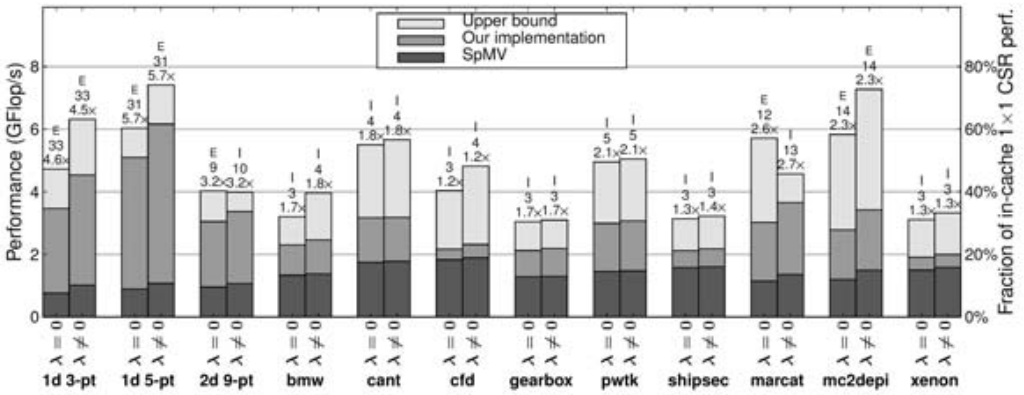
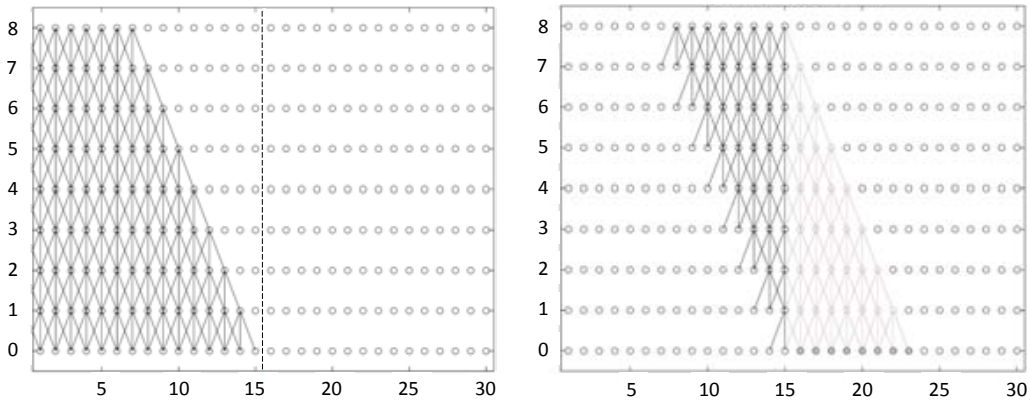


Figure 7.1. Performance of CA-Akx (labelled 'Our implementation', the medium grey bars) versus Akx (labelled 'SpMV', the dark grey bars) on an Intel Clovertown (shared-memory parallel) machine, from the implementation described by Mohiyuddin *et al.* (2009) and Mohiyuddin (2012). Medium grey bars indicate the best performance over all possible k (the chosen integer k appears above each bar); the cases $\lambda = 0$ and $\lambda \neq 0$ correspond to polynomials of the form $p_j(z) = \prod_j (z - \lambda_j)$, that is, monomials in the case $\lambda = 0$, and unweighted Newton polynomials otherwise. The Akx implementation ('SpMV') uses a highly optimized classical SpMV algorithm, and the upper bound (the light grey bars) indicates the performance predicted by scaling the SpMV performance by using a performance model based on computational intensity. The labels at the bottom correspond to a suite of test matrices. We refer to the cited works for more details, including the 'I' and 'E' designations, which describe implementation details.

7.2.1. Example: parallel CA-Akx, stencil on a (1D) mesh

Figure 7.2 helps illustrate parallel CA-Akx (for tridiagonal A) with $k = 8$, $n \geq 30$, and $P \geq 2$. Each row of circles represents the entries of $x_j = A^j x_0$, for $j \in \{0, \ldots, 8\}$, which we partition in contiguous block rows of width 15 across the processors; matrix rows are partitioned similarly. We show just the first 30 components of the vectors, owned by two processors, one to the left of the dashed vertical line and the other to the right. The diagonal and vertical lines show the dependences (arcs are implicitly directed downward): the three lines below each circle (component i of x_j) connect to the circles on which its value depends (components $i-1$, i, and $i+1$ of x_{j-1}), since A is tridiagonal.

In Akx, the conventional approach (not depicted in Figure 7.2), each processor computes its local x_j entries, exchanging their boundary elements of x_{j-1} (immediate left and right of the dashed vertical line) before each $j \in \{1, \ldots, 8\}$, to compute Ax from x, A^2x from Ax, and so on. Thus, the number of messages (along a critical path) is $\Theta(k)$. We will reorganize Akx to arrive at CA-Akx, which may perform redundant arithmetic and additional data movement but can reduce latency cost. The argument here



(a) locally computable components (b) remote dependences in CA-Akx

Figure 7.2. Communication in parallel Akx approaches for computing $[Ax, \ldots, A^k x]$ with tridiagonal A and $k = 8$.

is informal; in Section 7.2.5 we give concrete complexity results, and see that for *well-partitioned* matrices these trade-offs can be worthwhile.

In Figure 7.2(a) we show the left processor's locally computable vector elements, that is, those that can be computed without communicating with the right processor; clearly the left processor cannot compute all its vector elements without receiving information from the right. In CA-Akx, each processor first computes their locally computable elements, in an effort to overlap this computation with subsequent communication. In Figure 7.2(b), the dependences are again shown by diagonal and vertical lines below each circle, but now dependences on data owned by the right processor are shown in grey. The eight vertices of x_0 with grey fill ($x_0(16)$ to $x_0(23)$, in the bottom row and just to the right of the dashed vertical line) are sent from the right to the left processor in one message (not $\Theta(k)$), and the left processor then computes all of the vertices on paths from these vertices to local elements of x_k, which appears as a parallelogram (locally computable elements in Figure 7.2(a) have been computed at this point). Note that the grey-circled elements will be computed redundantly, by both processors; the corresponding matrix rows must be stored redundantly as well.

While typically the bandwidth cost of CA-Akx exceeds that of Akx, in special cases like this tridiagonal example the costs are the same, that is, k words are sent from each processor to its neighbours in order to compute $Ax, \ldots, A^k x$. Furthermore, one can construct examples where, given a fixed partition of $[x_0, \ldots, x_k]$, CA-Akx will asymptotically decrease the parallel bandwidth cost (versus Akx): suppose $G(A)$ is a set of $P - 1$ isolated cliques, so that all but the last two processors are responsible for their own cliques, while processors $P-1$ and P split the work of the remaining clique. The communication volume of Akx, only due to the last two processors,

increases with k, while that of CA-Akx stays fixed, and with the same cost as Akx for $k = 1$ (the ghost zones are the same size but only need to be exchanged once); for sufficiently large P, the loss of parallelism causes a negligible increase in computation cost along a critical path. While this is an unrealistic example (it can be beneficial to use fewer than p processors), it demonstrates the inherent sensitivity of the algorithmic costs to the nonzero structure and data layout; in general, we suggest using a DAG model of the Akx computation (discussed next) to explore the algorithm parameter space and develop performance models.

7.2.2. The Akx DAG

Now we formalize and generalize to other matrices the DAG $G = (V, E)$ for the computation of K_k that was sketched for a tridiagonal example in Section 7.2.1. We first form a *layer graph* of A. The $(k+1)n$ vertices $V = I \cup O$ are the union of inputs $I = \bigcup_{i=1}^{n} \{x_0(i)\}$, representing the components of the starting vector x_0, and outputs $O = \bigcup_{j=1}^{k} \bigcup_{i=1}^{n} \{x_j(i)\}$, representing the components of the computed basis vectors $\{x_1, \ldots, x_k\}$. For each nonzero A_{ij}, and for each $\ell \in \{1, \ldots, k\}$ there is an edge $(x_\ell(i), x_{\ell-1}(j)) \in E$. So far, this layer graph can be viewed equivalently as the tensor product of $G(A)$ with a simple directed path, or as the graph of the Kronecker product of A with a shift matrix. Next, we add edges to the layer graph according to the additional dependences introduced by the polynomial recurrence, that is, nonzeros in the $(k+1) \times k$ upper Hessenberg matrix H_k (7.1). Pick the smallest $u \in \{0, \ldots, k\}$ such that H_k is zero outside the band of $u+1$ diagonals $\{-1, 0, \ldots, u-1\}$, that is, H_k has upper semi-bandwidth $u-1$. (When $u \geq 1$, $u+1$ corresponds to the 'length' of the polynomial recurrence (7.1).) Then, for each $i \in \{1, \ldots, n\}$, for each $j \in \{1, \ldots, u\}$, and for each $\ell \in \{j, \ldots, k\}$, there is an edge $(x_\ell(i), x_{\ell-j}(i)) \in E$. Note that when $j = 1$, some of these edges may already exist in the layer graph, due to nonzero diagonal elements of A, and also note that we could exploit a more general nonzero structure of H_k, as well as vanishing diagonal elements of $A - h_{jj}I$, by dropping edges.

In typical Akx applications, the input x_0 and the outputs $x_1, \ldots, x_k$ are general n-vectors, and thus treated as dense. We make this assumption when discussing our Akx algorithms, which will still be correct, albeit possibly inefficient, if zeros are present: if it is known that some $x_j(i) = 0$, then we can drop all incoming and outgoing edges from the corresponding vertex, and label it both as an input and output. Thus, our analysis assumes A has no zero rows. One can imagine many other algebraic relations between elements of A and the basis vectors which could be exploited; we discuss another example below in Section 7.3.

We now discuss two simplifications we have made in this DAG construction. First, if the sparse matrices A and H_k are explicit, their accesses may

incur data movement, sequentially or in parallel, typically expressed by introducing input vertices for the nonzeros of A and H_k. We will sidestep this subtlety by associating each edge in E with a nonzero of A and/or H_k, and invoking this association in our analysis below. This also gives us the flexibility of modelling the implicit case, when A and H_k may not require any data movement because their entire data structure representations fit in fast/local memory. Second, we have hidden the details of computing the vector elements $x_j(i)$ by evaluating a polynomial recurrence: in practice, this is implemented as a sequence of scalar multiplications (elements of A and H_k by elements of $x_0, \ldots, x_{k-1}$) and a summation of these products.

Next, we generalize parallel Akx and CA-Akx from the tridiagonal example above to general sparsity structures (Section 7.2.3), and then to the sequential case (Section 7.2.4). We present general upper bounds for the algorithmic complexity of these approaches, using the models introduced in Section 1.2. Finally, in Section 7.2.5, we simplify these upper bounds in the case of $(2b+1)^d$-point stencils on d-dimensional meshes, which allows us to easily identify parameter ranges where the CA-Akx approaches can be beneficial versus the Akx approaches.

7.2.3. Parallel Akx algorithms

Given the DAG G, we can now give general parallel Akx and CA-Akx algorithms (see Algorithms 7.1–7.2), for which we define the notation below. We explicitly show point-to-point messages, but assume they incur no cost if the message is empty.

Let $G = (V, E)$ be the Akx DAG described in the previous section. Let $R(X)$ be the set of vertices reachable from $X \subseteq V$, including X, and let $R^T(X)$ be a reachable set in the transpose graph G^T (*i.e.*, G with all edges reversed). In the remainder of this section, we use lower-case p to denote the number of processors. Let $\Pi = \{I_1, \ldots, I_p\}$ be a p-way partition of $\{1, \ldots, n\}$. For $q \in \{1, \ldots, p\}$, let $P_q = \{x_j(i) : i \in I_q \wedge j \in \{1, \ldots, k\}\}$, that is, all the components of vectors assigned to processor q, and let $G(P_q)$ be its (vertex-induced) subgraph. (Recall that x_j is both a vector and a vertex subset, depending on context.) We also introduce the notation $P_{q,j} = P_q \cap x_j$, as well as $R_q(X) = R(X) \cap P_q$ and $R_{q,j}(X) = R(X) \cap P_{q,j}$.

For both algorithms, we initially distribute $[A, x_0]$ row-wise so that processor q owns rows indexed by I_q; $A(i)$ denotes the ith row of A. (In special cases, such as when A is a shift matrix, a non-row-wise vector layout may be better.) We do not claim that it is optimal to use a row-wise partition of A and a commensurate row-wise partition of $[x_0, \ldots, x_k]$. A theoretical comparison of this approach with more general layouts, for instance a column-wise or fine-grained (Çatalyürek and Aykanat 2001) partition of A, is open, and we hope to address this in future work. For the class of structured matrices we consider in this work, that is, stencils on meshes, it seems

Algorithm 7.1 Parallel Akx: code for processor q

for $j = 1$ to k **do**
  for each processor $r \neq q$ **do**
    Send vector entries $R_{q,j-1}(P_{r,j})$ to processor r
    Receive vector entries $R_{r,j-1}(P_{q,j})$ from processor r
  end for
  Compute vector entries $P_{q,j} \setminus R^T(x_{j-1} \setminus P_q)$
  Wait for receives to finish
  Compute vector entries $P_{q,j} \cap R^T(x_{j-1} \setminus P_q)$
end for

Algorithm 7.2 Parallel CA-Akx: code for processor q

for each processor $r \neq q$ **do**
  Send vector entries $R_{q,0}(P_r)$ to processor r
  Send matrix rows $\{A(i) : x_j(i) \in R_q(P_r) \setminus x_0\}$ to processor r
  Receive vector entries $R_{r,0}(P_q)$ from processor r
  Receive matrix rows $\{A(i) : x_j(i) \in R_r(P_q) \setminus x_0\}$ from processor r
end for
Compute vector entries $(R(P_q) \setminus x_0) \setminus R^T(x_0 \setminus P_q)$
Wait for receives to finish
Compute vector entries $(R(P_q) \setminus x_0) \cap R^T(x_0 \setminus P_q)$

that a column-wise CA-Akx approach could yield similar (parallel) latency savings to the row-wise version we present. However, the other algorithmic costs seem to increase more drastically with k.

We assume each processor owns a copy of H_k (nnz $= O(k^2)$). Thus, in the worst case, we must assume $k = O(\sqrt{M})$, where M is a processor's local memory size. However, in general this will not be the tightest constraint on k.

The Akx approach proceeds as in the earlier example: for each $j \in \{1, \ldots, k\}$, processor q synchronizes with other processors $r \neq q$ before computing $P_{q,j}$ (the entries of x_j for which they are responsible), by fetching $R_{r,j-1}(P_{q,j})$, its remote reachable vertices in x_{j-1} (the neighbouring ghost zones), resulting in k rounds of messages.

The CA-Akx approach is also similar to before: now, processor q fetches the non-empty sets $R_{r,0}(P_q)$ for $r \neq q$, all remote x_0 vertices reachable from the vector entries for which they are responsible, in one round of messages. In both Akx and CA-Akx we assume an initial (non-overlapping) row-wise layout of A. In CA-Akx, processors may need to communicate rows of A

when $k > 1$, unlike Akx, where matrix rows never need to be communicated or replicated. In our motivating application, however, a sequence $K_k^{(1)}, K_k^{(2)}, \ldots$ of many Krylov bases are computed with the same matrix A, so it is feasible to replicate rows of A at the beginning and amortize the cost. Moreover, if A is implicit, and its data structure representation is sufficiently small compared to M, then no communication of its entries is needed.

In general, while CA-Akx reduces latency costs by up to a factor of k, it incurs greater bandwidth, arithmetic, and storage costs than Akx, as processors may perform redundant computation to avoid communication (see Table 7.1). Whether these trade-offs are worthwhile depends on many machine and algorithm parameters, in addition to the nonzero structure; we refer to the detailed analysis and performance modelling in Demmel *et al.* (2007*c*), Hoemmen (2010) and Mohiyuddin (2012). When the matrix is 'well-partitioned', these extra costs are lower-order terms, as will be illustrated in Section 7.2.5. Note that there is no change in the parallel communication cost if general polynomials p_j are used, instead of the monomials, as in the example earlier. This is not the case for more general DAG parallelizations (*e.g.*, the 'PA2' approach in Demmel *et al.* 2007*c*), where the parallel bandwidth may increase linearly in the length of the recurrence, that is, $\Theta(u)$, in the notation of Section 7.2.2. (In our KSM applications, we consider at longest three-term recurrences, *i.e.*, $u = 2$.) It is important to note that redundant computation is not necessary to avoid communication; we leave a generalization of CA-Akx to more general partitions of V (*i.e.*, parallelizations) for future study.

Now we compare the algorithmic complexity of parallel Akx and CA-Akx and collect the results in Table 7.1; these costs are precise but opaque, and they will be evaluated in Section 7.2.5 on a model problem for illustration. For any processor $q \in \{1, \ldots, p\}$, we bound the number of arithmetic operations performed F, the number of words sent and received W, and the number of messages sent and received S. In the case of CA-Akx, we split W into two terms, the latter of which, W_A, can be amortized over successive calls with the same matrix A. The computational cost of each vector element $x_j(i)$ varies depending on the polynomial recurrence, requiring an additional $2\min\{u, j\} + 1$ flops, under our assumption from earlier that H_k has u nonzero superdiagonals, where u is minimal. The SpMV cost (the first term inside the summation) does not vary with j, and depends only on the sparsity of the rows $A(i)$; in particular, computing $A(i)x_j$ requires $2\,\mathrm{nnz}(A(i)) - 1$ flops. We also bound the memory capacity M_X needed for entries of the vectors $[x_0, \ldots, x_k]$, and M_A needed for rows of A when A is explicit and where each nonzero (value and/or position) costs c words of storage. These costs assume that, for every processor, the local memory size $M \geq M_X + M_A$, which is also a limit on the maximum message size

Table 7.1. Upper bounds on algorithmic costs of parallel Akx and CA-Akx for each processor q. Note that W for CA-Akx is split into two terms, where the second term W_A can be amortized over successive invocations with the same matrix A.

Akx	
F	$2 \cdot \sum_{x_j(i) \in P_q \setminus x_0} \text{nnz}(A(i)) + \min\{u, j\}$
W	$k\big(\sum_{r \neq q} \lvert R_{r,0}(P_{q,1}) \rvert + \lvert R_{q,0}(P_{r,1}) \rvert\big)$
S	$k\big(\lvert\{r \neq q : R_{r,0}(P_{q,1}) \neq \emptyset\}\rvert + \lvert\{r \neq q : R_{q,0}(P_{r,1}) \neq \emptyset\}\rvert\big)$
M_X	$\lvert R(P_{q,1}) \setminus P_q \rvert + (k+1)\lvert I_q \rvert$
M_A	$c \cdot \sum_{i \in I_q} \text{nnz}(A(i))$

CA-Akx	
F	$2 \cdot \sum_{x_j(i) \in R(P_q) \setminus x_0} \text{nnz}(A(i)) + \min\{u, j\}$
W	$\sum_{r \neq q} \lvert R_{r,0}(P_q) \rvert + \lvert R_{q,0}(P_r) \rvert + W_A$
W_A	$c \cdot \sum_{r \neq q} \big(\sum_{i : x_j(i) \in R_r(P_q) \setminus x_0} \text{nnz}(A(i)) + \sum_{i : x_j(i) \in R_q(P_r) \setminus x_0} \text{nnz}(A(i))\big)$
S	$\lvert\{r \neq q : R_r(P_q) \neq \emptyset\}\rvert + \lvert\{r \neq q : R_q(P_r) \neq \emptyset\}\rvert$
M_X	$\lvert R(P_q) \rvert$
M_A	$c \cdot \sum_{i : x_j(i) \in R(P_q) \setminus x_0} \text{nnz}(A(i))$

in our model. Here and later in the section, we model the case where A is implicit by taking c = 0, so, *e.g.*, $M_A = O(1)$ for both Akx and CA-Akx, and additionally in CA-Akx, we neglect W_A, A's contribution to W.

For both Akx and CA-Akx, the data layout optimization problem of minimizing the costs in Table 7.1 is most straightforwardly treated as a graph partitioning problem, but we believe a hypergraph model gives better insight into the communication costs. The hypergraph models for parallel SpMV (mentioned in Section 6.2) immediately apply to Akx, and can be easily extended to CA-Akx. For a single SpMV with a row-wise partition of A, we consider the column-net hypergraph model (Catalyurek and Aykanat 1999), where vertices represent matrix rows, and hyperedges (nets) represent matrix columns, so net j connects the vertices corresponding to rows i with a nonzero entry A_{ij}. By varying the cut-size metric, we can model both bandwidth and latency costs in SpMV, and thus for Akx. The communication cost of CA-Akx is equivalent to that of computing a single SpMV $y = A^k x$ (ignoring cancellation and assuming A has a nonzero

diagonal), with the same row-wise partition of A. So we can use the same approach for CA-Akx, except that we consider the column-net hypergraph of A^k. Of course, computing this hypergraph can be a costly operation – typically more costly than the subsequent matrix powers computation – and so the cost would have to be amortized over many calls to CA-Akx. We have explored probabilistic approaches to approximately computing the hypergraph, based on an $O(\text{nnz})$ algorithm for estimating the transitive closure (Cohen 1994, 1997), and preliminary results are promising. A simpler approximation is to construct the hypergraph of A^j for some $j < k$; interestingly, our experiments on regular $G(A)$ (*i.e.*, meshes) did not show significant improvement in partition quality increasing j past 1. Since the SpMV DAG is a subgraph of the Akx DAG, it is reasonable to suspect that good SpMV partitions might suggest good Akx partitions: this warrants further study. We note that it is harder to precisely measure the computation cost of CA-Akx than Akx (or SpMV) using the hypergraph model: for Akx, this is a simple function of the sparsity of A, while for CA-Akx it depends on the nonzero fill introduced in successive powers of A, rather than just the sparsity of A^k. Also, we recall from Section 6.2 that this approach is particular to the parallel case; it is an open problem to develop a general hypergraph model for sequential SpMV communication. However, once this model is developed, we expect that it can be applied to sequential (CA-) Akx partitioning as well.

7.2.4. Sequential Akx algorithms

The parallel Akx and CA-Akx approaches in the previous section can be modified to run on a sequential machine; we give explicit data movement instructions, and suppose that all values in fast memory are discarded after every iteration of the 'q' loop. We will also call these approaches Akx and CA-Akx (see Algorithms 7.3–7.4), and the machine model, sequential or parallel, will be clear from the context. We still assume a p-way row-wise partition of $[A, x_0, \ldots, x_k]$, but p is not related to the number of processors (now 1). Each of the p block rows of $[A, x_0, \ldots, x_k]$ is assumed to be laid out contiguously in memory (the entries of $x_1, \ldots, x_k$ are uninitialized to start with). We let $N(X)$ denote the neighbours of a vertex subset X, that is, the vertices reachable from some $x \in X$ by paths of length 1 in the Akx graph $G = (V, E)$, defined in the last section. We distinguish between the case when A has implicit nonzeros and positions ('implicit case'), and the three cases when A has explicit nonzeros and/or positions ('explicit case'). CA-Akx is intended to avoid communication in the explicit case, but may not yield an asymptotic communication savings in the implicit case.

We consider the explicit case, and say 'read A' to mean 'read the explicit values (nonzeros and/or positions) of A from slow memory to fast memory'.

Algorithm 7.3 Sequential Akx

1: **for** $j = 1$ to k **do**
2: **for** $q \in \{1, \ldots, p\}$ **do**
3: Load vector entries $N(P_{q,j})$
4: Load matrix rows $\{A(i) : i \in I_q\}$
5: Compute vector entries $P_{q,j}$
6: Store vector entries $P_{q,j}$
7: **end for**
8: **end for**

Algorithm 7.4 Sequential CA-Akx

1: **for** $q \in \{1, \ldots, p\}$ **do**
2: Load vector entries $R(P_q) \cap x_0$
3: Load matrix rows $\{A(i) : x_j(i) \in R(P_q) \setminus x_0\}$
4: Compute vector entries $R(P_q) \setminus x_0$
5: Store vector entries $P_q \setminus x_0$
6: **end for**

For now, we assume that A does not fit in fast memory of size M; we relax this assumption, below, alongside the discussion of the implicit case.

Sequential Akx is the conventional approach of performing k SpMVs in sequence. Some components of A must be read multiple times, incurring an $O(k \operatorname{nnz})$ bandwidth cost. Furthermore, poor temporal locality in the vector accesses can potentially increase their bandwidth cost from $O(kn)$ to $O(k \operatorname{nnz})$: recall from Bender *et al.* (2010) that for SpMV, in many cases the vector element communication is the dominant cost (versus reading A). Similarly to the parallel case, while we assume a row-wise execution of the SpMVs, sequential Akx can be adapted to work with any available SpMV routine.

Sequential CA-Akx[7] is essentially a sequential execution of parallel CA-Akx. That is, it computes the Krylov basis in a block row-wise fashion, attempting to reduce the cost of reading A at the price of redundant computation and additional data movement. As in the parallel case, evaluating these trade-offs depends on many machine and algorithm parameters: see Demmel *et al.* (2007*c*), Hoemmen (2010) and Mohiyuddin (2012). These works also show that it is possible to modify CA-Akx to avoid all redundant computation.

[7] Demmel *et al.* (2007*c*) made a distinction for CA-Akx between the case when $[x_0, \ldots, x_k]$ fits in fast memory but A does not, and when neither vectors nor A fit in fast memory; here, we call both cases CA-Akx.

Table 7.2. Upper bounds on algorithmic costs of sequential Akx and CA-Akx.

Akx	
F	$2 \cdot \sum_{q=1}^{p} \sum_{x_j(i) \in P_q \setminus x_0} \text{nnz}(A(i)) + \min\{u, j\}$
W	$\sum_{j=1}^{k} \sum_{q=1}^{p} \lvert N(P_{q,j}) \rvert + \lvert I_q \rvert + c \cdot \sum_{i \in I_q} \text{nnz}(A(i))$
S	$k \cdot \sum_{q=1}^{p} 2 + \lvert \{r : P_r \cap N(P_q) \neq \emptyset\} \rvert$
$M_{X,\text{slow}}, M_{X,\text{fast}}$	$(k+1)n,\ \max_{q=1}^{p} \lvert N(P_{q,k}) \rvert + \lvert I_q \rvert$
$M_{A,\text{slow}}, M_{A,\text{fast}}$	$c\,\text{nnz}(A),\ c \max_{q=1}^{p} \sum_{i \in I_q} \text{nnz}(A(i))$
CA-Akx	
F	$2 \cdot \sum_{q=1}^{p} \sum_{x_j(i) \in R(P_q) \setminus x_0} \text{nnz}(A(i)) + \min\{u, j\}$
W	$\sum_{q=1}^{p} \lvert R(P_q) \setminus x_0 \rvert + \lvert P_q \setminus x_0 \rvert + c \cdot \sum_{i : x_j(i) \in R(P_q) \setminus x_0} \text{nnz}(A(i)))$
S	$\sum_{q=1}^{p} 2 + \lvert \{r : P_r \cap R(P_q \neq \emptyset\} \rvert$
$M_{X,\text{slow}}, M_{X,\text{fast}}$	$(k+1)n,\ \max_{q=1}^{p} \lvert R(P_q) \rvert$
$M_{A,\text{slow}}, M_{A,\text{fast}}$	$c\,\text{nnz}(A),\ c \max_{q=1}^{p} \sum_{i : x_j(i) \in R(P_q) \setminus x_0} \text{nnz}(A(i))$

Now we compare the algorithmic complexity of Akx and CA-Akx (see Table 7.2); again, these costs are precise but opaque, and we will illustrate them for a model problem in the next section. We bound the number of arithmetic operations F performed, the number of words W read from and written to slow memory, and the number of messages S in which these words were moved. We also measure the total memory $M_{X,\text{slow}}, M_{A,\text{slow}}$ needed to store the vectors $[x_0, \ldots, x_k]$ and matrix A in slow memory, as well as the corresponding fast memory $M_{X,\text{fast}}, M_{A,\text{fast}}$ required. The table considers the explicit case where each nonzero (explicit value and/or position) costs c words of storage; in the implicit case, for both Akx and CA-Akx, we take $M_A = O(1)$ and neglect A's contribution to W, and, in the case of CA-Akx, to S (*i.e.*, restricting to r such that $P_{0,r} \cap R(P_q) \neq \emptyset$). The algorithms assume that the slow memory capacity is at least $M_{X,\text{slow}} + M_{A,\text{slow}}$ and likewise the fast memory size $M \geq M_{X,\text{fast}} + M_{A,\text{fast}}$. In fact, M may need to be up to twice this size, in order to reorganize data in fast memory to allow reading and writing of contiguous blocks, to attain the given latency bound.

In the implicit case, Akx may be asymptotically optimal, thus CA-Akx may not offer an asymptotic improvement: suppose that $y = Ax$ can be performed with a single read of x and a single write of y; by applying this SpMV

algorithm k times, Akx moves $2kn$ words in total, while a lower bound is the $(k+1)n$ words to read x_0 and write $[x_1, \ldots, x_k]$. However, if only the last vector x_k is desired (*e.g.*, multigrid smoothing, or the power method), then CA-Akx can reduce bandwidth and latency costs by $O(k)$, by only writing out the last vector's rows from each block, *i.e.*, $P_{q,k}$. Or, if the Akx computation is later used to compute the $k \times k$ matrix $G = [x_1, \ldots, x_k]^T [x_1, \ldots, x_k]$, then we can interleave these two computations, leaving a $k \times k$ matrix of partial sums in fast memory, and never reading/writing any of the vectors $x_1, \ldots, x_k$; a similar interleaving is possible with TSQR (Section 3.3.5). These approaches may also yield asymptotic benefits in the explicit case when nnz $= o(kn)$, and constant factor savings when nnz $= O(kn)$, for instance when A, but not all the vectors, fits in fast memory.

7.2.5. Akx for stencils on meshes

We now discuss the asymptotic complexity for the sequential and parallel Akx and CA-Akx approaches for stencils on meshes, generalizing the three-point stencil on a one-dimensional mesh in Section 7.2.1 (*i.e.*, the tridiagonal example) to a $(2b+1)^d$-point stencil on a d-dimensional mesh, with n^d vertices. More precisely, the matrices A are defined by their graphs $G(A)$: identify a vertex with each element of $\{1, \ldots, n\}^d$, and connect each pair of vertices within distance b (in the ∞-norm) by a pair of directed edges in both directions. This will allow us to simplify the general expressions in Tables 7.1–7.2 to illustrate the potential performance advantages of CA-Akx for a family of sparse matrices that models many common examples arising from discretized partial differential equations.

We first analyse the complexity in the parallel case (see Table 7.3). We now suppose there are $p = \rho^d$ processors, where $m = n/\rho$ is an integer, and the mesh vertices are partitioned so that each processor owns a contiguous $m \times \cdots \times m$ subcube. To simplify the counting, we add edges to convert the cubic mesh into a torus, by connecting opposite cube faces so that every vertex lies on exactly $2d$ simple cycles of length n (adding edges in this manner only increases the upper bounds). Because of the periodic boundaries, each processor has an equal amount of computation and communication, so we can measure the critical path by following any processor. We identify five costs for any processor: the number of arithmetic operations F, the number of words moved W, the number of messages sent/received S, and the memory needed to store the vectors M_X and the matrix entries M_A. (Recall that when A is stored explicitly, each entry costs c words of memory.)

In Table 7.3 we compare the costs of parallel Akx and CA-Akx row by row, assuming we use the same ρ^d-way row-wise partition of $[A, x_0, \ldots, x_k]$ for both, and that both approaches have sufficient memory. In general, this comparison may be unfair, because CA-Akx has higher memory requirements than Akx; however, we will restrict our attention to a range of

Table 7.3. Upper bounds on algorithmic costs of parallel Akx and CA-Akx for each processor q, for a $(2b+1)^d$-point stencil on a $n \times \cdots \times n$ d-dimensional mesh (explicit nonzero values) partitioned into $p = \rho^d$ subcubes with edge length an integer $m = n/\rho$, so each of the p processors owns a subcube. Note that W for CA-Akx is split into two terms, where the second term W_A can be amortized over successive invocations with the same matrix A.

Akx	
F	$2m^d \cdot \sum_{j=1}^{k} (2b+1)^d + \min\{u, j\}$
W	$2k(\min\{n, m+2b\}^d - m^d)$
S	$2k(\min\{\rho, 1+2\lceil b/m \rceil\}^d - 1)$
M_X	$km^d + \min\{n, m+2b\}^d$
M_A	$c(2b+1)^d m^d$
CA-Akx	
F	$2 \cdot \sum_{j=1}^{k} ((2b+1)^d + \min\{u, j\}) \min\{n, m+2b(j-1)\}^d$
W	$2(\min\{n, m+2bk\}^d - m^d) + W_A$
W_A	$2c(2b+1)^d(\min\{n, m+2b(k-1)\}^d - m^d)$
S	$2(\min\{\rho, 1+2\lceil bk/m \rceil\}^d - 1)$
M_X	$\sum_{j=0}^{k} \min\{n, m+2bj\}^d$
M_A	$c(2b+1)^d \min\{n, m+2b(k-1)\}^d$

parameters where both approaches have the same asymptotic memory footprint. In the case $k = O(m/b)$, we see a factor $\Theta(k)$ decrease in S at the cost of increasing the other four costs by a factor $O(1)$, except possibly for W in the explicit case, which increases by a factor of $O(1+cb^d)$, due to W_A, incurred by distributing matrix rows, each of size $O(cb^d)$, when $k > 1$. (As mentioned above, there is no need to distribute matrix rows in the implicit case, so $W_A = 0$ and the increase in W is also $O(1)$.) Typically, b, c, and d are not asymptotic parameters but rather small constants, so it is reasonable to assume $cb^d = O(1)$, in order to keep the growth in W bounded by a constant factor. However, in our applications, it is common to perform a sequence of Akx invocations with the same matrix, so the ghost zone rows can be distributed once, amortizing W_A over $\Omega(cb^d)$ invocations to keep the growth in W bounded by a constant factor.

The situation is similar in the sequential case (see Table 7.4). Again, we would like to compare costs componentwise (for the same row-wise partition), but this is now complicated by different (fast) memory requirements. We again suppose $k = O(m/b)$. Since $M_{X,\text{fast}}$ is greater for CA-Akx than for Akx by a factor of $O(k/u)$, then, when $u = \Theta(1)$, it is possible to pick the number of blocks p to be smaller for Akx by a factor of $O(k)$, in which case the latency costs of the two approaches are comparable. In the explicit case with $u = \Theta(1)$, we observe a $\Theta(k)$-fold decrease in the component of the bandwidth cost due to reading A – this is the main benefit – but no savings due to reading/writing vector elements. In the implicit case with $u = \Theta(1)$, the bandwidth costs are comparable for the two approaches (actually, CA-Akx is more expensive due to lower-order terms). For longer recurrences with $u = \Theta(k)$, we cannot reduce the number of blocks for Akx by more than a constant factor, so we see a $\Theta(k)$-fold decrease in latency cost for CA-Akx. Note that changing the number of blocks p does not affect the asymptotic computation or bandwidth costs for either approach, so we conclude that CA-Akx only performs a constant factor more computation than Akx. In the case $u = \Theta(k)$, we observe a $\Theta(k)$-fold decrease in bandwidth cost (in addition to the $\Theta(k)$ latency savings) for CA-Akx.

7.3. Blocking covers

In general, the CA-Akx approach is only beneficial if the additional costs due to the ghost zones grow slowly with k. Leiserson, Rao and Toledo (1997) expressed this intuition (in the sequential case) with their notion of a neighbourhood cover: in our notation, the set of subgraphs induced by $\{v_i : x_j(i) \in R(P_q)\}$ for $q \in \{1, \ldots, p\}$ form a k-neighbourhood cover of the vertices $\{v_i\}$ of $G(A)$. Informally, the efficacy of the CA-Akx approach depends on each subgraph fitting in fast memory and not overlapping too much with the others, but finding a good cover is not always possible, for instance when $G(A)$ has low diameter (or, A is not well-partitioned). Leiserson *et al.* (1997) proposed removing a subset of vertices from $G(A)$ so that the remaining subgraph has a good cover, computing the Krylov basis with the subgraph, and then updating the basis vectors to correct for the removed vertices. When the number of vertices is small, and A is fixed over sufficiently many Akx invocations, the additional costs are negligible.

We have generalized this approach in Knight, Carson and Demmel (2014), in two ways. First, we additionally address the parallel case, that is, CA-Akx. Second, we consider exploiting the more general splitting $A = D + B$, where D has a good cover and B has low rank. The trick is to exploit the identity $(D + B)^j = D^j + \sum_{i=1}^{j} D^{i-1} B A^{j-i}$ and an efficient representation $B = UV^T$. The algorithm has four phases. First, we compute $V^T[U, DU, \ldots, D^{k-2}U]$; second, $V^T[x, Dx, \ldots, D^{k-1}x]$. Both involve Akx

Table 7.4. Upper bounds on algorithmic costs of sequential Akx and CA-Akx, for a $(2b+1)^d$-point stencil on a $n \times \cdots \times n$ d-dimensional mesh (explicit nonzero values) partitioned into $p = \rho^d$ subcubes with edge length an integer $m = n/\rho$.

Akx	
F	$2n^d \cdot \sum_{j=1}^{k} (2b+1)^d + \min\{u, j\}$
W	$p \cdot \sum_{j=1}^{k} \big(\min\{n, m+2b\}^d + (1 + \max\{0, \min\{u, j\} - 1\})m^d + c(2b+1)^d m^d\big)$
S	$kp(1 + \min\{\rho, 1 + 2\lceil b/m \rceil\}^d)$
$M_{X,\text{slow}}, M_{X,\text{fast}}$	$(k+1)n^d,\ \min\{n, m+2b\}^d + (1 + \max\{0, u-1\})m^d$
$M_{A,\text{slow}}, M_{A,\text{fast}}$	$c\,\text{nnz}(A),\ c(2b+1)^d m^d$
CA-Akx	
F	$2p \cdot \sum_{j=1}^{k} ((2b+1)^d + \min\{u, j\}) \min\{n, m + 2b(j-1)\}^d$
W	$p\big(\min\{n, m+2bk\}^d + km^d + c(2b+1)^d \min\{n, m+2b(k-1)\}^d\big)$
S	$p(1 + \min\{\rho, 1 + 2\lceil bk/m \rceil\}^d)$
$M_{X,\text{slow}}, M_{X,\text{fast}}$	$\sum_{j=0}^{k} \min\{n, m+2bj\}^d$
$M_{A,\text{slow}}, M_{A,\text{fast}}$	$c\,\text{nnz}(A),\ c(2b+1)^d \min\{n, m+2b(k-1)\}^d$

computations with D, and a matrix–matrix multiplication with V^T and the Krylov basis matrix. Third, the small matrix $V^T[Ax, \ldots, A^{k-1}x]$ is computed cheaply and without communication, by exploiting the identity above. Fourth, the desired basis $Ax, \ldots, A^k x$ is obtained by interleaving another Akx computation with D and MVs of the form $U \cdot V^T A^j x$. Ideally, many Krylov bases (with A) are desired, so computing $V^T[U, DU \ldots, D^{k-2}U]$ can be performed once, and its cost amortized. However, the trade-offs are more complicated than with the Akx algorithms above, so we refer to Knight *et al.* (2014) for further discussion and performance modelling. This approach generalizes from the monomials to other polynomials, as considered earlier in this section. In addition, one can exploit the case when B has low-rank blocks; in Knight *et al.* (2014) we show how to adapt this approach to hierarchical semiseparable matrices (see, *e.g.*, Xia, Chandrasekaran, Gu and Li 2010).

We call this approach the blocking covers algorithm, after Leiserson *et al.* (1997); their approach corresponds to the special case where the columns of U are taken from an identity matrix, and V^T contains the corresponding rows of A. Another approach was described in Demmel *et al.* (2007*c*) and Hoemmen (2010), motivated by preconditioning; the algorithm from Knight *et al.* (2014) sketched above improves on this approach's costs by factors of k, but a comparison in finite precision is open. One can view the blocking covers algorithm as an extension to the Akx design space, although the reorganization changes the DAG significantly.

7.4. Discussion and related work

Our primary motivation for reorganizing Krylov basis computations (*e.g.*, $Ax, \ldots, A^k x$) is to reduce their communication costs. Whereas general, tight lower bounds are open, we suggested that, in the absence of cancellation, any classical[8] algorithm incurs computation and communication costs at least as great as computing $A \cdot X$ for some $n \times k$ matrix X. And whereas we do not have general, tight communication lower bounds for SpMV either, it is reasonable to expect that when computing $A \cdot X$ in parallel, the number of messages should be independent of k, and sequentially, the number of times A is read from slow memory should be independent of k, assuming k is not too large. We demonstrated that, for a family of stencil matrices and a range of parameters, sequential and parallel 'tiled' approaches (CA-Akx) satisfy these criteria, while conventional approaches (Akx) do not.

In practice, we care about improved performance with respect to a physical measure such as time or energy. For example, if we model the time per n-word message as $\alpha + \beta n$, and the time per arithmetic operation as γ, then we can estimate the run time by $\alpha S + \beta W + \gamma F$, as explained in the Introduction; one can estimate energy cost in a similar manner. Extensive performance modelling for the approaches discussed here, as well as other approaches that reduce redundant computation, appeared in Demmel, Hoemmen, Mohiyuddin and Yelick (2008*b*); see also the preceding technical report by Demmel *et al.* (2007*c*). A shared-memory implementation subsequently appeared in Mohiyuddin *et al.* (2009) (see the thesis by Mohiyuddin 2012 for additional details), and demonstrated speed-ups for sparse matrices from a variety of domains (see Figure 7.1). We refer to those works for details about practical implementations of the Akx approaches given here.

We gave a detailed description of the Akx dependency DAG, to emphasize that the CA-Akx and CA-Akx approaches are examples of iteration space tiling (or blocking), as are the reorganizations of dense linear algebra computations in Section 3. Many authors have studied tiling, at many

[8] In the sense of 'classical SpMV' in Section 6.

levels of abstraction. As an approach for performance optimization, tiling was possibly inspired by domain decomposition techniques for solving PDEs (Schwarz 1870), or out-of-core stencil codes (Pfeifer 1963): 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 (Irigoin and Triolet 1988). Typical tiling approaches for stencils rely on static code transformations, based on analysis of loop bounds; recent cache-aware approaches are detailed in Datta *et al.* (2008) and Dursun *et al.* (2009). A successful cache-oblivious approach that attains Hong and Kung's lower bound in the sequential case, but based on the ideal cache model (Frigo *et al.* 1999), was given in Frigo and Strumpen (2005), and subsequently generalized to shared memory parallel machines (Frigo and Strumpen 2009, Tang *et al.* 2011). However, this poses a complication for matrices with general nonzero structures, in which case the loop bounds may not be known until run time, and tiling incurs an online run time overhead (Douglas *et al.* 2000, Strout, Carter and Ferrante 2001). The hope is that this overhead can be amortized over many Krylov basis computations with the same matrix A.

8. Communication-avoiding Krylov subspace methods

Krylov subspace methods (KSMs) are a class of iterative algorithms commonly used for finding eigenvalues and eigenvectors or singular values and singular vectors of matrix A, or solving linear systems or least-squares problems $Ax = b$, when A is large and sparse. Iteration m of a KSM can be viewed as a projection process onto subspace $\mathcal{K}_m$ orthogonal to another subspace $\mathcal{L}_m$, where $\mathcal{K}_m$ is the *Krylov subspace*

$$\mathcal{K}_m(A, v) = \text{span}\{v, Av, \ldots, A^{m-1}v\}. \tag{8.1}$$

For KSMs which solve linear systems, the approximate solution is chosen from $\mathcal{K}_m$ according to some optimality criterion. This optimality criterion and the choice of $\mathcal{L}_m$ distinguish various Krylov methods. For a thorough introduction to KSMs, see Saad (2003).

In conventional KSM implementations, each iteration of the projection process consists of one or more sparse matrix–vector multiplications (SpMVs) and inner products. On modern computer architectures, these operations are both *communication-bound*: the movement of data, rather than the computation, is the limiting factor in performance. Recent efforts have thus focused on *communication-avoiding* KSMs (CA-KSMs) (Carson, Knight and Demmel 2013, Demmel *et al.* 2007*c*, Hoemmen 2010), based

on s-step KSM formulations (Chronopoulos and Gear 1989*a*, Chronopoulos and Swanson 1996, Gannon and Van Rosendale 1984, Hindmarsh and Walker 1986, Van Rosendale 1983, Sturler 1996, Toledo 1995). CA-KSMs reorder the computations in conventional KSMs to perform $O(s)$ computation steps of the algorithm for each communication step, allowing an $O(s)$ reduction in total communication cost. In practice, this can translate into significant speed-ups (Mohiyuddin *et al.* 2009). Note that in this section, the term 'conventional KSM' is used to refer to standard, that is, not communication-avoiding, implementations.

In Sections 8.2 and 8.3 below, we derive communication-avoiding variants of four KSMs. We first derive communication-avoiding variants of Arnoldi and nonsymmetric Lanczos in Section 8.2, algorithms commonly used in solving eigenvalue problems with nonsymmetric A. In Section 8.3 we derive communication-avoiding variants of the generalized minimal residual method (GMRES) and the biconjugate gradient method (BICG) for solving linear systems $Ax = b$ with nonsymmetric A, which are based on Arnoldi and Lanczos, respectively. We selected these four methods for simplicity, and to demonstrate the variety of different CA-KSMs that exist. We have also developed communication-avoiding versions of other KSMs, including communication-avoiding biconjugate gradient stabilized (CA-BICGSTAB) (Carson *et al.* 2013). We note that in the case that A is symmetric positive definite (SPD), the communication-avoiding nonsymmetric Lanczos and BICG methods reduce to give communication-avoiding variants of the more commonly used Lanczos and conjugate gradient (CG) methods, respectively.

There are two challenges to creating fast and reliable CA-KSMs. First, depending on the KSM and input matrix, the communication bottleneck may either be computing a basis of the Krylov subspace or the subsequent (dense) vector operations, such as dot products. To accelerate the former, we will use the CA-Akx kernel presented in Section 7. Hoemmen *et al.* (see, *e.g.*, Demmel *et al.* 2007*c*, Hoemmen 2010, Mohiyuddin *et al.* 2009) were the first to make use of the matrix powers kernel optimization for general sparse matrices, which reduces the communication cost by a factor of $O(s)$ for well-partitioned matrices, fusing together a sequence of s SpMV operations into one kernel invocation. This kernel is used in the CA-KSM to compute an $(s+1)$-dimensional Krylov basis $\mathcal{K}_{s+1}(A, v)$. Depending on the nonzero structure of A (more precisely, of $\{A^j\}_{j=1}^s$), this enables communication-avoidance in both serial and parallel implementations. As well as SpMV operations, KSMs also incur communication costs due to orthogonalization performed in each iteration, usually involving a series of dot products, which incur a costly global synchronization on parallel computers. For Lanczos-based KSMs, the simplest strategy is to block together dot products using a Gram matrix; this can lead to an s-fold decrease in latency in both parallel and sequential algorithms. In the case of Arnoldi-based KSMs (such as

GMRES), the orthogonalization operations can be blocked by computing a (thin) QR factorization of a tall and skinny matrix. Using the Tall Skinny QR algorithm in Demmel *et al.* (2012) (and Section 3.3.5), this leads to an s-fold decrease in latency in the parallel case, as well as an s-fold decrease in latency and bandwidth in the sequential case (Hoemmen 2010). There are many complementary approaches to reducing communication in KSMs which differ from this approach, including reducing synchronizations, overlapping communication and computation, allowing asynchronous iterations, using block Krylov methods to exploit locality, and using alternative methods such as Chebyshev Iteration. For a good overview, see Hoemmen (2010, §1.6).

Second, achieving numerical stability for CA-KSMs is more challenging than for classical KSMs. The most straightforward reorganizations of KSMs to CA-KSMs give identical results in exact arithmetic, but (depending on the input matrix) may completely fail to converge because of numerical instability. This is not surprising, since the vectors $[x, Ax, \ldots, A^k x]$ computed by the Akx kernel converge to the eigenvector of the dominant eigenvalue of A as k grows, and so form an increasingly ill-conditioned basis of the Krylov subspace. This was observed by early researchers in these methods (this related work is described in Section 8.1), who proposed using different, better-conditioned polynomial bases $[x, p_1(A)x, \ldots, p_k(A)x]$, whose computation was also discussed in Section 7. Choosing appropriate polynomials $p_j(A)$ will be discussed further in Section 8.5.1. But this is not enough to guarantee numerical stability comparable to the conventional algorithm in all cases. An additional technique to further improve numerical stability is a generalization of the residual replacement approach of Van der Vorst and Ye (1999) to improve the correlation between independent recurrences for updating the solution vector and its residual. We have shown that in many cases, the approach of combining well-conditioned polynomial bases and residual replacement strategies can make CA-KSMs reliable (Carson and Demmel 2014). Complete understanding and mitigation of numerical instabilities remains future work.

The rest of this section is organized as follows. Section 8.1 describes related work. Sections 8.2 and 8.3 describe eigenvalue problems and linear systems, respectively. Section 8.4 details speed-up results from previous work that demonstrate the performance benefits of these approaches. Section 8.5 discusses finite precision considerations. Lastly, Section 8.6 discusses preconditioning techniques for CA-KSMs.

8.1. Related work

There is a wealth of related work in the area of s-step KSMs. We highlight a few results and direct the reader to the thorough overview given in Hoemmen (2010, §1.5-6).

The first instance of an s-step method in the literature is Van Rosendale's conjugate gradient method (Van Rosendale 1983). Van Rosendale's implementation was motivated by exposing more parallelism. Chronopoulos and Gear (1989*b*) later created an s-step GMRES method with the goal of exposing more parallel optimizations. With a similar goal, Kim and Chronopoulos (1992) derived an s-step nonsymmetric Lanczos method. Walker (1988) used s-step bases as a method for improving stability in GMRES by replacing the Modified Gram–Schmidt orthogonalization process with Householder QR.

The above studies found that convergence often could not be guaranteed for $s > 5$ using the (inherently unstable) monomial basis for construction of the Krylov subspaces (see Section 7). This motivated research into the use of other, more well-conditioned bases for the Krylov subspace.

Hindmarsh and Walker (1986) used a scaled (normalized) monomial basis to improve convergence, but only saw minimal improvement. Joubert and Carey (1992) implemented a scaled and shifted Chebyshev basis, which provided more accurate results. Bai, Hu and Reichel (1994) also saw improved convergence using a Newton basis. Philippe and Reichel (2012) have demonstrated that good basis parameters can be computed inexpensively and dynamically updated throughout the iteration. The construction of other bases for the Krylov subspace will be covered more thoroughly in Section 8.5.1.

8.2. Eigenvalue problems

In this subsection, we derive communication-avoiding variants of two fundamental Krylov subspace methods for solving eigenvalue problems with nonsymmetric A. In Section 8.2.1 we derive communication-avoiding Arnoldi (CA-Arnoldi), and communication-avoiding nonsymmetric Lanczos (CA-BIOC) can be found in Section 8.2.2. These two methods were selected for their simplicity, and because they form the fundamental basis of many other CA-Krylov methods, including CA-GMRES (Section 8.3.1) and CA-BICG (Section 8.3.2).

8.2.1. Arnoldi

Arnoldi begins with an $n \times n$ matrix A and an $n \times 1$ starting vector $v \neq 0$. After s steps, assuming no breakdown occurs, the method constructs an $(s+1) \times s$ nonsingular upper Hessenberg matrix $\underline{H}$ such that

$$AQ = \underline{Q}\underline{H}, \tag{8.2}$$

in which $\underline{Q}$ is an $n \times (s+1)$ orthonormal matrix

$$\underline{Q} = [Q, q_{s+1}] = [q_1, \ldots, q_s, q_{s+1}]$$

Algorithm 8.1 Arnoldi

Require: $n \times n$ matrix A, and length n starting vector v
Output: Matrices Q and $\underline{H}$ satisfying (8.2)

```
β = ‖v‖_2, q_1 = v/β
for j = 1 to s do
    w_j = A q_j
    for i = 1 to j do
        h_ij = ⟨w, q_i⟩
        w_j = w_j − h_ij q_i
    end for
    h_{j+1,j} = ‖w_j‖_2
    q_{j+1} = w_j / h_{j+1,j}
end for
```

(with $q_1 = v/\|v\|_2$) whose columns comprise a basis of the Krylov subspace

$$\mathcal{K}_{s+1}(A, v) = \text{span}\{v, Av, A^2v, \ldots, A^sv\}.$$

We do not indicate s when we write Q or $\underline{H}$, because s is usually understood from the context. We write $h_{ij} = \underline{H}(i, j)$, and we write H for the principal $s \times s$ submatrix of $\underline{H}$, so that

$$\underline{H} = \begin{pmatrix} H \\ 0, \ldots, 0, h_{s+1,s} \end{pmatrix}.$$

Depending on the context, an underline under a letter representing a matrix means either 'add one more column to the right side' (*e.g.*, $\underline{Q}$) or 'add one more row to the bottom' (*e.g.*, $\underline{H}$) of the matrix.

Algorithm 8.1 shows the usual formulation of Arnoldi iteration. For simplicity, we assume no breakdown occurs. It uses Modified Gram–Schmidt (MGS) to orthogonalize successive basis vectors. There are other forms of Arnoldi iteration, which use other orthogonalization methods. However, MGS-based Arnoldi (Algorithm 8.1) is most often employed in practice, either by itself (usually with restarting, to be handled in Section 8.2.1.2) or as the inner loop of a method such as Implicitly Restarted Arnoldi (Sorensen 1992).

The key operations of Arnoldi iteration are the computation of the upper Hessenberg matrix $\underline{H}$ and the orthonormal basis vectors $q_1, \ldots, q_{s+1}$. (We use the letter 'q', rather than the customary 'v', because q suggests orthonormality by recalling the QR factorization.) The upper Hessenberg matrix $H = Q^TAQ$ is the projection of the matrix A onto the subspace $\text{span}\{q_1, \ldots, q_s\}$. Various operations on $\underline{H}$ yield information for solving linear systems or eigenvalue problems involving A. For example, the Arnoldi

method finds approximations for the eigenvalues of A using the eigenvalues of H (the *Ritz values*). GMRES solves $Ax = b$ approximately by starting with an initial guess x_0, performing Algorithm 8.1 with $r = b - Ax_0$, and then solving the least-squares problem $\min_y \|\underline{H}y - \beta e_1\|_2$ to obtain coefficients y for the approximate solution x_s in terms of the basis vectors $q_1, \ldots, q_s$. The Full Orthogonalization Method (FOM) (see, *e.g.*, Saad 2003) gets coefficients y for the approximate solution by solving the linear system $Hy = \beta e_1$.

If $h_{j+1,j} = 0$ in Algorithm 8.1, Arnoldi breaks down. In exact arithmetic, this only occurs when the smallest j has been reached such that a degree $j-1$ polynomial p_{j-1} exists with $p_{j-1}(A)v = 0$. The Krylov subspace basis cannot be made any larger, given the starting vector v. This is called a *lucky breakdown*, for two reasons: when solving eigenvalue problems, the set of eigenvalues of H equals a subset of the eigenvalues of A, and when solving linear systems, the current approximate solution equals the exact solution. For simplicity, we assume the starting vector v is not deficient in any of the desired eigenvectors, and no breakdown occurs; in practice, methods used in conventional KSMs for handling breakdown can be extended to CA-KSMs.

The Arnoldi process requires us to store all the basis vectors and do a full orthogonalization at each step. If s iterations are performed, then memory requirements scale as a multiple of s, and the computational expense of the vector operations scales quadratically with s. Letting s grow until the algorithm converges to the desired accuracy may therefore require too much memory or too much computation. As a result, the Arnoldi process may be *restarted* by forgetting $\underline{H}$ and all basis vectors except q_{s+1}, making the last (or best, where 'best' means the residual vector with the smallest norm) residual vector the new starting vector, and beginning the iteration over again. That would entail enclosing Algorithm 8.1 in an outer loop, a simple step which we do not show. We then say that the *restart length* is s, and that Algorithm 8.1 represents one *restart cycle*.

Restarting bounds the memory requirements by a multiple of s. However, it causes a loss of information stored in the discarded basis vectors and $\underline{H}$ matrix. This loss can adversely affect how fast the iteration converges, for both linear systems and eigenvalue problems. Picking the right restart length involves a trade-off between convergence rate and computational cost, and is constrained by memory capacity.

We will first present a simple communication-avoiding version of Arnoldi iteration, CA-Arnoldi(s). In this variant, the blocking factor s is the same as the restart length. We begin by showing how to rearrange Arnoldi so as to complete s steps of a single restart cycle with the same communication requirements as a single step of Algorithm 8.1. The basic idea is to use the matrix powers kernel to generate s basis vectors, and then orthogonalize them all at once using TSQR.

The requirement of restarting every s steps is not intrinsic to CA-Arnoldi. In the subsequent section, we present CA-Arnoldi(s,t), where s and the restart length $s \cdot t$ can be chosen independently (*i.e.*, the restart length can be chosen to meet constraints on convergence rate, memory capacity, *etc.*). This approach uses a combination of TSQR and block Gram–Schmidt orthogonalization (BGS) to avoid communication in the dense vector operations.

8.2.1.1. CA-Arnoldi(s). Walker (1988) developed a version of GMRES that we consider a precursor to the CA-Arnoldi(s) algorithm. The usual implementation of Arnoldi (Algorithm 8.1) generates the unitary basis vectors one at a time and the upper Hessenberg matrix one column at a time, using Modified Gram–Schmidt orthogonalization. Walker wanted to use Householder–QR instead to orthogonalize the basis vectors, since Householder–QR produces more orthogonal vectors than MGS in finite precision arithmetic.[9] However, conventional Householder–QR requires that all the vectors to orthogonalize be available at once. Algorithm 8.1 only generates one basis vector at a time, and cannot generate another until the current one has been orthogonalized against all the previous basis vectors. Walker dealt with this by first generating $s+1$ vectors, which form a basis for the Krylov subspace, and then computing their QR factorization. From the resulting R factor and knowledge about the basis, he could reconstruct the upper Hessenberg matrix $\underline{H}$ and use that to solve the least-squares problem underlying GMRES. This performed the work of s steps of GMRES. Then, he restarted GMRES and repeated the process until convergence. Walker did not use a matrix powers kernel to compute the basis vectors, nor did he have a communication-avoiding QR factorization. Bai *et al.* (1994) based their Newton-basis GMRES on Walker's Householder GMRES, improving numerical stability by using a different s-step basis.

We summarize Walker's algorithm in our own, more general notation. Given a starting vector v, the algorithm first computes the vectors $v_1 = v$, $v_2 = Av$, $v_3 = A^2 v, \ldots, v_{s+1} = A^s v$. We write

$$\underline{\mathcal{V}} = [\mathcal{V}, v_{s+1}] = [v_1, \ldots, v_s, v_{s+1}].$$

These vectors form a monomial basis for the Krylov subspace $\mathcal{K}_{s+1}(A, r)$. Note that different polynomials (besides monomials $A, A^2, \ldots, A^s$) can be used for constructing a basis for the Krylov subspace: see Section 8.5.1. Throughout this section, we use calligraphic letters such as $\mathcal{V}$ to denote the generated s-step bases for the required Krylov subspace(s) (except for

[9] This turns out to matter more for Arnoldi than for GMRES. The loss of orthogonality in finite precision arithmetic due to using MGS does not adversely affect the accuracy of solving linear systems with GMRES: see Greenbaum, Rozložník and Strakoš (1997). However, at the time of Walker's work this was not known.

$\mathcal{K}_m$, which always denotes a Krylov subspace of size m). For any choice of polynomials, the computation of $\underline{\mathcal{V}}$ can be written in matrix form as

$$A\mathcal{V} = \underline{\mathcal{V}}\,\underline{B}, \tag{8.3}$$

where $\underline{B}$ is an $(s+1) \times s$ *change of basis* matrix. Note that for the monomial basis, $\underline{B} = [e_2, e_3, \ldots, e_{s+1}]$.

Note the similarity between (8.3) and the Arnoldi relation (8.2); we will use this below to derive the CA-Arnoldi(s) algorithm. We denote by B the $s \times s$ principal submatrix of $\underline{B}$, so that

$$\underline{B} = \begin{pmatrix} B \\ 0, \ldots, 0, \underline{B}(s+1, s) \end{pmatrix}.$$

Note that we assume that the QR factorization of $\underline{\mathcal{V}}$ leaves the unit starting vector unchanged. Some QR factorizations may give the basis vectors q_j a different unitary scaling than the basis vectors computed by conventional Arnoldi. Similarly, the upper Hessenberg matrices may differ by a unitary diagonal similarity transform. This does not affect Ritz value calculations, but it will affect the least-squares problem which GMRES solves in order to get the approximate solution to $Ax = b$ (to be covered in Section 8.3.1). For CA-GMRES, the only scaling factor that matters is that of the first basis vector q_1. Furthermore, if CA-GMRES uses a QR factorization under which the direction of the first column is invariant, then CA-GMRES computes the same approximate solution in exact arithmetic as conventional GMRES at all iterations. Otherwise, one additional inner product will be necessary in order to compute the change in direction of q_1; see Hoemmen (2010).

We write the QR factorization of $\underline{\mathcal{V}}$: $\underline{Q}\,\underline{R} = \underline{\mathcal{V}}$ and $QR = \mathcal{V}$. Here, we use the 'thin' QR factorization in which $\underline{R}$ is $(s+1) \times (s+1)$ and R is the $s \times s$ leading principal submatrix of R. Then, we can use $\underline{R}$ and (8.3) to reconstruct the Arnoldi relation of (8.2):

$$\begin{aligned} AQ &= \underline{Q}\,\underline{H}, \\ AQR &= \underline{Q}\,\underline{R}\,\underline{R}^{-1}\,\underline{H}R, \\ A\mathcal{V} &= \underline{\mathcal{V}}\,\underline{R}^{-1}\,\underline{H}R. \end{aligned}$$

This means that if $A\mathcal{V} = \underline{\mathcal{V}}\,\underline{B}$ for some $(s+1) \times s$ matrix $\underline{B}$, we have

$$\underline{H} = \underline{R}\,\underline{B}R^{-1}. \tag{8.4}$$

(For an analogous derivation in the case of Lanczos iteration, see the beginning of Meurant (2006).) Naturally one can exploit the structures of $\underline{B}$, $\underline{R}$, and $\underline{R}^{-1}$ in order to calculate this expression more quickly, as shown below.

Algorithm 8.2 shows our CA-Arnoldi(s) algorithm (which again assumes no breakdown). It restarts after every s steps. In line 6 of the algorithm, we can exploit the structure of $\underline{B}$ and $\underline{R}$ in order to reduce the cost of

Algorithm 8.2 CA-Arnoldi(s)

Require: $n \times n$ matrix A and length n starting vector v
Output: Matrices $\underline{Q}$ and $\underline{H}$ satisfying (8.2)
1: $\beta = \|v\|_2$, $q_1 = v/\beta$
2: **while** we want to restart **do**
3: Fix the $(s+1) \times s$ basis matrix $\underline{B}$
4: Compute $\underline{\mathcal{V}}$ via CA-Akx (see Section 7)
5: Compute the QR factorization $\underline{\mathcal{V}} = \underline{Q}\underline{R}$ via TSQR
6: $\underline{H} = \underline{R}\underline{B}R^{-1}$ (equation (8.5))
7: **if** we want to restart **then**
8: Set $v = q_{s+1}$ (the last column of $\underline{Q}$) and $\beta = 1$
9: Optionally change $\underline{B}$ based on eigenvalue approximations gathered from $\underline{H}$
10: **end if**
11: **end while**

computing $\underline{H}$. If we break down $\underline{R}$ into

$$\underline{R} = \begin{pmatrix} R & z \\ 0_{1,s} & \rho \end{pmatrix},$$

where R is $s \times s$, z is $s \times 1$, and ρ is a scalar, and break down $\underline{B}$ into

$$\underline{B} = \begin{pmatrix} B \\ 0, \ldots, 0, b \end{pmatrix},$$

where B is $s \times s$ and $b = \underline{B}(s+1, s)$, then

$$\begin{aligned} \underline{H} = \underline{R}\underline{B}R^{-1} &= \begin{pmatrix} R & z \\ 0_{1,s} & \rho \end{pmatrix} \begin{pmatrix} B \\ 0, \ldots, 0, b \end{pmatrix} R^{-1} \\ &= \begin{pmatrix} RBR^{-1} + b \cdot ze_s^T R^{-1} \\ \rho b \cdot e_s^T R^{-1} \end{pmatrix}. \end{aligned}$$

If we further let $\tilde{\rho} = R(s, s)$, then $e_s^T R^{-1} = \tilde{\rho}^{-1}$ and therefore

$$\underline{H} = \begin{pmatrix} RBR^{-1} + \tilde{\rho}^{-1} b \cdot ze_s^T \\ \tilde{\rho}^{-1} \rho b \cdot e_s^T \end{pmatrix} \equiv \begin{pmatrix} H \\ 0, \ldots, 0, h \end{pmatrix}. \tag{8.5}$$

This formula can be used to compute $\underline{H}$ in line 6 of Algorithm 8.2.

8.2.1.2. CA-Arnoldi(s, t). We now derive CA-Arnoldi(s, t), where the basis length s in CA-Arnoldi(s, t) can be shorter than the restart length $s \cdot t$. Here, t refers to the number of 'outer iterations', or times that the matrix powers kernel is invoked and its output vectors are made orthogonal to previous vectors and among themselves. Note that CA-Arnoldi(s) is the same as CA-Arnoldi($s, t = 1$).

The ability to choose s shorter than the restart length $s \cdot t$ has at least two advantages.

(1) s-step methods are not numerically stable if s is too large, but if the restart length r is too short, the Krylov method may converge slowly or not at all ('stagnation'). In that case, CA-Arnoldi(s, t) can use a known stable value of s, while still choosing a restart length r for which Arnoldi is known to converge sufficiently fast.

(2) The best value of s for performance may be much shorter than typical restart lengths.

Previous s-step Arnoldi or GMRES algorithms, including Walker's Householder GMRES (Walker 1988) and the Newton-basis GMRES algorithm of Bai *et al.* (1991, 1994), had the same limitation as CA-Arnoldi(s). Like CA-Arnoldi(s), CA-Arnoldi(s, t) communicates a factor of s less than conventional Arnoldi with the same restart length $r = s \cdot t$. CA-Arnoldi(s, t) requires the same storage as conventional Arnoldi. It may require a lower-order term more of floating-point arithmetic operations than conventional Arnoldi, but this is due only to the matrix powers kernel (see Section 7 for details).

We first briefly discuss block Gram–Schmidt methods, which will be used in CA-Arnoldi(s, t); for further details and related work, see Hoemmen (2010).

Orthogonalizing a vector against an orthogonal basis in order to add the vector to the basis is an important kernel in most KSMs. Some KSMs, such as Arnoldi and GMRES, explicitly orthogonalize each new basis vector against all previous basis vectors.

We showed in Demmel *et al.* (2012) that Classical Gram–Schmidt (CGS) and Modified Gram–Schmidt (MGS) communicate asymptotically more than the lower bound, both in parallel and between levels of the memory hierarchy. This is a problem for KSMs such as Arnoldi and GMRES, which may spend a large fraction of their time orthogonalizing basis vectors. Whatever progress we make replacing SpMV with CA-Akx (Section 7), we also have to address the orthogonalization phase(s) in order to make our methods truly communication-avoiding.

Block CGS is the natural generalization of CGS where a vector becomes a block of k vectors, a dot product becomes a matrix multiplication of two $n \times k$ blocks, and dividing a vector by its 2-norm becomes a (TS)QR decomposition of an $n \times k$ block. Blocked versions of Gram–Schmidt are not new: see, for example, Jalby and Philippe (1991), Vanderstraeten (1999) and Stewart (2008).

These block variants reduce the amount of communication required. If the number of columns in each block is k, block CGS and block MGS algorithms

require a factor of $\Theta(k)$ and $\Theta(k^2)$, respectively, fewer messages in parallel than their analogous non-blocked versions, and the sequential versions also both require a factor of $\Theta(k)$ fewer words transferred between slow and fast memory. Mohiyuddin *et al.* (2009) and Hoemmen (2010) suggested combining block Gram–Schmidt with TSQR (Section 3.3.5), to improve the communication costs to attain the lower bounds of Demmel *et al.* (2012).

We will thus use block Gram–Schmidt methods to orthogonalize the 'new' set of basis vectors against the previously orthogonalized basis vectors. This may be done for all sk basis vectors at once (the block CGS approach) or $k-1$ times for each group of s previous basis vectors (the block MGS approach).

In exact arithmetic, both block CGS and block MGS compute the same updated basis vectors; the differences are in performance and accuracy. The accuracy of block MGS is comparable to that of orthogonalizing a vector against k unit-length orthogonal vectors using standard MGS. The accuracy of block CGS is comparable to that of orthogonalizing a vector against k unit-length orthogonal vectors with standard CGS. Loss of orthogonality is an issue for all aforementioned Gram–Schmidt methods; we refer to Stewart (2008) for a concise summary and bibliography of numerical stability concerns when performing Gram–Schmidt orthogonalization in finite precision arithmetic. We discuss reorthogonalization for CA-KSMs in Section 8.5.2.4.

For simplicity, we will use block CGS in CA-Arnoldi(s,t) and refer to Hoemmen (2010) for further details on orthogonalization methods, including block MGS.

We now introduce the three levels of notation used in CA-Arnoldi(s,t). For more outer levels, we use taller and more elaborate typefaces. We use lower-case Roman type to denote single basis vectors, upper-case calligraphic type for $n \times s$ matrices whose columns are the Krylov subspace basis vectors generated in a single outer iteration, and upper-case blackletter type for a collection of $s(k+1)$ groups of orthogonal basis vectors from all previous outer iterations up to k. For example, v_{sk+j} is a single basis vector in outer iteration k, $\mathcal{V}_k = [v_{sk+1}, v_{sk+2}, \ldots, v_{sk+s}]$ is an $n \times s$ matrix whose columns are s basis vectors in a group for outer iteration k, and $\mathfrak{Q}_k$ is a collection of $k+1$ groups of s orthogonal basis vectors at outer iteration k, that is, $\mathfrak{Q}_k = [Q_0, Q_1, \ldots, Q_k]$.

Using this notation, if CA-Arnoldi(s,t) performs all its iterations without breaking down, it produces the matrix relation

$$A\mathfrak{Q}_k = \underline{\mathfrak{Q}}_k \underline{\mathfrak{H}}_k, \tag{8.6}$$

where

$$\mathfrak{Q}_k = [Q_0, Q_1, \ldots, Q_k],$$
$$\underline{\mathfrak{Q}}_k = [\mathfrak{Q}_k, q_{s(k+1)+1}],$$

$$\mathfrak{H}_k = \begin{cases} H_0 & \text{for } k = 0, \\ \begin{pmatrix} \mathfrak{H}_{k-1} & \mathfrak{H}_{k-1,k} \\ h_{k-1} e_1 e_{s(k-1)}^T & H_k \end{pmatrix} & \text{for } k > 0, \end{cases}$$

and

$$\underline{\mathfrak{H}}_k = \begin{pmatrix} \mathfrak{H}_k \\ h_k e_{s(k+1)}^T \end{pmatrix}.$$

Suppose that we perform s steps of Arnoldi, either conventional Arnoldi or the s-step variant, using the $n \times n$ matrix A and the starting vector q_1. Assuming that the iteration does not break down, this results in an $n \times (s+1)$ matrix of basis vectors $\underline{Q}_0$, with

$$\underline{Q}_0 = [Q_0, q_{s+1}] = [q_1, q_2, \ldots, q_s, q_{s+1}].$$

Its columns are an orthogonal basis for the Krylov subspace span$\{q_1, Aq_1, \ldots, A^s q_1\}$. Arnoldi also produces an $(s+1) \times s$ nonsingular upper Hessenberg matrix $\underline{H}_0$, with

$$\underline{H}_0 = \begin{pmatrix} H_0 \\ h_0 e_s^T \end{pmatrix}.$$

The upper Hessenberg matrix $\underline{H}_0$ satisfies

$$AQ_0 = \underline{Q}_0 \underline{H}_0 = Q_0 H_0 + h_0 \cdot q_{s+1} e_s^T. \tag{8.7}$$

We have already shown how to compute $\underline{Q}_0$ and $\underline{H}_0$ using CA-Arnoldi(s) instead of conventional Arnoldi. However we choose to compute $\underline{Q}_0$ and $\underline{H}_0$, (8.7) still holds in exact arithmetic.[10]

If this were CA-Arnoldi(s), we would have to restart now. Instead, we take the last basis vector q_{s+1}, and make it the input of the matrix powers kernel. We set $v_{s+1} = q_{s+1}$, and the matrix powers kernel produces an $n \times (s+1)$ matrix of basis vectors $\underline{\mathcal{V}}_1$ with

$$\underline{\mathcal{V}}_1 = [q_{s+1} = v_{s+1}, v_{s+2}, \ldots, v_{2s+1}].$$

We write

$$\begin{aligned} \mathcal{V}_1 &= [v_{s+1}, v_{s+2}, \ldots, v_{2s}], \\ \underline{\mathcal{V}}_1 &= [\mathcal{V}_1, v_{2s+1}], \\ \acute{\mathcal{V}}_1 &= [v_{s+2}, \ldots, v_{2s}], \\ \underline{\acute{\mathcal{V}}}_1 &= [\acute{\mathcal{V}}_1, v_{2s+1}]. \end{aligned}$$

The underline, as before, indicates 'one more at the end', and the acute

[10] In practice, we may want to begin a run of CA-Arnoldi(s, t) with s iterations of conventional Arnoldi first, in order to compute Ritz values, which are used in successive outer iterations to compute a good basis: see Section 8.5.1. This is why we allow the first outer iteration to be computed either by conventional Arnoldi or by CA-Arnoldi(s).

accent (pointing up and to the right) indicates 'one fewer at the beginning'. The vector omitted 'at the beginning' is $q_{s+1} = v_{s+1}$, which is already orthogonal to the previous basis vectors; we no longer need or want to change it.

The columns of $\acute{\underline{\mathcal{V}}}_1$ have not yet been orthogonalized against the columns of $\underline{Q}_0$. We set

$$\mathfrak{Q}_0 = Q_0 \quad \text{and} \quad \underline{\mathfrak{Q}}_0 = \underline{Q}_0. \tag{8.8}$$

Orthogonalizing the columns of $\acute{\underline{\mathcal{V}}}_1$ against the previous basis vectors (here the columns of $\underline{\mathfrak{Q}}_0$) is equivalent to updating a QR factorization by adding the s columns of $\acute{\underline{\mathcal{V}}}_1$ to the right of $\underline{\mathfrak{Q}}_0$. This results in an efficiently computed QR factorization of $[\underline{\mathfrak{Q}}_0, \acute{\underline{\mathcal{V}}}_1]$:

$$[\underline{\mathfrak{Q}}_0, \acute{\underline{\mathcal{V}}}_1] = [\underline{\mathfrak{Q}}_0, \acute{\underline{Q}}_1] \cdot \begin{pmatrix} I_{s+1,s+1} & \acute{\underline{\mathfrak{R}}}_{0,1} \\ 0 & \acute{\underline{R}}_1 \end{pmatrix}. \tag{8.9}$$

We perform this update in two steps. First, we use a block Gram–Schmidt operation (see Section 8.2.1.2) to orthogonalize the columns of $\acute{\underline{\mathcal{V}}}_1$ against the columns of $\underline{\mathfrak{Q}}_0$:

- $\acute{\underline{\mathfrak{R}}}_{0,1} = \underline{\mathfrak{Q}}_0^H \acute{\underline{\mathcal{V}}}_1$,
- $\acute{\underline{\mathcal{V}}}_1' = \acute{\underline{\mathcal{V}}}_1 - \underline{\mathfrak{Q}}_0 \cdot \acute{\underline{\mathfrak{R}}}_{0,1}$.

Then, we compute the QR factorization of $\acute{\underline{\mathcal{V}}}_1'$, which makes the new basis vectors mutually orthogonal as well as orthogonal to the previous basis vectors:

$$\acute{\underline{\mathcal{V}}}_1' = \acute{\underline{Q}}_1 \acute{\underline{R}}_1.$$

Now that we have the desired QR factorization update and a new set of s orthogonalized Arnoldi basis vectors, we can update the upper Hessenberg matrix as well. Just as in CA-Arnoldi(s), the basis vectors $\underline{\mathcal{V}}_1$ satisfy $A\mathcal{V}_1 = \underline{\mathcal{V}}_1 \underline{B}_1$ for the nonsingular upper Hessenberg matrix $\underline{B}_1$, which is defined by the choice of basis. That means

$$A[\mathfrak{Q}_0, \mathcal{V}_1] = [\mathfrak{Q}_0, \underline{\mathcal{V}}_1] \underline{\mathfrak{B}}_1, \tag{8.10}$$

where the $(2s+1) \times 2s$ matrix $\underline{\mathfrak{B}}_1$ satisfies

$$\underline{\mathfrak{B}}_1 = \begin{pmatrix} H_0 & 0_{s,s} \\ h_0 e_1 e_s^T & \underline{B}_1 \end{pmatrix} = \begin{pmatrix} \mathfrak{B}_1 \\ b_1 e_{2s}^T \end{pmatrix}. \tag{8.11}$$

Combining (8.10) with the QR factorization update in (8.9) and the definition of $\underline{\mathfrak{B}}_1$ in (8.11), we get

$$A[\mathfrak{Q}_0, Q_1] \begin{pmatrix} I & \mathfrak{R}_{0,1} \\ 0 & R_1 \end{pmatrix} = [\mathfrak{Q}_0, \underline{\mathcal{V}}_1] \begin{pmatrix} I & \underline{\mathfrak{R}}_{0,1} \\ 0 & \underline{R}_1 \end{pmatrix} \begin{pmatrix} H_0 & 0_{s,s} \\ h_0 e_1 e_s^T & \underline{B}_1 \end{pmatrix}, \tag{8.12}$$

where

$$\mathfrak{R}_{0,1} = \mathfrak{Q}_0^H \mathcal{V}_1 = [e_1, \mathfrak{Q}_0^H \acute{\mathcal{V}}_1],$$
$$\underline{\mathfrak{R}}_{0,1} = \mathfrak{Q}_0^H \underline{\mathcal{V}}_1 = [e_1, \mathfrak{Q}_0^H \underline{\acute{\mathcal{V}}}_1],$$

and R_1 and $\underline{R}_1$ are the R factors in the QR factorization of $\mathcal{V}_1$ and $\underline{\mathcal{V}}_1$, respectively. (Similarly, $R_1(1,1) = \underline{R}_1(1,1) = 1$.) This is just a rearrangement of the updated R factor in equation (8.9) and involves no additional computation.

We set $\mathfrak{H}_0 = H_0$ and $\underline{\mathfrak{H}}_0 = \underline{H}_0$. By (8.12), we can recover the $(2s+1) \times 2s$ upper Hessenberg matrix $\underline{\mathfrak{H}}_1$ which is the same as would be computed by $2s$ steps of conventional Arnoldi iteration. We have

$$A[\mathfrak{Q}_0, Q_1] = [\mathfrak{Q}_0, \underline{Q}_1]\underline{\mathfrak{H}}_1,$$

and therefore

$$\underline{\mathfrak{H}}_1 = \begin{pmatrix} I & \underline{\mathfrak{R}}_{0,1} \\ 0 & \underline{R}_1 \end{pmatrix} \begin{pmatrix} \mathfrak{H}_0 & 0_{s,s} \\ h_0 e_1 e_s^T & \underline{B}_1 \end{pmatrix} \begin{pmatrix} I & \mathfrak{R}_{0,1} \\ 0 & R_1 \end{pmatrix}^{-1}. \tag{8.13}$$

The formula for $\underline{\mathfrak{H}}_1$ in (8.13) can be simplified so that $\underline{\mathfrak{H}}_1$ can be computed as an update without needing to change $\underline{\mathfrak{H}}_0$.

Applying the update technique in this section $k < t-1$ times results in a total of $k+1$ 'outer iterations' of CA-Arnoldi(s,t). In outer iteration k, the matrix powers kernel outputs new basis vectors $\underline{\acute{\mathcal{V}}}_k$. Block CGS is then used to orthogonalize vectors in $\underline{\acute{\mathcal{V}}}_k$ against previously computed vectors $\underline{\mathfrak{Q}}_{k-1}$, producing the $(sk+1) \times s$ factor $\underline{\acute{\mathfrak{R}}}_{k-1,k}$. After computing the QR factorization $\underline{\acute{V}}_k' = \underline{\acute{Q}}_k \underline{\acute{R}}_k$, the global QR factors can be denoted by

$$\underline{\mathfrak{Q}}_k = [\underline{\mathfrak{Q}}_{k-1}, \underline{\acute{Q}}_k], \tag{8.14}$$

and

$$\underline{\mathfrak{R}}_k = \begin{pmatrix} I_{s(k-1)+1, s(k-1)+1} & \underline{\acute{\mathfrak{R}}}_{k-1,k} \\ 0_{s, s(k-1)+1} & \underline{\acute{R}}_k \end{pmatrix}. \tag{8.15}$$

The Arnoldi relation that results is

$$\begin{aligned} A\mathfrak{Q}_k &= \mathfrak{Q}_k \mathfrak{H}_k + h_k q_{s(k+1)+1} e_{s(k+1)}^T \\ &= \underline{\mathfrak{Q}}_k \underline{\mathfrak{H}}_k, \end{aligned} \tag{8.16}$$

where the outputs of CA-Arnoldi(s,t) are the $n \times (s(k+1)+1)$ matrix of basis vectors $\underline{\mathfrak{Q}}_k$, and the $(s(k+1)+1) \times s(k+1)$ upper Hessenberg matrix $\underline{\mathfrak{H}}_k$. Here,

$$\underline{\mathfrak{H}}_k = \underline{\mathfrak{R}}_k \underline{\mathfrak{B}}_k \mathfrak{R}_k^{-1}, \tag{8.17}$$

Algorithm 8.3 CA-Arnoldi(s, t)

Require: $n \times n$ matrix A and $n \times 1$ starting vector v
Output: Matrices $\underline{\mathfrak{Q}}_k$ and $\underline{\mathfrak{H}}_k$ satisfying (8.6)
1: $h_0 = \|r\|_2$, $q_1 = v/h_0$
2: **for** $k = 0$ to $t - 1$ **do**
3: Fix basis conversion matrix $\underline{B}_k$
4: Compute $\acute{\underline{\mathcal{V}}}_k$ using CA-Akx
5: **if** $k = 0$ **then**
6: Compute QR factorization $\underline{\mathcal{V}}_0 = \underline{Q}_0 \underline{R}_0$ via TSQR
7: $\underline{\mathfrak{Q}}_0 = \underline{Q}_0$
8: $\underline{\mathfrak{H}}_0 = \underline{R}_0 \underline{B}_0 R_0^{-1}$
9: **else**
10: $\acute{\underline{\mathfrak{R}}}_{k-1,k} = \underline{\mathfrak{Q}}_{k-1}^H \acute{\underline{\mathcal{V}}}_k$ (block CGS)
11: $\acute{\underline{\mathcal{V}}}'_k = \acute{\underline{\mathcal{V}}}_k - \underline{\mathfrak{Q}}_{k-1} \acute{\underline{\mathfrak{R}}}_{k-1,k}$ (block CGS, continued)
12: Compute QR factorization $\acute{\underline{\mathcal{V}}}'_k = \acute{\underline{Q}}_k \acute{\underline{R}}_k$
13: Update $\underline{\mathfrak{Q}}_k$ and $\underline{\mathfrak{R}}_k$ by (8.14) and (8.15), resp.
14: Update $\underline{\mathfrak{H}}_k$ by (8.17)
15: **end if**
16: **end for**

where

$$\mathfrak{R}_k = \begin{pmatrix} I_{s(k-1)+1,s(k-1)+1} & \acute{\mathfrak{R}}_{k-1,k} \\ 0_{s,s(k-1)+1} & \acute{R}_k \end{pmatrix} \quad \text{and} \tag{8.18}$$

$$\underline{\mathfrak{B}}_k = \begin{pmatrix} \mathfrak{H}_{k-1} & 0_{s(k-1),s} \\ h_{k-1} e_1 e_{s(k-1)}^T & \underline{B}_k \end{pmatrix}. \tag{8.19}$$

The CA-Arnoldi(s, t) algorithm, shown in Algorithm 8.3, follows directly from the described QR factorization update. Again, we assume that no breakdown occurs. For simplicity, this variant uses a block CGS update procedure in lines 10 and 11. The update produces the same results (in exact arithmetic) as would a full QR factorization of all the basis vectors, but saves work. The update also preserves the scalings of previous basis vectors, which a full QR factorization might not do. There are many alternative implementation choices for computing the QR factorization update and the upper Hessenberg update: for details, see Hoemmen (2010, §3.3.3–4).

8.2.2. Nonsymmetric Lanczos (BIOC)

Given an $n \times n$ matrix A and starting vector y_0, m steps of the Lanczos process (assuming no breakdown) produce the decompositions

$$AY_m = Y_{m+1} \underline{T}_m \quad \text{and} \quad A^H \tilde{Y}_m = \tilde{Y}_{m+1} \underline{\tilde{T}}_m, \tag{8.20}$$

where $Y_m = [y_0, \ldots, y_{m-1}]$ and $\underline{T}_m$ is an $(m+1) \times m$ tridiagonal matrix (similarly for $\tilde{Y}_m$ and $\underline{\tilde{T}}_m$). The matrices Y_{m+1} and $\tilde{Y}_{m+1}$ are bases for the Krylov subspaces $\mathcal{K}_{m+1}(A, y_0)$ and $\mathcal{K}_{m+1}(A^H, \tilde{y}_0)$, respectively, and are biorthogonal, that is, $\tilde{Y}_{m+1}^H Y_{m+1} = I$. The eigenvalues of T_m (the upper $m \times m$ submatrix of $\underline{T}_m$) are called Petrov values (or Ritz values for SPD A), and are useful approximations for the eigenvalues of A.

In this section we present a communication-avoiding version of the 'BIOC' variant of nonsymmetric Lanczos (see, *e.g.*, Gutknecht 1997), which we call CA-BIOC. We note that in the case that A is SPD, elimination of certain computations from BIOC and CA-BIOC gives algorithms for symmetric Lanczos and communication-avoiding symmetric Lanczos, respectively. The BIOC algorithm, shown in Algorithm 8.4, is governed by two coupled two-term recurrences (rather than the single three-term recurrence (8.20)) which can be written as

$$\begin{aligned} Y_m &= V_m U_m, & \tilde{Y}_m &= \tilde{V}_m \tilde{U}_m, \\ AV_m &= Y_{m+1} \underline{L}_m, & A^H \tilde{V}_m &= \tilde{Y}_{m+1} \underline{\tilde{L}}_m, \end{aligned} \tag{8.21}$$

where V_m and $\tilde{V}_m$ are biconjugate, and as before, Y_m and $\tilde{Y}_m$ are biorthogonal. The matrices

$$\underline{L}_m = \begin{pmatrix} \phi_0 & & & \\ \gamma_0 & \phi_1 & & \\ & \ddots & \ddots & \\ & & \gamma_{m-2} & \phi_{m-1} \\ & & & \gamma_{m-1} \end{pmatrix} \quad \text{and} \quad U_m = \begin{pmatrix} \psi_0 & & & \\ 1 & \psi_1 & & \\ & \ddots & \ddots & \\ & & 1 & \psi_{m-1} \end{pmatrix} \tag{8.22}$$

are the LU factors of $\underline{T}_m$ in (8.20). Although BIOC breakdown can occur if the (pivot-free) LU factorization of $\underline{T}_m$ does not exist, the four bidiagonal matrices in (8.21) are preferable to the two tridiagonal matrices in (8.20) for a number of reasons (Parlett 1995). Derivations for a communication-avoiding three-term recurrence variant of symmetric Lanczos can be found in Hoemmen (2010).

Now, suppose we want to perform blocks of $s \geq 1$ iterations at once. That is, we wish to calculate

$$\begin{aligned} &[v_{sk+1}, \ldots, v_{sk+s}], \quad [\tilde{v}_{sk+1}, \ldots, \tilde{v}_{sk+s}], \\ &[y_{sk+1}, \ldots, y_{sk+s}], \quad [\tilde{y}_{sk+1}, \ldots, \tilde{y}_{sk+s}], \end{aligned}$$

given $\{v_{sk}, \tilde{v}_{sk}, y_{sk}, \tilde{y}_{sk}\}$, for $k \geq 0$.

By induction on lines $\{6, 7, 10, 11\}$ of conventional BIOC (Algorithm 8.4), it can be shown that, for iterations $sk + j$, $0 < j \leq s$, the vector iterates

Algorithm 8.4 BIOC

Require: $n \times n$ matrix A and length n starting vectors $y_0, \tilde{y}_0 \in \mathbb{C}^n$, such that $\|y_0\|_2 = \|\tilde{y}_0\|_2 = 1$, and $\delta_0 = \tilde{y}_0^H y_0 \neq 0$

Output: Matrices V_m, $\tilde{V}_m$, Y_m, $\tilde{Y}_m$, $\underline{L}_m$, $\underline{\tilde{L}}_m$, U_m, and $\tilde{U}_m$ satisfying (8.21)

1: Set $v_0 = \tilde{v}_0 = y_0$.
2: **for** $m = 0, 1, \ldots$, until convergence **do**
3: $\quad \hat{\delta}_m = \tilde{v}_m^H A v_m$
4: $\quad \phi_m = \hat{\delta}_m / \delta_m$
5: $\quad$ Choose $\gamma_m, \tilde{\gamma}_m \neq 0$
6: $\quad y_{m+1} = (A v_m - \phi_m y_m)/\gamma_m$
7: $\quad \tilde{y}_{m+1} = (A^H \tilde{v}_m - \overline{\phi}_m \tilde{y}_m)/\tilde{\gamma}_m$
8: $\quad \delta_{m+1} = \tilde{y}_{m+1}^H y_{m+1}$
9: $\quad \psi_m = \overline{\tilde{\gamma}}_m \delta_{m+1}/\hat{\delta}_m$, $\tilde{\psi}_m = \overline{\gamma_m \delta_{m+1}/\hat{\delta}_m}$
10: $\quad v_{m+1} = y_{m+1} - \psi_m v_m$
11: $\quad \tilde{v}_{m+1} = \tilde{y}_{m+1} - \tilde{\psi}_m \tilde{v}_m$
12: **end for**

satisfy

$$\begin{aligned} v_{sk+j}, y_{sk+j} &\in \mathcal{K}_{s+1}(A, v_{sk}) + \mathcal{K}_s(A, y_{sk}), \\ \tilde{v}_{sk+j}, \tilde{y}_{sk+j} &\in \mathcal{K}_{s+1}(A^H, \tilde{v}_{sk}) + \mathcal{K}_s(A^H, \tilde{y}_{sk}), \end{aligned} \tag{8.23}$$

where we exploit the nesting of the Krylov bases, that is, $\mathcal{K}_j(A, v_{sk}) \subseteq \mathcal{K}_{j+1}(A, v_{sk})$ (and similarly for other starting vectors). Then, to perform iterations $sk + j$ for $0 < j \leq s$, we will use the Krylov basis matrices

$$\begin{aligned} \underline{\mathcal{V}}_k &= [\rho_0(A)v_{sk}, \rho_1(A)v_{sk}, \ldots, \rho_s(A)v_{sk}], & \operatorname{span}(\underline{\mathcal{V}}_k) &= \mathcal{K}_{s+1}(A, v_{sk}), \\ \underline{\tilde{\mathcal{V}}}_k &= [\rho_0(A^H)\tilde{v}_{sk}, \rho_1(A^H)\tilde{v}_{sk}, \ldots, \rho_s(A^H)\tilde{v}_{sk}], & \operatorname{span}(\underline{\tilde{\mathcal{V}}}_k) &= \mathcal{K}_{s+1}(A^H, \tilde{v}_{sk}), \\ \underline{\mathcal{Y}}_k &= [\rho_0(A)y_{sk}, \rho_1(A)y_{sk}, \ldots, \rho_{s-1}(A)y_{sk}], & \operatorname{span}(\underline{\mathcal{Y}}_k) &= \mathcal{K}_s(A, y_{sk}), \\ \underline{\tilde{\mathcal{Y}}}_k &= [\rho_0(A^H)\tilde{y}_{sk}, \rho_1(A^H)\tilde{y}_{sk}, \ldots, \rho_{s-1}(A^H)\tilde{y}_{sk}], & \operatorname{span}(\underline{\tilde{\mathcal{Y}}}_k) &= \mathcal{K}_s(A^H, \tilde{y}_{sk}), \end{aligned} \tag{8.24}$$

where $\rho_j(z)$ is a polynomial of degree j, satisfying a three-term recurrence

$$\begin{aligned} &\rho_0(z) = 1, \quad \rho_1(z) = (z - \hat{\alpha}_0)\rho_0(z)/\hat{\gamma}_0, \quad \text{and} \\ &\rho_j(z) = ((z - \hat{\alpha}_{j-1})\rho_{j-1}(z) - \hat{\beta}_{j-2}\rho_{j-2}(z))/\hat{\gamma}_{j-1}, \quad \text{for } j > 1. \end{aligned} \tag{8.25}$$

This derivation generalizes to polynomials satisfying longer recurrences, although three-term recurrences are most commonly used.

Using the defined Krylov matrices (8.24) and the relations (8.23), we can represent components of the BIOC iterates in $\mathbb{C}^n$ by their coordinates in the Krylov bases, that is, subspaces of $\mathbb{C}^n$ of dimension at most $2s + 1$. We introduce coefficient vectors $\{v'_j, \tilde{v}'_j, y'_j, \tilde{y}'_j\}$, each of length $2s + 1$, to

represent the length n vectors $\{v_{sk+j}, \tilde{v}_{sk+j}, y_{sk+j}, \tilde{y}_{sk+j}\}$. That is,

$$\begin{aligned} v_{sk+j} &= [\underline{\mathcal{V}}_k, \underline{\mathcal{Y}}_k]v'_j, \qquad y_{sk+j} = [\underline{\mathcal{V}}_k, \underline{\mathcal{Y}}_k]y'_j, \\ \tilde{v}_{sk+j} &= [\underline{\tilde{\mathcal{V}}}_k, \underline{\tilde{\mathcal{Y}}}_k]\tilde{v}'_j, \qquad \tilde{y}_{sk+j} = [\underline{\tilde{\mathcal{V}}}_k, \underline{\tilde{\mathcal{Y}}}_k]\tilde{y}'_j, \end{aligned} \tag{8.26}$$

where the base cases for these recurrences are given by

$$v'_0 = \tilde{v}'_0 = [1, 0_{1,2s}]^T \quad \text{and} \quad y'_0 = \tilde{y}'_0 = [0_{1,s+1}, 1, 0_{1,s-1}]^T. \tag{8.27}$$

Then, the iterate updates (lines {6, 7, 10, 11} of Algorithm 8.4) in the Krylov basis become

$$[\underline{\mathcal{V}}_k, \underline{\mathcal{Y}}_k]y'_{j+1} = (A[\underline{\mathcal{V}}_k, \underline{\mathcal{Y}}_k]v'_j - \phi_{sk+j}[\underline{\mathcal{V}}_k, \underline{\mathcal{Y}}_k]y'_j)/\gamma_{sk+j}, \tag{8.28}$$

$$[\underline{\tilde{\mathcal{V}}}_k, \underline{\tilde{\mathcal{Y}}}_k]\tilde{y}'_{j+1} = (A^H[\underline{\tilde{\mathcal{V}}}_k, \underline{\tilde{\mathcal{Y}}}_k]\tilde{v}'_j - \tilde{\phi}_{sk+j}[\underline{\tilde{\mathcal{V}}}_k, \underline{\tilde{\mathcal{Y}}}_k]\tilde{y}'_j)/\tilde{\gamma}_{sk+j}, \tag{8.29}$$

$$[\underline{\mathcal{V}}_k, \underline{\mathcal{Y}}_k]v'_{j+1} = [\underline{\mathcal{V}}_k, \underline{\mathcal{Y}}_k]y'_{j+1} + \psi_{sk+j}[\underline{\mathcal{V}}_k, \underline{\mathcal{Y}}_k]v'_j, \quad \text{and} \tag{8.30}$$

$$[\underline{\tilde{\mathcal{V}}}_k, \underline{\tilde{\mathcal{Y}}}_k]\tilde{v}'_{j+1} = [\underline{\tilde{\mathcal{V}}}_k, \underline{\tilde{\mathcal{Y}}}_k]\tilde{y}'_{j+1} + \tilde{\psi}_{sk+j}[\underline{\tilde{\mathcal{V}}}_k, \underline{\tilde{\mathcal{Y}}}_k]\tilde{v}'_j. \tag{8.31}$$

for iterations $sk + j$, where $0 \le j \le s - 1$.

Next, we represent the multiplications by A and A^H (lines 6 and 7 of Algorithm 8.4) in the new coordinates, in order to manipulate (8.28) and (8.29). We first note that the recurrence (8.25) for generating the matrices $\underline{\mathcal{V}}_k$ and $\underline{\mathcal{Y}}_k$ can be written in matrix form as

$$\begin{aligned} A\mathcal{V}_k &= \underline{\mathcal{V}}_k\underline{B}_k, \quad A\mathcal{Y}_k = \underline{\mathcal{Y}}_k\underline{B}_k, \\ A^H\tilde{\mathcal{V}}_k &= \underline{\tilde{\mathcal{V}}}_k\underline{B}_k, \quad A^H\tilde{\mathcal{Y}}_k = \underline{\tilde{\mathcal{Y}}}_k\underline{B}_k, \end{aligned} \tag{8.32}$$

with

$$\underline{B}_k = \begin{pmatrix} \hat{\alpha}_0 & \hat{\beta}_0 & & \\ \hat{\gamma}_0 & \hat{\alpha}_1 & \ddots & \\ & \hat{\gamma}_1 & \ddots & \hat{\beta}_{s-2} \\ & & \ddots & \hat{\alpha}_{s-1} \\ & & & \hat{\gamma}_{s-1} \end{pmatrix} \in \mathbb{C}^{(s+1)\times s}. \tag{8.33}$$

Note that to perform iterations $0 \le j \le s - 1$, we need to perform multiplication of A with each v_{sk+j} (likewise for A^H and $\tilde{v}_{sk+j}$). By (8.23) and (8.26), for $0 \le j \le s - 1$,

$$\begin{aligned} Av_{sk+j} &= A[\underline{\mathcal{V}}_k, \underline{\mathcal{Y}}_s]v'_j = A[\mathcal{V}_k, 0_{n,1}, \mathcal{Y}_k, 0_{n,1}]v'_j = [\underline{\mathcal{V}}_k, \underline{\mathcal{Y}}_k]\underline{B}'_k v'_j, \\ A^H\tilde{v}_{sk+j} &= A^H[\underline{\tilde{\mathcal{V}}}_k, \underline{\tilde{\mathcal{Y}}}_k]\tilde{v}'_j = A^H[\tilde{\mathcal{V}}_k, 0_{n,1}, \tilde{\mathcal{Y}}_k, 0_{n,1}]\tilde{v}'_j = [\underline{\tilde{\mathcal{V}}}_k, \underline{\tilde{\mathcal{Y}}}_k]\underline{B}'_k\tilde{v}'_j, \end{aligned} \tag{8.34}$$

where

$$\underline{B}'_k = \begin{bmatrix} [\underline{B}_k & 0_{s+1,1}] & \\ & [B_k & 0_{s,1}] \end{bmatrix}, \tag{8.35}$$

and B_k is the matrix consisting of the first s rows and first $s - 1$ columns

of $\underline{B}_k$. We now substitute (8.26) and (8.34) into conventional BIOC. The vector updates in each of lines 6, 7, 10, and 11 of Algorithm 8.4 are now expressed as a linear combination of the columns of the Krylov basis matrices. We can match coefficients on the right- and left-hand sides to obtain the governing recurrences

$$y'_{j+1} = (\underline{B}'_k v'_j - \phi_{sk+j} y'_j)/\gamma_{sk+j}, \tag{8.36}$$

$$\tilde{y}'_{j+1} = (\underline{B}'_k \tilde{v}'_j - \overline{\phi}_{sk+j} \tilde{y}'_j)/\tilde{\gamma}_{sk+j}, \tag{8.37}$$

$$v'_{j+1} = y'_{j+1} + \psi_{sk+j} v'_j, \tag{8.38}$$

$$\tilde{v}'_{j+1} = \tilde{y}'_{j+1} + \tilde{\psi}_{sk+j} \tilde{v}'_j, \tag{8.39}$$

for $0 \le j \le s-1$.

We also need scalar quantities $\hat{\delta}_{sk+j}$ and δ_{sk+j} which are computed from inner products involving the vector iterates. We represent these inner products (lines 3 and 8 of Algorithm 8.4) in the new basis, using the Gram(-like) matrices

$$G_k^* = [\tilde{\underline{\mathcal{Y}}}_k, \tilde{\underline{\mathcal{Y}}}_k]^H [\underline{\mathcal{Y}}_k, \underline{\mathcal{Y}}_k] \tag{8.40}$$

of size $(2s+1) \times (2s+1)$. We can then write the required inner products as

$$\tilde{y}_{sk+j+1}^H y_{sk+j+1} = \tilde{y}_{j+1}'^H G_k^* y'_{j+1}, \tag{8.41}$$

$$\tilde{v}_{sk+j}^H A v_{sk+j} = \tilde{v}_j'^H G_k^* \underline{B}'_k v'_j. \tag{8.42}$$

In our BIOC formulation, we have allowed freedom in choosing the values γ_m. It is common to choose starting vectors $\|y_0\|_2 = \|\tilde{y}_0\|_2$ and choose γ_m and $\tilde{\gamma}_m$ in line 5 of Algorithm 8.4 such that $\|y_{m+1}\|_2 = \|\tilde{y}_{m+1}\|_2 = 1$, that is,

$$\gamma_m = \|A v_m - \phi_m y_m\|_2, \quad \tilde{\gamma}_m = \|A^H \tilde{v}_m - \overline{\phi}_m \tilde{y}_m\|_2.$$

In this case, CA-BIOC also requires us to compute the matrices

$$G_k = [\underline{\mathcal{Y}}_k, \underline{\mathcal{Y}}_k]^H [\underline{\mathcal{Y}}_k, \underline{\mathcal{Y}}_k], \quad \tilde{G}_k = [\tilde{\underline{\mathcal{Y}}}_k, \tilde{\underline{\mathcal{Y}}}_k]^H [\tilde{\underline{\mathcal{Y}}}_k, \tilde{\underline{\mathcal{Y}}}_k],$$

in each outer loop. Note that computing these matrices does not asymptotically affect communication costs if they can be computed simultaneously with G_k^*. Then we can compute γ_m and $\tilde{\gamma}_m$ by

$$\|A v_{sk+j} - \phi_{sk+j} y_{sk+j}\|_2 = \sqrt{(\underline{B}'_k v'_j - \phi_{sk+j} y'_j)^H G_k (\underline{B}'_k v'_j - \phi_{sk+j} y'_j)}, \tag{8.43}$$

$$\|A^H \tilde{v}_{sk+j} - \overline{\phi}_{sk+j} \tilde{y}_{sk+j}\|_2 = \sqrt{(\underline{B}'_k \tilde{v}'_j - \tilde{\phi}_{sk+j} \tilde{y}'_j)^H \tilde{G}_k (\underline{B}'_k \tilde{v}'_j - \overline{\phi}_{sk+j} \tilde{y}'_j)}, \tag{8.44}$$

in each inner loop iteration with no communication.

Algorithm 8.5 Communication-avoiding BIOC

Require: $n \times n$ matrix A and length n starting vectors $y_0, \tilde{y}_0 \in \mathbb{C}^n$, such that $\|y_0\|_2 = \|\tilde{y}_0\|_2 = 1$, and $\delta_0 = \tilde{y}_0^H y_0 \neq 0$

Output: Matrices V_m, $\tilde{V}_m$, Y_m, $\tilde{Y}_m$, $\underline{L}_m$, $\underline{\tilde{L}}_m$, U_m, and $\tilde{U}_m$ satisfying (8.21)

Set $v_0 = \tilde{v}_0 = y_0$
for $k = 0, 1, \ldots,$ until convergence **do**
  Assemble $\underline{B}'_k$ according to (8.35) and (8.33)
  Compute $[\underline{\mathcal{V}}_k, \underline{\mathcal{Y}}_k]$ and $[\underline{\tilde{\mathcal{V}}}_k, \underline{\tilde{\mathcal{Y}}}_k]$ according to (8.24)
  Compute G_k^* according to (8.40)
  Initialize $\{v'_0, \tilde{v}'_0, y'_0, \tilde{y}'_0\}$ according to (8.27)
  for $j = 0$ to $s - 1$ **do**
    $\hat{\delta}_{sk+j} = \tilde{v}'^H_j G_k^* \underline{B}'_k v'_j$
    $\phi_{sk+j} = \hat{\delta}_{sk+j} / \delta_{sk+j}$
    Choose $\gamma_{sk+j}, \tilde{\gamma}_{sk+j} \neq 0$
    $y'_{j+1} = (\underline{B}'_k v'_j - \phi_{sk+j} y'_j)/\gamma_{sk+j}$

    $\tilde{y}'_{j+1} = (\underline{B}'_k \tilde{v}'_j - \overline{\phi}_{sk+j} \tilde{y}'_j)/\tilde{\gamma}_{sk+j}$

    $\delta_{sk+j+1} = \tilde{y}'^H_{j+1} G_k^* y'_{j+1}$
    $\psi_{sk+j} = \overline{\tilde{\gamma}}_{sk+j} \delta_{sk+j+1} / \hat{\delta}_{sk+j} \quad \tilde{\psi}_{sk+j} = \overline{\gamma_{sk+j} \delta_{sk+j+1} / \hat{\delta}_{sk+j}}$
    $v'_{j+1} = y'_{j+1} + \psi_{sk+j} v'_j$

    $\tilde{v}'_{j+1} = \tilde{y}'_{j+1} + \tilde{\psi}_{sk+j} \tilde{v}'_j$

  end for
  Recover $\{v_{sk+s}, \tilde{v}_{sk+s}, y_{sk+s}, \tilde{y}_{sk+s}\}$ according to (8.26)
end for

We now assemble the CA-BIOC method in Algorithm 8.5.

8.3. Linear systems

In this subsection, we derive communication-avoiding variants of the Generalized Minimal Residual method (GMRES) and the biconjugate gradient method (BICG) for solving $n \times n$ nonsymmetric linear systems $Ax = b$. These methods are based on Arnoldi and Lanczos, respectively, presented in Section 8.2. We selected these methods for simplicity, although we have also developed communication-avoiding versions of other KSMs, including communication-avoiding biconjugate gradient stabilized (CA-BICGSTAB) (Carson *et al.* 2013). A slightly different variant of the conjugate gradient method (CG), which uses a single three-term recurrence rather than coupled two-term recurrences, can be found in Hoemmen (2010).

Algorithm 8.6 GMRES

Require: $n \times n$ linear system $Ax = b$, and initial guess x_0
Output: Approximate solution x_m to $Ax = b$
1: $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$, $q_1 = r_0/\beta$
2: **for** $j = 1$ to m **do**
3: $w_j = Aq_j$
4: **for** $i = 1$ to j **do** ▷ Use MGS to orthogonalize w_j against $q_1, \ldots, q_j$
5: $h_{ij} = w_j^H q_i$
6: $w_j = w_j - h_{ij} q_i$
7: **end for**
8: $h_{j+1,j} = \|w_j\|_2$
9: $q_{j+1} = w_j / h_{j+1,j}$
10: $y_j = \text{argmin}_y \|\underline{H} y - \beta e_1\|_2$
11: $x_j = x_0 + Q y_j$
12: **end for**

To derive methods in this section, we can make use of CA-Arnoldi(s, t) and CA-BIOC from the previous subsection. It only remains to determine how to choose the approximate solutions x_j from the Krylov subspaces produced by CA-Arnoldi(s, t) or CA-BIOC.

8.3.1. Arnoldi-based solvers

The Generalized Minimal Residual method (GMRES) of Saad and Schultz (1986), shown in Algorithm 8.6, is a Krylov subspace method for solving nonsymmetric linear systems. It does so by choosing the solution update from the span of the Arnoldi basis vectors that minimizes the 2-norm residual error. This entails solving a least-squares problem with the Arnoldi upper Hessenberg matrix, which is $(m + 1) \times m$ at iteration m of GMRES. Saad and Schultz's formulation of GMRES combines (conventional, MGS-based) Arnoldi with a Givens rotation scheme for maintaining a QR factorization of the upper Hessenberg matrix at each iteration. The Givens rotation scheme (not explicitly shown in Algorithm 8.6) exposes the residual error of the least-squares problem at every iteration j, which (in exact arithmetic) is the GMRES residual error $\|b - Ax_j\|_2$ in iteration j. Thus GMRES need not compute the approximate solution x_j in order to estimate convergence.

We can compute exactly the same approximate solution as m iterations of Saad and Schultz's GMRES if we perform m iterations of CA-Arnoldi(s, t), where s and t are chosen so that $m = s \cdot t$. This results in an $(st + 1) \times st$ upper Hessenberg matrix $\underline{\mathfrak{H}}_{t-1}$. We can even use Givens rotations to maintain its QR factorization and get the same convergence estimate as in Saad and Schultz (1986). The only difference is that we have to perform s Givens

Algorithm 8.7 CA-GMRES

Require: $n \times n$ linear system $Ax = b$, and initial guess x_0
Output: Approximate solution x_{st} to $Ax = b$

1: $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$, $q_1 = r_0/\beta$
2: **for** $k = 0$ to $t - 1$ **do**
3: Fix basis conversion matrix $\underline{B}_k$
4: Compute $\acute{\underline{V}}_k$ using matrix powers kernel
5: **if** $k = 0$ **then**
6: Compute QR factorization $\underline{V}_0 = \underline{Q}_0 \underline{R}$
7: Set $\underline{\mathfrak{Q}}_0 = \underline{Q}_0$ and $\underline{\mathfrak{H}}_0 = \underline{R}_0 \underline{B}_0 R_0^{-1}$
8: **else**
9: $\acute{\underline{\mathfrak{R}}}_{k-1,k} = \underline{\mathfrak{Q}}_{k-1}^H \acute{\underline{V}}_k$
10: $\acute{\underline{V}}_k' = \acute{\underline{V}}_k - \underline{\mathfrak{Q}}_{k-1} \acute{\underline{\mathfrak{R}}}_{k-1,k}$
11: Compute QR factorization $\acute{\underline{V}}_k' = \acute{\underline{Q}}_k \acute{\underline{R}}_k$
12: Update $\underline{\mathfrak{Q}}_k$ and $\underline{\mathfrak{R}}_k$ by (8.14) and (8.15), resp.
13: Update $\underline{\mathfrak{H}}_k$ by (8.17)
14: **end if**
15: $y_{s(k+1)} = \text{argmin}_y \|\underline{\mathfrak{H}}_k y - \beta e_1\|_2$
16: $x_{s(k+1)} = x_0 + \mathfrak{Q}_k y_k$
17: **end for**

rotations at a time, since CA-Arnoldi(s, t) adds s columns at a time to the upper Hessenberg matrix with every outer iteration. The Givens rotations technique has little effect on raw performance (one would normally solve the least-squares problem with a QR factorization and use Givens rotations to factor the upper Hessenberg matrix anyway), but it offers an inexpensive convergence test at every outer iteration. The resulting algorithm, based on CA-Arnoldi(s, t) and equivalent to conventional GMRES in exact arithmetic, we call *communication-avoiding GMRES* or CA-GMRES (Algorithm 8.7). The notation used is the same as in Section 8.2.1.

CA-GMRES includes all of CA-Arnoldi(s, t) as shown in Algorithm 8.3, and, at every outer iteration k, locally solves the least-squares problem

$$y_k = \text{argmin}_y \|\underline{\mathfrak{H}}_k y - \beta e_1\|_2 \tag{8.45}$$

to compute a new approximate solution $x_{s(k+1)} = x_0 + \mathfrak{Q}_k y_k$. Here, $\beta = \|b - Ax_0\|_2$, the 2-norm of the initial residual. Although not shown here, one can update the QR factorization of $\underline{\mathfrak{H}}_k$ by first applying all the previous $sk+1$ Givens rotations, say $G_1, \ldots, G_{sk+1}$, to the new s columns of $\underline{\mathfrak{H}}_k$, and then applying s more Givens rotations $G_{sk+2}, \ldots, G_{s(k+1)+1}$ to $\underline{H}_k$ to reduce it to upper triangular form. The Givens rotations are also applied to the right-hand side of the least-squares problem, resulting in what we denote

Algorithm 8.8 BICG

Require: Initial approximation x_0 for solving $Ax = b$, let $v_0 = r_0 = b - Ax_0$
Output: Approximate solution x_{m+1} to $Ax = b$ with residual $r_{m+1} = b - Ax_{m+1}$

Choose $\tilde{r}_0$ arbitrarily s.t. $\delta_0 = \tilde{r}_0^H r_0 \neq 0$, and let $\tilde{v}_0 = \tilde{r}_0$
for $m = 0, 1, 2, \ldots$, until convergence **do**
  $\alpha_m = \delta_m / \tilde{v}_m^H A v_m$
  $x_{m+1} = x_m + \alpha_m v_m$
  $r_{m+1} = r_m - \alpha_m A v_m$
  $\tilde{r}_{m+1} = \tilde{r}_m - \overline{\alpha_m} A^H \tilde{v}_m$
  $\delta_{m+1} = \tilde{r}_{m+1}^H r_{m+1}$
  $\beta_m = \delta_{m+1} / \delta_m$
  $v_{m+1} = r_{m+1} + \beta_m v_m$
  $\tilde{v}_{m+1} = \tilde{r}_{m+1} + \overline{\beta_m} \tilde{v}_m$
end for

at outer iteration k by ζ_k. The last component of ζ_k, in exact arithmetic, equals the 2-norm of the absolute residual error $b - Ax_{s(k+1)}$. See Hoemmen (2010) for details.

8.3.2. Lanczos-based solvers

We now derive communication-avoiding BICG (CA-BICG) for solving nonsymmetric linear systems $Ax = b$. We first review BICG (Algorithm 8.8) and demonstrate its relation to BIOC; for a thorough treatment see Gutknecht (1997), for example. It follows that we can easily obtain CA-BICG from CA-BIOC.

The BICG method starts with an initial solution guess x_0 and corresponding residuals $r_0 = \tilde{r}_0 = b - Ax_0$. In each iteration m, the solution x_m is updated by a vector selected from $\mathcal{K}_m(A, r_0)$ such that $(b - Ax_m) \perp \mathcal{K}_m(A^H, \tilde{r}_0)$, the so-called Petrov–Galerkin condition.

The solution can then be written as $x_m = x_0 + Y_m c_m$, where Y_m and $\tilde{Y}_m$ are the matrices of biorthogonal basis vectors of $\mathcal{K}_m(A, r_0)$ and $\mathcal{K}_m(A^H, \tilde{r}_0)$, respectively, produced by BIOC with starting guess $y_0 = \tilde{y}_0 = r_0$.

Enforcing the Petrov–Galerkin condition gives $c_m = T_m^{-1} e_1$. Then, using (8.21) and (8.22),

$$\begin{aligned} x_{m+1} &= x_0 + Y_m T_m^{-1} e_1 \\ &= x_0 + Y_m U_m^{-1} L_m^{-1} e_1 \\ &= x_0 + V_m L_m^{-1} e_1 \\ &= -\frac{\phi_m}{\gamma_m} x_m - \frac{1}{\gamma_m} v_m. \end{aligned}$$

Algorithm 8.9 Communication-avoiding BICG

Require: Initial approximation x_0 for solving $Ax = b$, let $v_0 = r_0 = b - Ax_0$
Output: Approximate solution $x_{s(k+1)}$ to $Ax = b$ with residual $r_{s(k+1)} = b - Ax_{s(k+1)}$

Choose $\tilde{r}_0$ arbitrarily s.t. $\delta_0 = \tilde{r}_0^H r_0 \neq 0$, and let $\tilde{v}_0 = \tilde{r}_0$
for $k = 0, 1, 2, \ldots$, until convergence **do**
  Assemble $\underline{B}'_k$ according to (8.35) and (8.33)
  Compute $[\underline{\mathcal{V}}_k, \underline{\mathcal{R}}_k]$ and $[\underline{\tilde{\mathcal{V}}}_k, \underline{\tilde{\mathcal{R}}}_s]$ according to (8.24)
  Compute G_k^* according to (8.40)
  Initialize $\{v'_0, \tilde{v}'_0, r'_0, \tilde{r}'_0, x_0\}$ according to (8.27)
  for $j = 0$ to $s - 1$ **do**
    $\alpha_{sk+j} = \delta_{sk+j} / \tilde{v}_j'^H G_k^* \underline{B}'_k v'_j$
    $x'_{j+1} = x'_j + \alpha_{sk+j} v'_j$
    $r'_{j+1} = r'_j - \alpha_{sk+j} \underline{B}'_k v'_j$
    $\tilde{r}'_{j+1} = \tilde{r}'_j - \overline{\alpha_{sk+j}} \underline{B}'_k \tilde{v}'_j$
    $\delta_{sk+j+1} = \tilde{r}_{j+1}'^H G_k^* r'_{j+1}$
    $\beta_{sk+j} = \delta_{sk+j+1} / \delta_{sk+j}$
    $v'_{j+1} = r'_{j+1} + \beta_{sk+j} v'_j$
    $\tilde{v}'_{j+1} = \tilde{r}'_{j+1} + \overline{\beta_{sk+j}} \tilde{v}'_j$
  end for
  Recover $\{r_{sk+s}, \tilde{r}_{sk+s}, v_{sk+s}, \tilde{v}_{sk+s}, x_{sk+s}\}$ according to (8.26) and (8.48)
end for

The updates to r_{m+1} then become

$$r_{m+1} = r_0 - AV_m L_m^{-1} e_1 = -\frac{\phi_m}{\gamma_m} r_m + \frac{1}{\gamma_m} A v_m,$$

and similarly for $\tilde{r}_m$.

To meet the BICG consistency condition that $\rho(0) = 1$ for the BICG residual vectors, the matrix $\underline{T}_m$ (and $\underline{\tilde{T}}_m$) must have zero-sum columns. Since $\underline{L}_m = \underline{T}_m U_m^{-1}$, $\underline{L}_m$ inherits this property from $\underline{T}_m$, and thus we must set the scalars $\gamma_i = -\phi_i = -\tilde{v}_i^H A v_i / \tilde{r}_i^H r_i$. Similar relations hold for the left residual vectors, that is, $\tilde{\gamma}_i = -\tilde{\phi}_i$. Updates to x_{m+1} and r_{m+1} are then

$$x_{m+1} = x_m + \alpha_m v_m \quad \text{and} \quad r_{m+1} = r_m - \alpha_m A v_m, \tag{8.46}$$

where $\alpha_m = 1/\phi_m = \tilde{r}_i^H r_i / \tilde{v}_i^H, A v_i$. From BIOC, we have the relation $R_m = Y_m = V_m U_m$, and thus the biconjugate vectors v_{m+1} are updated, as in BIOC, as

$$v_{m+1} = r_{m+1} - \psi_m v_m = r_{m+1} + \beta_m v_m,$$

where $\beta_m = -\psi_m$ (and similarly for $\tilde{v}_{m+1}$).

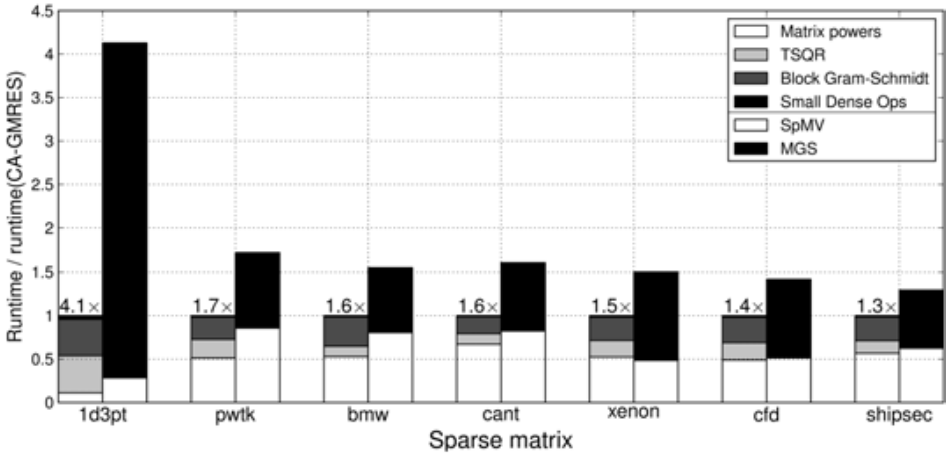


Figure 8.1. Speed-ups for CA-GMRES on Nehalem.

One can therefore obtain BICG by running BIOC with $y_0 = r_0$, $\gamma_i = -\phi_i$, and with the additional computation of x_{m+1} as in (8.46).

Then, to write CA-BICG we can make use of CA-BIOC in the same way. In CA-BICG, we can use (8.40), (8.41) and (8.42) to obtain the required scalar quantities. As before, we let $r'_j, \tilde{r}'_j, v'_j, \tilde{v}'_j$, and now x'_j, represent the coefficients for the BICG vectors in the generated bases. The solution update of (8.46) in the inner loop then becomes

$$x'_{j+1} = x'_j + \alpha_{sk+j} v'_j, \tag{8.47}$$

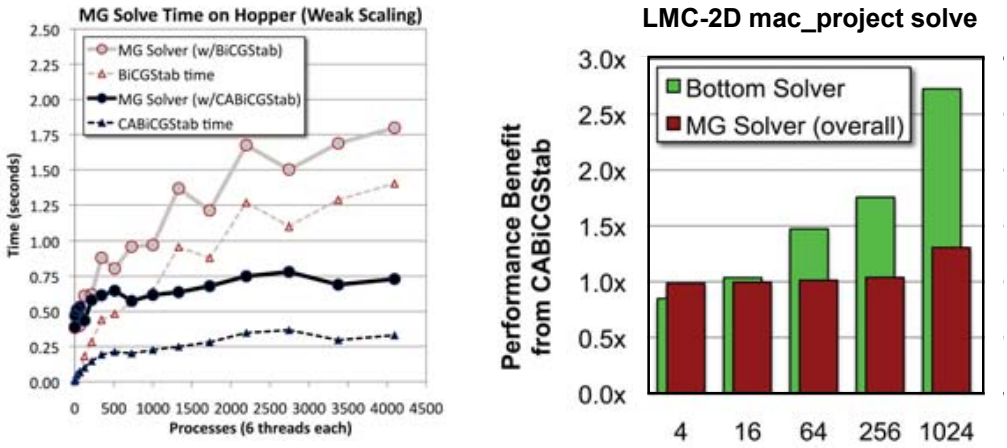
and the iterates can be recovered by

$$x_{sk+j} = x_{sk} + [\underline{\mathcal{V}}_k, \underline{\mathcal{R}}_k] x'_j \tag{8.48}$$

for $0 \le j \le s$. Note that this gives the initial condition $x'_0 = 0_{2s+1}$. We can now present the resulting CA-BICG method, shown in Algorithm 8.9.

8.4. Speed-up results

We now list a few select experiments which have demonstrated the performance benefits of CA-KSMs. Figure 8.1, presented in Hoemmen (2010), compares a shared-memory parallel implementation of CA-GMRES (Algorithm 8.7) with conventional GMRES, on an Intel Nehalem node (both algorithms used eight cores). Both communication-avoiding and conventional algorithm implementations were highly tuned for the best performance; see Mohiyuddin *et al.* (2009) for details. In particular, CA-GMRES used both sequential and parallel techniques to minimize on-chip/off-chip communication and communication between cores. The x-axis shows the matrices tested. These matrices, all available through the University of



(a) (b)

Figure 8.2. (a) Speed-ups for CA-BICGSTAB bottom solver and overall GMG application on Cray XE6. (b) Speed-ups for CA-BICGSTAB bottom solver and overall LMC-2D application on Cray XE6, where the x-axis gives the number of cores used.

Florida Sparse Matrix Collection (Davis and Hu 2011), represent a wide variety of application domains. The vertical axis shows the run time of CA-GMRES (the left bar in each pair) and conventional GMRES (the right bar), normalized so the time of CA-GMRES is 1. For these tests, CA-GMRES achieves a speed-up of 1.3× to 4.1×. Comparing white bars on the left to white bars on the right shows the speed-up due to avoiding communication in the sparse operations (using the matrix powers kernel). Comparing light grey, dark grey, and black bars on the left to black bars on the right shows speed-ups due to avoiding communication in the dense orthogonalization operations. This demonstrates that, for most cases, avoiding communication in both the dense and sparse linear algebra kernels was necessary to obtain the best speed-ups. We note that it was important to cotune the different kernels (especially matrix powers and TSQR), rather than tune them independently, since the optimal parameter choices varied significantly for some test problems.

Figure 8.2(a) shows that CA-BICGSTAB (another BIOC-based algorithm: see Carson *et al.* 2013 for derivation) achieves over a 4× speed-up over conventional BICGSTAB as the bottom solve routine in a geometric multigrid solver (GMG) (Williams *et al.* 2014) run on a Cray XE6 (Hopper at NERSC). Figure 8.2(b) shows over 2.5× speed-ups for CA-BICGSTAB in an application where GMG is used as the solver in a low-Mach-number 2D combustion application (LMC-2D mac_project), leading to an overall 1.3× speed-up of the entire GMG solver on 1024 cores. See Williams *et al.* (2014) for details.

8.5. Numerical behaviour in finite precision

In exact arithmetic, CA-KSMs produce the same iterates as the conventional variants. This means that theoretical rates of convergence for conventional KSMs in exact arithmetic also hold for the corresponding CA-KSMs. It has long been recognized, however, that the finite precision behaviour of CA-KSMs differs greatly from that of their conventional counterparts for $s > 1$. Accumulation of round-off error can have undesirable effects, such as decrease in maximum attainable accuracy, decrease in convergence rate, and even failure to converge in the worst case. In this section, we explore these phenomena and show ways to overcome them.

8.5.1. Convergence

When one considers the performance of iterative methods in practice, one must consider two factors.

(1) The *time per iteration*, which depends on the kernels required, the machine parameters, autotuning/optimization of local operations, as well as matrix structure and partition.

(2) The *number of iterations required for convergence*, that is, until the stopping criterion is met. This quantity depends on the eigenvalue distribution of the system, as well as round-off error in finite precision. In practice, round-off errors resulting from finite precision computation heavily influence the rate of convergence; for example, large round-off errors can lead to loss of (bi)orthogonality between the computed Lanczos vectors, causing convergence to slow.

With these two variables in mind, we can approximate the total time required for an iterative method as:

$$\text{Total time} = (\text{time per iteration}) \times (\text{number of iterations}). \tag{8.49}$$

Both quantities are thus important in devising efficient algorithms. In order to achieve $O(s)$ speed-ups using CA-KSMs, we must not only reduce the time per iteration by $O(s)$, but we must also ensure a convergence rate close to that of the conventional method in finite precision arithmetic. Note that in the case that communication cost dominates the run time, even small s (*e.g.*, $s = 2$) can lead to significant s-fold speed-ups. Choosing such a small s may eliminate most numerical stability concerns, but ideally we would like to be able to choose s larger to improve performance even more.

It has been empirically observed that the convergence rate of s-step methods is tied to the condition number of the constructed Krylov basis: the higher the basis condition number, the slower the convergence rate. Bounds on condition number growth for the monomial basis grow exponentially with s. This limited the range of suitable s values, and thus limited the potential

savings in communication cost, motivating the use of bases known to have better conditioning than the monomials, namely Newton and Chebyshev polynomials: see, for example, Bai *et al.* (1994), Walker (1988) and Joubert and Carey (1992). We discuss the use of these bases in the subsections below.

The three-term recurrence satisfied by polynomials $\{\rho_j(z)\}_{j=0}^s$ in (8.25) can be represented by a tridiagonal matrix $\underline{B}$ (8.33) consisting of the recurrence coefficients. In the subsections below, we show how to choose these coefficients based on properties of A to construct better-conditioned Newton and Chebyshev bases.

8.5.1.1. Newton. The (scaled) Newton polynomials are defined by the recurrence coefficients

$$\hat{\alpha}_j = \theta_j, \quad \hat{\beta}_j = 0, \quad \hat{\gamma}_j = \sigma_j, \tag{8.50}$$

where the σ_j are *scaling factors* and the *shifts* θ_j are chosen to be eigenvalue estimates. If the shifts θ_j were exactly equal to the extreme eigenvalues of A, then computing the polynomials $\rho_j(A)$ would be analogous to working with a smaller and better-conditioned matrix missing those extreme eigenvalues; in practice, approximate θ_j are often enough to obtain the desired effect.

After choosing s shifts, we permute them according to a Leja ordering (see, *e.g.*, Reichel 1990). Informally, the Leja ordering recursively selects each θ_j to maximize a certain measure on the set $\{\theta_i\}_{i \leq j} \subset \mathbb{C}$; this helps to avoid repeated or nearly repeated shifts. Real matrices may have complex eigenvalues, so the Newton basis may introduce complex arithmetic; to avoid this, we use the modified Leja ordering (Hoemmen 2010), which exploits the fact that complex eigenvalues of real matrices occur in complex conjugate pairs. The modified Leja ordering means that for every complex shift $\theta_j \notin \mathbb{R}$ in the sequence, its complex conjugate $\overline{\theta_j}$ is adjacent, and the complex conjugate pairs (θ_j, θ_{j+1}) are permuted so that $\operatorname{Im}(\theta_j) > 0$. This leads to the recurrence coefficients

$$\hat{\alpha}_j = \operatorname{Re}(\theta_j), \quad \hat{\beta}_j = \begin{cases} -\operatorname{Im}(\theta_j)^2 & \theta_j = \overline{\theta_{j+1}} \wedge \operatorname{Im}(\theta_j) > 0, \\ 0 & \text{otherwise}, \end{cases} \quad \hat{\gamma}_j = \sigma_j. \tag{8.51}$$

Both Leja orderings can be computed efficiently. For CA-KSMs, choosing σ_j is challenging: we cannot simply scale each basis vector by its Euclidean norm as it is generated, since this eliminates our communication savings. In our experiments, we pick all scaling factors $\sigma_j = 1$ (*i.e.*, no scaling), instead relying entirely on matrix equilibration, described in Section 8.5.2.1, to keep the spectral radius $\rho(A)$ close to 1. Note that approximate θ_j values can be computed using information produced during the iterations. One could compute initial estimates after the first few iterations and then update them periodically in each new outer loop.

8.5.1.2. Chebyshev. Chebyshev polynomials have a unique minimax property: over all (appropriately scaled and shifted) real polynomials of a specified degree on a specified real interval, the Chebyshev polynomial of that degree minimizes the maximum absolute value on the interval. When their region of definition is extended to ellipses in the complex plane, the Chebyshev polynomials still satisfy the minimax property asymptotically, though not (always) exactly. The minimax property makes Chebyshev polynomials both theoretically and practically useful for Krylov subspace methods.

The Chebyshev basis requires two parameters, complex numbers d and c, where $d \pm c$ are the foci of a bounding ellipse for the spectrum of A. The scaled, shifted, and rotated Chebyshev polynomials $\{\tilde{\tau}_j\}_{j\geq 0}$ can then be written as

$$\tilde{\tau}_j(z) = \tau_j((d-z)/c)/\tau_j(d/c) =: \tau_j((d-z)/c)/\sigma_j, \tag{8.52}$$

where the Chebyshev polynomials (of the first kind) $\{\tau_j\}_{j\geq 0}$ are

$$\begin{aligned} &\tau_0(z) = 1, \quad \tau_1(z) = z, \quad \text{and} \\ &\tau_j(z) = 2z\tau_{j-1}(z) - \tau_{j-2}(z), \quad \text{for } j > 1. \end{aligned} \tag{8.53}$$

Substituting (8.52) into (8.53), we obtain

$$\begin{aligned} &\tilde{\tau}_0(z) = 1, \quad \tilde{\tau}_1(z) = \sigma_0(d-z)/(c\sigma_1), \quad \text{and} \\ &\tilde{\tau}_j(z) = 2\sigma_{j-1}(d-z)\tilde{\tau}_{j-1}(z)/(c\sigma_j) - \sigma_{j-2}\tilde{\tau}_{j-2}(z)/\sigma_j, \quad \text{for } j > 1. \end{aligned} \tag{8.54}$$

Extracting coefficients, we obtain

$$\hat{\alpha}_j = d, \quad \hat{\beta}_j = -c\sigma_j/(2\sigma_{j+1}), \quad \hat{\gamma}_j = \begin{cases} -c\sigma_1/\sigma_0 & j = 0, \\ -c\sigma_{j+1}/(2\sigma_j) & j > 0. \end{cases} \tag{8.55}$$

Note that the transformation $z \to (d-z)/c$ maps the ellipse with foci $f_{1,2} = d \pm c$ to the ellipse with foci at ∓ 1, especially the line segment (f_1, f_2) to $(-1, 1)$. If A is real, then the ellipse is centred on the real axis; thus $d \in \mathbb{R}$, so c is either real or imaginary. In the former case, arithmetic will be real. In the latter case ($c \in i\mathbb{R}$), we avoid complex arithmetic by replacing $c = c/i$. This is equivalent to rotating the ellipses by 90°.

For real matrices, Joubert and Carey (1992) also give a three-term recurrence, where

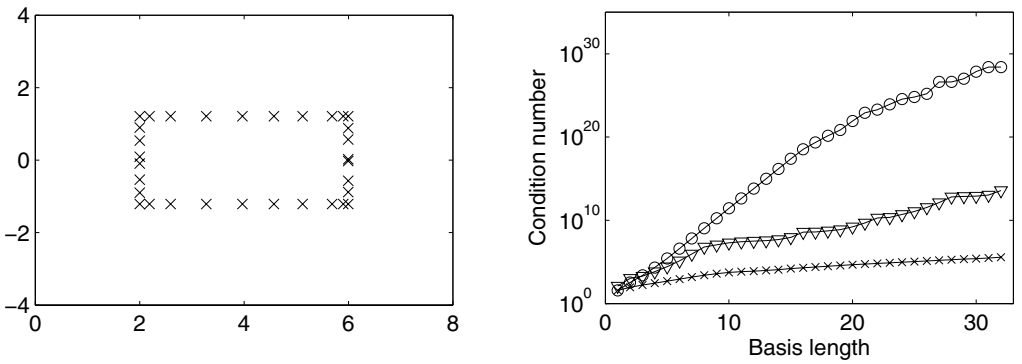
$$\hat{\alpha}_j = d, \quad \hat{\beta}_j = c^2/(4g), \quad \hat{\gamma}_j = \begin{cases} 2g & j = 0, \\ g & j > 0. \end{cases} \tag{8.56}$$

It is assumed that the spectrum is bounded by the rectangle

$$\{z : |\mathrm{Re}\,(z) - d| \leq a, |\mathrm{Im}\,(z)| \leq b\}$$

in the complex plane, where $a \geq 0$, $b \geq 0$, and d are real. Here we choose

$$c = \sqrt{a^2 - b^2}, \quad g = \max\{a, b\}.$$



(a) (b)

Figure 8.3. Basis properties for cdde: (a) computed Leja points (×) of cdde, plotted in the complex plane; (b) basis condition number growth rate. The x-axis denotes the basis length s and the y-axis denotes the basis condition number for monomial (○), Newton (×), and Chebyshev (▽) bases. Leja points in (a) were used to generate the bases shown in (b).

Note that for real-valued matrices, recurrence (8.56) is usually sufficient for capturing the necessary spectral information. As in the Newton case, eigenvalue estimates can be computed from the iterations and used to obtain or refine Chebyshev basis parameters.

8.5.1.3. Example. We now give two examples which illustrate how basis choice can greatly improve the numerical behaviour of CA-BICG. In all tests, the right-hand side b was constructed such that the true solution x is the n-vector with components $x_i = 1/\sqrt{n}$.

Our first test problem comes from the constant-coefficient convection diffusion equation

$$\begin{aligned} -\Delta u + 2p_1 u_x + 2p_2 u_y - p_3 u_y &= f \quad \text{in } [0,1]^2, \\ u &= g \quad \text{on } \partial[0,1]^2. \end{aligned}$$

This problem is widely used for testing and analysing numerical solvers (Bai, Day, Demmel and Dongarra 1997*a*). We discretized the PDE using centred finite difference operators on a 512×512 grid with $(p_1, p_2, p_3) = (25, 600, 250)$, resulting in a nonsymmetric matrix with $n = 262K$, $\text{nnz}(A) = 1.3M$, and condition number ~ 5.5. To select the Leja points, we used the convex hull of the known spectrum, given in Bai *et al.* (1997*a*). This is the best we can expect; in practice, one would use approximate eigenvalues (this is shown in the subsequent example). Figure 8.3 shows the selected Leja points and resulting basis condition number for the monomial, Newton and Chebyshev bases, for basis lengths up to $s = 32$, with a starting vector whose entries are $1/\sqrt{n}$. Figure 8.4 shows convergence results for CA-BICG for the

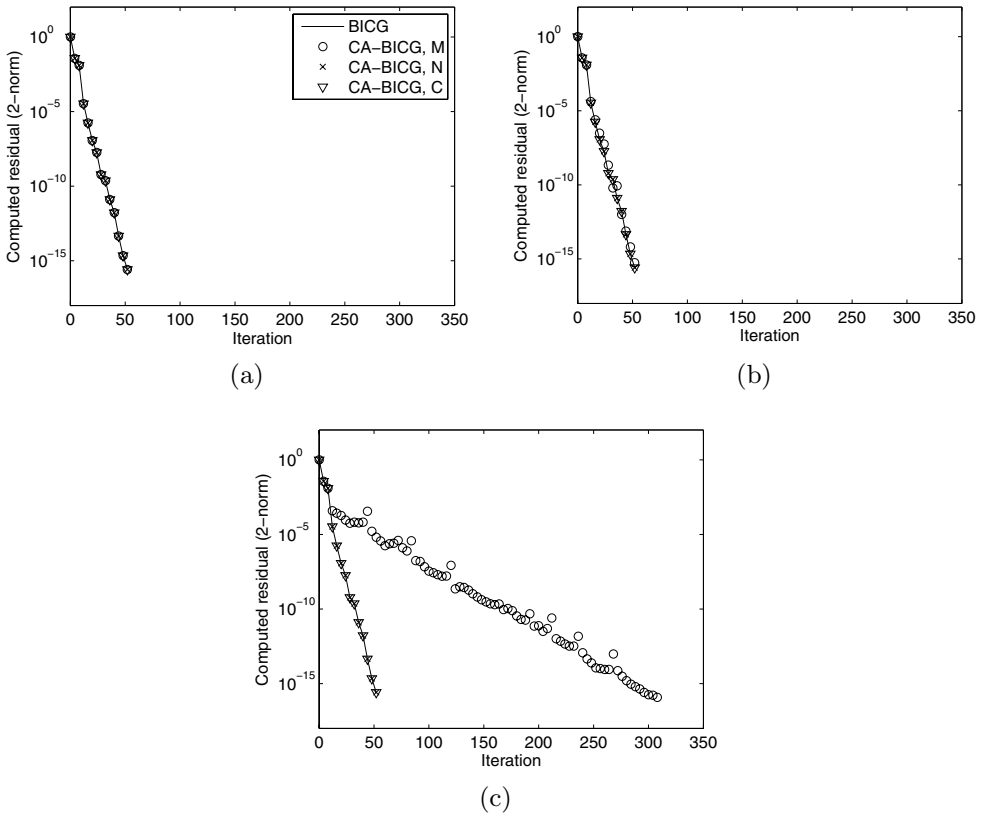
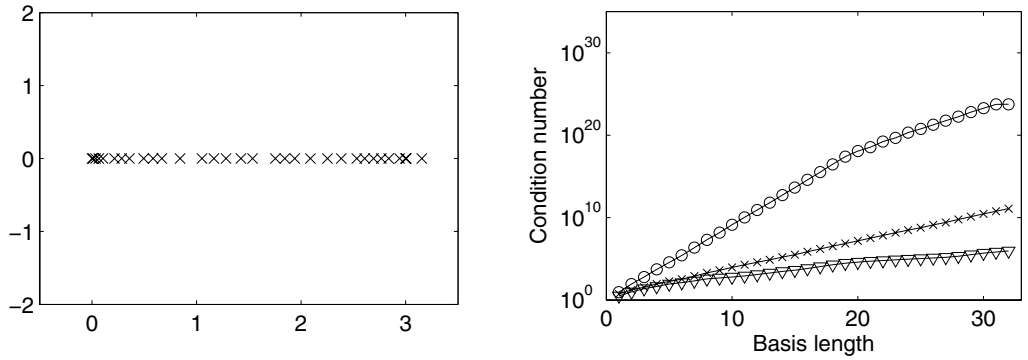


Figure 8.4. Convergence for cdde matrix for (CA)-BICG with various s values: (a) $s = 4$, (b) $s = 8$, (c) $s = 12$. The x-axis denotes the iteration number and the y-axis denotes the 2-norm of the (normalized) updated residual (*i.e.*, $\|r_m\|_2/\|b\|_2$). In the legend, 'M' denotes the monomial basis, 'N' denotes the Newton basis, and 'C' denotes the Chebyshev basis.

different bases, for $s \in \{4, 8, 12\}$, using the same starting vector as the right-hand side. We see that the Newton basis (CA-BICG, N) and the Chebyshev basis (CA-BICG, C) yield indistinguishable results from conventional BICG, but for $s = 12$, the monomial basis (CA-BICG, M) converges much more slowly.

The xenon1 matrix is a nonsymmetric matrix from the University of Florida Sparse Matrix Collection (Davis and Hu 2011). This matrix is used in analysing elastic properties of crystalline compounds (Davis and Hu 2011). Here, $n = 48.6K$ and $\text{nnz}(A) = 1.2M$. This test case is less well-conditioned than the first, with condition number $\sim 1.1 \cdot 10^5$. As a preprocessing step, we performed matrix equilibration, as described in Hoemmen (2010). The xenon1 matrix has all real eigenvalues. Here, we used approximate eigenvalues to generate the basis parameters, computed based on estimates of the maximum and minimum eigenvalues obtained from ARPACK



(a) (b)

Figure 8.5. Basis properties for xenon1: (a) computed Leja points (×) of xenon1, plotted in the complex plane; (b) basis condition number growth rate. The x-axis denotes the basis length s and the y-axis denotes the basis condition number for monomial (○), Newton (×), and Chebyshev (▽) bases. Leja points in (a) were used to generate the bases shown in (b).

(MATLAB `eigs`). Figure 8.5 shows the generated Leja points and resulting basis condition numbers for the monomial, Newton and Chebyshev bases, for basis lengths up to $s = 32$, with a starting vector whose entries are $1/\sqrt{n}$. Figure 8.6 shows convergence results for CA-BICG for the different bases, for $s \in \{4, 8, 12\}$, using the same starting vector as the right-hand side, with results similar to those in in Figure 8.4.

For CA-BICG with the monomial basis, we generally see the convergence rate decrease (relative to the conventional method) as s grows larger. This is due to round-off error in computation of the basis vectors in the matrix powers kernel, which results in a larger perturbation to the finite precision Lanczos recurrence that determines the rate of convergence (see, *e.g.*, Tong and Ye 2000). At some point, when s becomes large, CA-KSMs with the monomial basis will fail to converge due to (near) numerical rank deficiencies in the generated Krylov bases. For both of our test matrices, CA-BICG with the monomial basis fails to converge to the desired tolerance when $s > 12$. Of course limiting $s \leq 12$ limits the potential speed-up from avoiding communication to 12×, but this is still of practical value.

CA-BICG with the Newton and Chebyshev bases can maintain convergence closer to that of the conventional method, even for s as large as 12. This is especially evident if we have a well-conditioned matrix, as in Figure 8.4 (`cdde`). For this matrix, tests with the Newton and Chebyshev bases converge in the same number of iterations as the conventional method, for all tested s values.

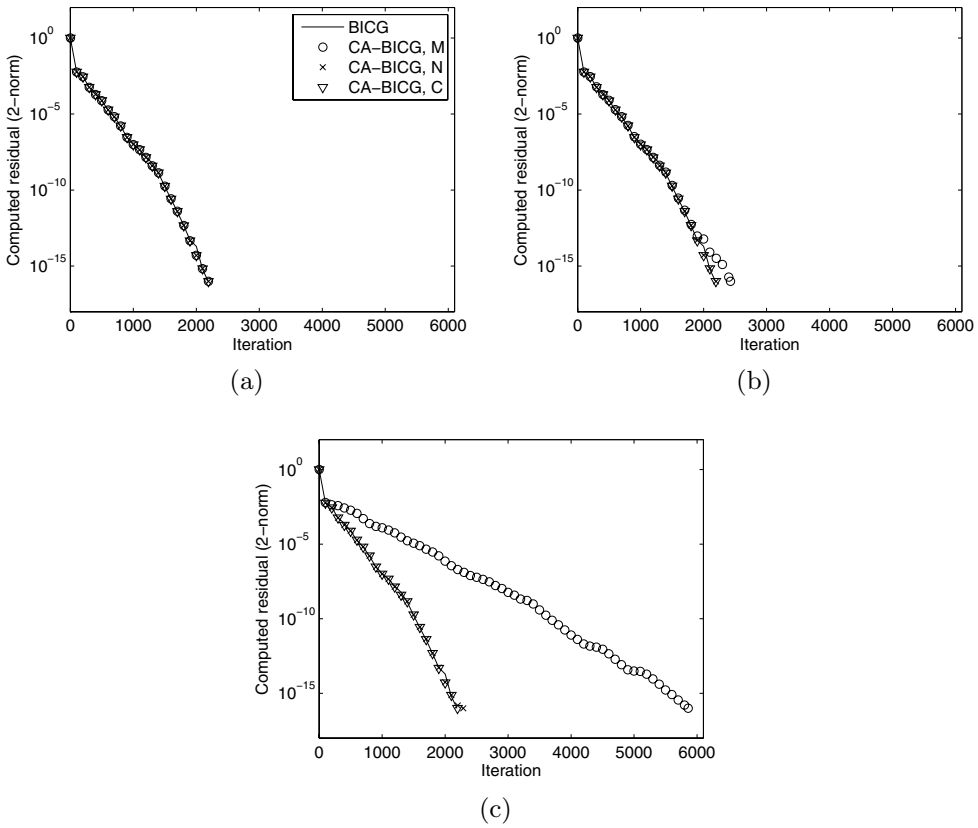


Figure 8.6. Convergence for xenon1 matrix, for (CA)-BICG with various s values: (a) $s = 4$, (b) $s = 8$, (c) $s = 12$. The x-axis denotes the iteration number and the y-axis denotes the 2-norm of the (normalized) updated residual (*i.e.*, $\|r_{m+1}\|_2/\|b\|_2$). In the legend, 'M' denotes the monomial basis, 'N' denotes the Newton basis, and 'C' denotes the Chebyshev basis.

8.5.2. Improving the stability of CA-KSMs

We discuss techniques for improving stability in CA-KSMs below. This includes matrix equilibration, which can be done as a single preprocessing step, as well as residual replacement strategies (Van der Vorst and Ye 1999) and lookahead techniques, which have been adapted from those developed for conventional KSMs; see, for example, Gutknecht and Ressel (2000) or Parlett, Taylor and Liu (1985).

8.5.2.1. Matrix equilibration. Successively scaling (*i.e.*, normalizing) the Krylov vectors as they are generated increases the numerical stability of the basis generation step. As discussed by Hoemmen (2010), successively scaling the basis vectors (*e.g.*, dividing by their Euclidean norms) is not possible in the CA variants, as it reintroduces global communication between SpMV operations. As an alternative, one can perform matrix equilibration. For nonsymmetric matrices, this involves applying diagonal row and column

scalings, D_r and D_c, such that each row and column of the equilibrated matrix $D_r A D_c$ has norm (close to) one. This is performed once offline before running the CA-KSM. Results in Hoemmen (2010) indicate that this technique is an effective alternative to successively scaling the basis vectors.

8.5.2.2. Residual replacement strategies for improving accuracy. The accuracy of conventional KSMs in finite precision has been studied extensively in the literature; see, for example, Greenbaum (1997*a*), Gutknecht and Strakoš (2000), Meurant and Strakoš (2006), Meurant (2006), Sleijpen and van der Vorst (1996) and Van der Vorst and Ye (1999). Such analyses stem from the observation that updates to x_m and r_m have different round-off patterns in finite precision. That is, the expression for x_m does not depend on r_m, nor does the expression for r_m depend on x_m. Therefore, computational errors made in x_m are not self-correcting. These errors can accumulate over many iterations and cause deviation of the *true residual*, $b - Ax_m$, and the *updated residual*, r_m. Writing the true residual as $b - Ax_m = r_m + (b - Ax_m - r_m)$, we can bound its norm by

$$\|b - Ax_m\| \leq \|r_m\| + \|b - Ax_m - r_m\|.$$

When the updated residual r_m is much larger than $b - Ax_m - r_m$, the true residual and the updated residual will be of similar magnitude. However, as the updated residual converges, that is, $\|r_m\| \to 0$, the size of the true residual depends on $\|b - Ax_m - r_m\|$. This term denotes the size of the deviation between the true and updated residuals. If this deviation grows large, it can limit the *maximum attainable accuracy*, that is, the accuracy with which we can solve $Ax = b$ on a computer with unit round-off ϵ.

Like conventional implementations, CA-KSMs suffer from round-off error in finite precision, which can decrease the maximum attainable accuracy of the solution. A quantitative analysis of round-off error in CA-KSMs can be found in Carson and Demmel (2014). Based on the work of Van der Vorst and Ye (1999) for conventional KSMs, we have explored implicit residual replacement strategies for CA-KSMs as a method to limit the deviation of true and computed residuals when high accuracy is required: see Carson and Demmel (2014).

We demonstrate the possible improvement in accuracy with a brief numerical example from Davis and Hu (2011), shown in Figure 8.7. We compare conventional CG with CA-CG, both with and without residual replacement, for $s = [4, 8, 12]$, and with the monomial, Newton and Chebyshev bases: see, for example, Philippe and Reichel (2012). Coefficients for the Newton and Chebyshev bases were computed using Leja-ordered points obtained from $O(s)$ Ritz value estimates, as described in Carson *et al.* (2013), Hoemmen (2010) and Philippe and Reichel (2012). We used row and column scaling to equilibrate the input matrix A, preserving symmetry, as

described in Hoemmen (2010). The right-hand side b was set such that $\|x\|_2 = 1$, $x_i = 1/\sqrt{N}$, and used the zero vector as the initial guess. All tests were performed in double precision, that is, $\epsilon \approx 10^{-16}$. In each test, the convergence criterion used was $\|r_{sk+j}\|/\|r_0\| \leq 10^{-16}$. Since $r_0 = b$ and $\|b\| \leq \|A\|\|x\|$,

$$\|r_{sk+j}\|/\|r_0\| \leq 10^{-16}$$

implies

$$\|r_{sk+j}\| = O(\epsilon)\|A\|\|x\|.$$

Figure 8.7(a,c,e) shows CA-CG, and Figure 8.7(b,d,f) shows CA-CG with residual replacement. Figures 8.7(a,b), 8.7(c,d) and 8.7(e,f) show tests run with $s = 4$, 8 and 12, respectively. For comparison, we plot conventional CG and conventional CG with replacement using algorithms given in Van der Vorst and Ye (1999). The x-axis is the iteration number and the y-axis is the 2-norm of the quantities listed in the legend, where 'true' is the true residual $b - Ax_m$, 'upd' is the updated residual r_m, and 'M', 'N' and 'C' denote tests with monomial, Newton and Chebyshev bases, respectively, and 'CA-CG-RR' denotes CA-CG with residual replacement.

In Figure 8.7(a,c,e) we see that the updated and true residuals match until they are fairly small, after which the updated residual keeps decreasing where the true residual stagnates (except for the monomial basis with $s = 12$, which does not converge at all).

In Figure 8.7(b,d,f), when the updated residual converges to $O(\epsilon)\|A\|\|x\|$, the residual replacement scheme for CA-CG improves convergence of the true residual to within $O(\epsilon)\|A\|\|x\|$. The highest number of replacements, which incur the cost of an additional outer loop in terms of communication (a matrix powers computation and orthogonalization operation), for any test was 12 (monomial with $s = 8$), less than 1% of the total number of iterations. Since the number of residual replacement steps is small relative to the total number of iterations, the communication cost of performing residual replacements is negligible in the context of CA-CG in this case.

8.5.2.3. Consistent/inconsistent formulations and lookahead. The version of conventional BICG on which CA-BICG (Algorithm 8.9) is based is called a *consistent* formulation, since the Lanczos vectors $\{r_m\}$ are restricted to be the unscaled residuals of the current candidate solution. In an *inconsistent* formulation, the residual vectors may be scaled arbitrarily. This flexibility gives more control over the sizes of the iterates and scalar coefficients, and thus the bounds on round-off error (Gutknecht 1997). Since they only involve a small amount of additional scalar computation, we can derive communication-avoiding formulations of the inconsistent BICG variant introduced in Gutknecht (1997) and Gutknecht and Ressel (2000) in the same way as we obtained Algorithm 8.9.

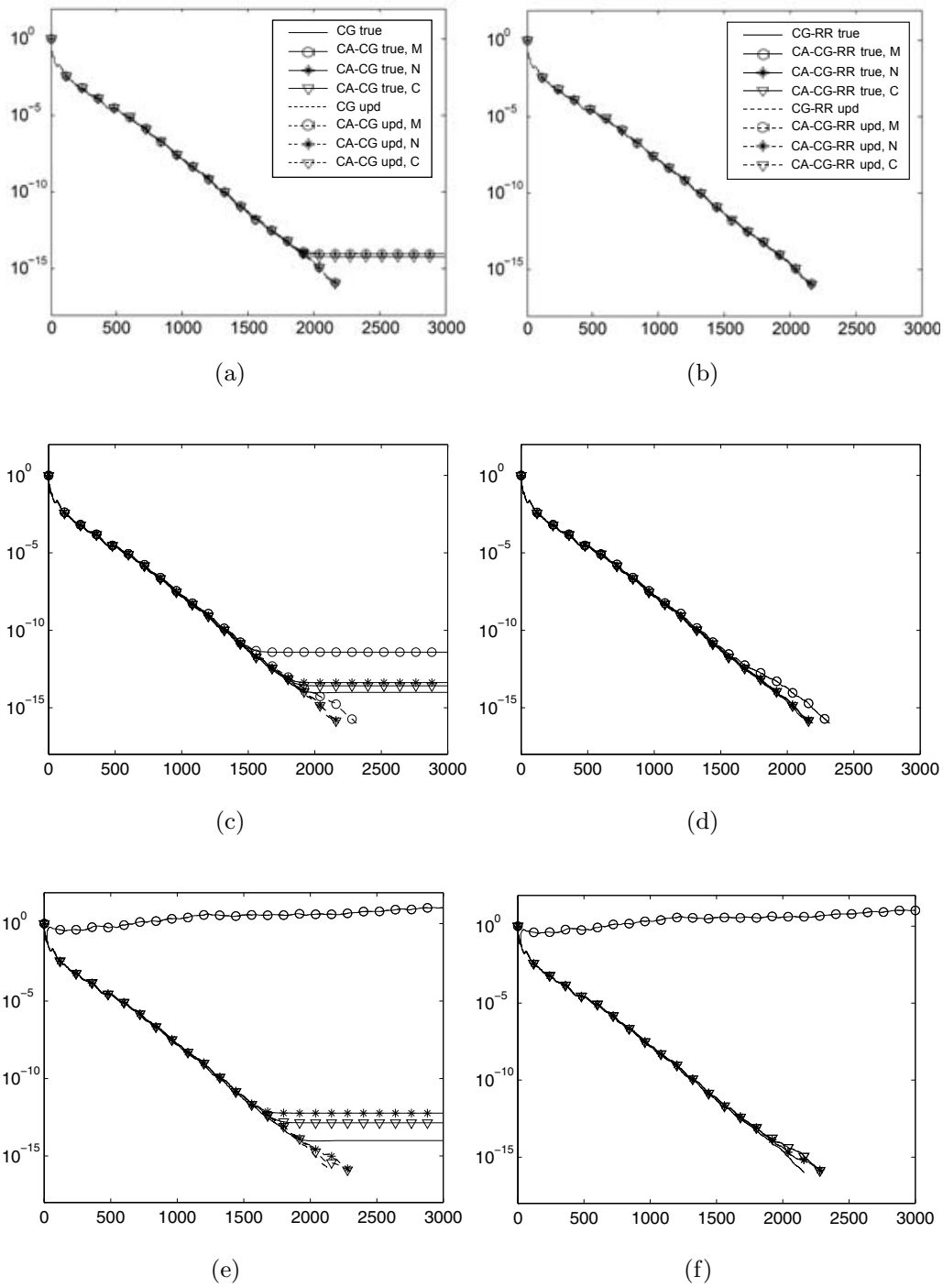


Figure 8.7. Convergence results for consph, an FEM/spheres problem: $n = 8.3 \cdot 10^4$, $\text{nnz} = 6.0 \cdot 10^6$, $\kappa(A) = 9.7 \cdot 10^3$, $\|A\|_2 = 9.7$.

Inconsistent formulations do not, however, help in the case of a Lanczos breakdown. Lookahead techniques may allow nonsymmetric Lanczos-type solvers, including BICG, to continue past a Lanczos breakdown. Kim and Chronopoulos (1992) have noted that for their s-step nonsymmetric Lanczos algorithm, in exact arithmetic, if breakdown occurs at iteration m in the conventional algorithm, then the same breakdown will only occur in the s-step variant if m is a multiple of s. They also comment that lookahead in the exact arithmetic s-step method amounts to ensuring that no blocked inner products between basis matrices have determinant zero. Hoemmen (2010), however, has commented that the situation is more complicated in finite precision, as it is difficult to determine which type of breakdown occurred.

Nonsymmetric Lanczos-based CA-KSMs can thus exploit the matrix powers optimization for two purposes: for increasing performance of the algorithm, and increasing stability by use of lookahead techniques to detect breakdown (Hoemmen 2010). As the s-step structure of our CA algorithms is similar to the technique used in lookahead algorithms, we can add communication-avoiding lookahead to our CA-KSMs for little additional computational cost. Preliminary experiments are promising.

8.5.2.4. Reorthogonalization. In finite precision, the vectors produced by Arnoldi- or Lanczos-based KSMs may lose orthogonality (or biorthogonality), and *reorthogonalization* may be necessary. Reorthogonalization is typically implemented as a second orthogonalization phase, following the usual Gram–Schmidt step.

All of the Gram–Schmidt algorithms discussed in Section 8.2.1.2 are equivalent in exact arithmetic, as long as the input matrix V has full column rank. However, in finite precision arithmetic, the approaches vary in accuracy; we refer to Stewart (2008) for a survey. If we think of Gram–Schmidt as computing the QR factorization $V = QR$, then there are at least two ways of defining the accuracy of the orthogonalization:

- the residual error $\|V - Q \cdot R\|$, or
- orthogonality of the columns of Q, *e.g.*, $\|I - Q^H Q\|$.

Stewart explains that almost all orthogonalization methods of interest guarantee a small residual error under certain conditions (such as V being of full numerical rank). However, fewer algorithms are able to guarantee that the vectors they produce are orthogonal to machine precision. In exact arithmetic, for all Gram–Schmidt schemes, $\|I - Q^H Q\| = 0$ as long as the input vectors are linearly independent. In finite precision computations, however, this norm many grow larger and larger, at least until the supposedly orthogonal vectors become linearly dependent. Schemes for improving the stability of orthogonalization schemes thus focus on bounding the loss of orthogonality. Note that the latter norm is feasible to compute in the usual

Table 8.1. Upper bounds on deviation from orthogonality of the Q factor from various QR factorization algorithms of the $m \times n$ matrix V (with $m \geq n$). Machine precision is ε. CGS2 means CGS with one full reorthogonalization pass and MGS2 means MGS with one full reorthogonalization pass. Cholesky–QR means computing the Cholesky factorization $V^H V = LL^H$ and computing $Q = VL^{-H}$.

Algorithm	$\|I - Q^H Q\|_2$ upper bound	Reference(s)
Householder–QR	$O(\varepsilon)$	[1]
TSQR	$O(\varepsilon)$	[2]
CGS2 or MGS2	$O(\varepsilon)$	[3, 4]
MGS	$O(\varepsilon\kappa(A))$	[5]
Cholesky–QR	$O(\varepsilon\kappa(A)^2)$	[6]
CGS	$O(\varepsilon\kappa(A)^{n-1})$	[7, 8]

[1] Golub and Van Loan (1996)
[2] Demmel *et al.* (2008*a*)
[3] Abdelmalek (1971)
[4] Kiełbasinski (1974)
[5] Björck (1967)
[6] Stathopoulos and Wu (2002)
[7] Kiełbasinski (1974)
[8] Smoktunowicz *et al.* (2006)

case of interest, where Q has many more rows than columns (so that the matrix $I - Q^H Q$ has small dimensions).

Demmel, Grigori, Hoemmen and Langou (2008*a*), in their Table 11, summarize the accuracy of various orthogonalization schemes, quoting Smoktunowicz, Barlow and Langou (2006) and Giraud, Langou and Rozloznik (2005), among others (which those works in turn cite). The various orthogonalization schemes summarized in Demmel *et al.* (2008*a*) include Householder–QR, TSQR, MGS and CGS (with and without reorthogonalization), and Cholesky–QR (which entails computing the Cholesky factorization $V^H V = LL^H$ and then $Q = VL^{-H}$). We repeat the summary in this section as Table 8.1. Note that Householder–QR and TSQR produce Q factors that are *unconditionally orthogonal*, that is, that are orthogonal whether or not the matrix V is numerically full rank. We do not make that distinction in our Table 8.1, because in an s-step Krylov method, the s-step basis should always be numerically full rank (ignoring breakdown). Otherwise orthogonalizing the basis may produce vectors not in the desired Krylov subspace.

In CGS and MGS, reorthogonalization is not always necessary. One may choose to do it anyway, to be safe and also to avoid the expense or trouble of checking: this is called *full reorthogonalization*. To avoid unnecessary reorthogonalization, one may perform *selective reorthogonalization*. There,

one tests each vector before and after orthogonalizing it against the previously orthogonalized vectors. If the test fails, one reorthogonalizes it against all the previously orthogonalized vectors. A typical criterion involves comparing the norm of the vector before and after the orthogonalization pass: if it drops by more than a certain factor, then the result may be mostly noise and thus reorthogonalization is required. For a review and evaluation of various reorthogonalization criteria, see Giraud and Langou (2003). There are also reorthogonalization tests specific to Krylov subspace methods such as symmetric Lanczos: see Bai and Day (2000) for an overview.

We refer to Hoemmen (2010) for a more thorough discussion of reorthogonalization for KSMs, especially communication cost considerations. However, we remark on one important detail for the blocked approaches: detecting when the input V is rank-deficient. Da Cunha, Becker and Patterson (2002) observed that one can construct a rank-revealing factorization algorithm for tall and skinny matrices, using any fast and accurate QR factorization algorithm (such as the one they propose, which, as we discuss in Demmel *et al.* (2012), is a precursor to our TSQR). One first computes the thin QR factorization using the fast and accurate algorithm, and then one applies any suitable rank-revealing decomposition to the resulting small R factor. We call 'RR-TSQR' the resulting rank-revealing decomposition, when the fast QR factorization algorithm used is TSQR; this approach was outlined earlier in Section 3.3.5. In Hoemmen (2010) we describe in detail how to use RR-TSQR to construct a robust version of BGS.

8.6. Communication-avoiding preconditioning

The matrix powers kernel summarized in the previous section computes $[p_0(A)v, p_1(A)v, \ldots, p_s(A)v]$ for a sparse matrix A, a vector v, and a set of polynomials $p_0(z), \ldots, p_s(z)$ with certain properties. The vectors produced form a basis for a Krylov subspace $\text{span}\{v, Av, A^2v, \ldots, A^sv\}$ in exact arithmetic, assuming that the Krylov subspace has dimension at least $s+1$. This kernel is useful for implementing unpreconditioned iterative methods for solving $Ax = \lambda x$ and $Ax = b$. In practice, though, solving $Ax = b$ efficiently with a Krylov method often calls for *preconditioning*. Including a preconditioner changes the required matrix powers kernel, in a way that depends on the iterative method and the form of preconditioning. In this section, we discuss the new kernels required and present types of preconditioners known to be compatible with the communication-avoiding approach. Given a preconditioned matrix powers kernel, one can then substitute it directly in the CA-KSMs derived above to obtain their preconditioned versions.

Preconditioning transforms the linear system $Ax = b$ into another one, with the goal of reducing the number of iterations required for a Krylov method to attain the desired accuracy; see, for example, Barrett *et al.*

(1994), Greenbaum (1997*b*) and Saad (2003). The form of the resulting linear system depends on the type of preconditioner. If M is some nonsingular operator of the same dimensions as A, then *left preconditioning* changes the system to $M^{-1}Ax = M^{-1}b$, and *right preconditioning* changes the system to $AM^{-1}(Mx) = b$. The intuition in either case is that if $M^{-1}A \approx I$ in some sense (see the next paragraph), then the iterative method converges quickly.[11] Similarly, if M can be factored as $M = M_1M_2$, then *split preconditioning* changes the system to $M_1^{-1}AM_2^{-1}(M_2x) = M_1^{-1}b$. (Here, the expression M_2x is interpreted as 'Solve $M_1^{-1}AM_2^{-1}y = M_1^{-1}b$ using the iterative method, and then solve $M_2x = y$'.) If A and M are self-adjoint and $M_1 = M_2^H$ (as in, for example, an incomplete Cholesky decomposition), then $M_1^{-1}AM_2^{-1}$ is also self-adjoint. We define the *preconditioned matrix* for a left preconditioner as $M^{-1}A$, for a right preconditioner as AM^{-1}, and for a split preconditioner as $M_1^{-1}AM_2^{-1}$. In all cases, in this section we denote the preconditioned matrix by $\tilde{A}$. The context will indicate whether this means left, right, or split preconditioning.

Behind the statement $M^{-1}A \approx I$ lies an entire field of study, which we cannot hope to summarize here. Often, developing an effective preconditioner requires domain-specific knowledge, either of the physical system which A represents, or of the continuous mathematical problem which the discrete operator A approximates. There is no 'general preconditioner' that almost always works well.

8.6.1. New kernels

Preconditioning changes the linear system, thus changing the Krylov subspace(s) which the iterative method builds. Therefore, preconditioning calls for new matrix powers kernels. Different types of iterative methods require different kernels. For example, for the preconditioned version of CA-GMRES (see Hoemmen 2010), the required kernel is

$$\underline{V} = [p_0(\tilde{A})v, p_1(\tilde{A})v, \ldots, p_s(\tilde{A})v]. \tag{8.57}$$

For the preconditioned symmetric Lanczos and preconditioned CA-CG algorithms (see Hoemmen 2010) that require A to be self-adjoint, the kernel(s) required depend on whether left, right, or split preconditioning is used. For a split self-adjoint preconditioner with $M_1 = M_2^H$, the kernel is the same as in equation (8.57), and only the preconditioned matrix $\tilde{A}$ is different. (We do not consider the case where A is self-adjoint but $M_1 \neq M_2^H$.) For left preconditioning, two kernels are needed: one for the *left-side basis*,

$$\underline{V} = [p_0(M^{-1}A)v, p_1(M^{-1}A)v, \ldots, p_s(M^{-1}A)v], \tag{8.58}$$

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

and one for the *right-side basis*,

$$\underline{W} = [p_0(AM^{-1})w, p_1(AM^{-1})w, \ldots, p_s(AM^{-1})w], \tag{8.59}$$

where the starting vectors v and w satisfy $Mv = w$. For right preconditioning, the kernels needed are analogous.

A straightforward implementation of each of the kernels (8.57), (8.58), and (8.59) would involve s sparse matrix–vector products with the matrix A, and s interleaved preconditioner solves (or $2s$, in the case of the split preconditioner). In this section, we present classes of preconditioners for which the left-preconditioned kernel can be computed with much less communication. In particular, we can compute each kernel in parallel for only $1 + o(1)$ times more messages than a single SpMV with A and a single preconditioner solve. We can also compute the left-preconditioned kernel for only $1 + o(1)$ times more words transferred between levels of the memory hierarchy than a single SpMV with A and a single preconditioner solve. (The right-preconditioned kernel uses an analogous technique, so we do not describe it here.)

8.6.2. Exploiting sparsity structure

One approach to accelerating the preconditioned kernel exploits sparsity in the preconditioned system. This approach works when A is sparse enough for the matrix powers kernel to be efficient, and the preconditioner is also sparse enough, such as diagonal or block diagonal preconditioners with small blocks. It may also work for more complicated preconditioners that can be constrained to have the same or nearly the same sparsity pattern as the matrix A, such as sparse approximate inverse (SPAI) preconditioners (see Chow 2000).

All the techniques described in Section 7 can be applied in order to compute the preconditioned kernel, if we know the sparsity structure of the preconditioned matrices involved. For example, for the split-preconditioned kernel (equation (8.57)), the sparsity structure of $M_1^{-1}AM_2^{-1}$ would be needed. For the two left-preconditioned kernels, the techniques of Section 7 would be applied separately to both the matrices $M^{-1}A$ (for kernel (8.58)) and AM^{-1} (for kernel (8.59)). This is particularly easy when M^{-1} is (block) diagonal. Note that we may apply the techniques of Section 7 to the preconditioned case without needing to compute matrix products such as $M^{-1}A$ explicitly; we only need to know the sparsity pattern, which gives the communication pattern via the connections between subdomains.

Some sparse approximate inverse preconditioners have a structure which could work naturally with the preconditioned matrix powers kernel. For example, Chow's ParaSails preconditioner (Chow 2001) constrains the sparsity pattern to that of a small power A^k of the sparse matrix A to precondition.

The (unpreconditioned) matrix powers kernel computes the structure of powers of the matrix A anyway, more or less, so one could imagine combining the two structural preprocessing steps to save time. Of course an important part of Chow's preconditioner is the 'filtering' step, which removes small nonzeros from the structure of the preconditioner in order to save computation and bandwidth costs, so such a combined structural computation would require significant work in order to be useful. We propose this as a topic for future investigation.

In the special case of M diagonal (or block diagonal with sufficiently small blocks), there is no change to the communication of parallel CA-Akx at all. The same comments apply to sequential Akx and sequential CA-Akx, which are based on parallel CA-Akx.

8.6.3. Hierarchical semiseparable preconditioners

Another class of preconditioners approximates the inverse of a continuous integral operator, in order to approximate or precondition the discretization of that integral operator. Hierarchical matrices are a good example: see, for example, Börm, Grasedyck and Hackbusch (2004), Hackbusch (2006) and Börm and Grasedyck (2006). The exact inverse of the discrete operator A is generally dense, even if A itself is sparse. Furthermore, many integral operator discretizations produce a dense matrix A. However, that matrix has a structure which can be exploited, either to produce an approximate factorization or a preconditioner.

Again using tridiagonals as a motivating example, suppose M is a tridiagonal matrix. Even though tridiagonal systems may be solved in linear time, M^{-1} is dense in general, making our techniques presented so far inapplicable. However, the inverse of a tridiagonal has another important property: any submatrix strictly above or strictly below the diagonal has rank one. This off-diagonal low-rank property (hierarchical semiseparability) is shared by many good preconditioners, and we may exploit it to avoid communication too. A communication-avoiding algorithm for computing the preconditioned matrix powers kernel with hierarchical semiseparable $\tilde{A}$ is given in Knight *et al.* (2014).

8.6.4. Polynomial preconditioners

Previous authors developing s-step methods did not suggest how to avoid communication when preconditioning. Saad (1985), though not discussing s-step KSMs, did suggest applying something like a matrix powers kernel to polynomial preconditioning, when the matrix has the form of a stencil on a regular mesh. However, polynomial preconditioning tends to have a decreasing payoff as the degree of the polynomial increases, in terms of the number of CG iterations required for convergence (Saad 1985).

8.6.5. Incomplete factorizations

Recently, Grigori and Moufawad (2013) have derived and implemented a communication-avoiding ILU(0) preconditioner, CA-ILU(0), for both sequential and parallel architectures. To avoid communication, the authors propose a reordering algorithm for structured matrices that results in sparse L^{-1} and U^{-1} factors.

8.6.6. Deflation

Given a set of c deflation vectors which make up the columns of $n \times c$ matrix W, the Deflated CG method presented in Saad, Yeung, Erhel and Guyomarc'h (2000) transforms the linear system $Ax = b$ into the deflated system $H^T AH\tilde{x} = H^T b$, where $H = I - W(W^T AW)^{-1}(AW)^T$. When the columns of W are approximate eigenvectors of A associated with the c smallest eigenvalues, $\kappa(H^T AH) \approx \lambda_n/\lambda_{c+1}$, which in theory leads to an improved convergence rate.

This deflation technique can be extended to CA-CG such that the resulting algorithm still allows an $O(s)$ reduction in communication cost without a significant increase in computation costs. While this work is considered ongoing, preliminary numerical experiments confirm that deflation can increase the convergence rate of CA-CG, and additionally can make possible the use of higher s values (which can improve performance depending on the computing platform).

8.7. Conclusions and future work

In this section, we have derived and discussed communication-avoiding variants of four representative KSMs: Arnoldi and Lanczos for eigenvalue problems, and the corresponding GMRES and BICG for solving linear systems. We have also developed communication-avoiding versions of other KSMs, including communication-avoiding conjugate gradient squared (CA-CGS) and communication-avoiding biconjugate gradient stabilized (CA-BICGSTAB) (Carson *et al.* 2013). Three-term recurrence versions of both CA-Lanczos and CA-CG, as well as preconditioned variants of CA-GMRES, CA-Lanczos, and CA-CG, can be found in the thesis of Hoemmen (2010). For SPD A, communication-avoiding variants of symmetric Lanczos and CG can be easily obtained by simplifying (using $A = A^H$) the communication-avoiding nonsymmetric Lanczos and BICG methods in this section, respectively.

Speed-up results from a recent study of CA-KSM performance, presented in Section 8.4, demonstrate how the communication-avoiding approach can lead to significant speed-ups in practical applications. Ongoing work includes both theoretical and practical studies of the numerical stability and convergence rate in finite precision CA-KSMs, presented in Section 8.5, as well as the continued development of communication-avoiding preconditioners, discussed in Section 8.6.

While we have focused on algorithms rather than implementation details in this section, many nontrivial implementation decisions, discussed in Section 7, are required for the kernels used in CA-KSMs. Optimizations required for performance are both machine- and matrix-dependent, which makes autotuning and code generation an attractive approach. There are many ongoing efforts with this goal: see, for example, Byun, Lin, Yelick and Demmel (2012) and LaMielle and Strout (2010). By further analytical and empirical study of the convergence and performance properties of CA-KSMs, we hope to identify problems and machines for which CA-KSMs are competitive in solving practical problems, allowing the design of effective autotuners and integration of CA-KSM codes into existing frameworks.

9. Conclusion

We have considered many of the most commonly used algorithms of numerical linear algebra, including both direct and iterative methods, and asked the following three questions: Are there lower bounds on the amount of communication, that is, data movement, that these algorithms must perform? Do existing widely used algorithms attain these lower bounds? If not, are there other algorithms that do? To summarize the answers briefly, there are indeed communication lower bounds, current algorithms frequently fail to attain these bounds, and there are many new algorithms that do attain them, demonstrating large speed-ups in theory and practice. We pointed out many open questions that remain, in terms of refining the lower bounds, finding numerically stable algorithms that attain them, and producing high-performance implementations for the wide variety of existing and emerging computer architectures.

As mentioned in Section 2.6.7, it is in fact possible to extend these lower bounds and optimal algorithms to a much more general class of problems, namely algorithms that access arrays with subscripts that are linear functions of loop indices. The three nested loops of matrix multiplication and other linear algebra algorithms are canonical examples, but of course many other algorithms can be expressed in this way. To summarize, it is time to rebuild many of the standard linear algebra and other libraries on which computational science and engineering depend.

REFERENCES[12]

J. O. Aasen (1971), 'On the reduction of a symmetric matrix to tridiagonal form', *BIT Numer. Math.* **11**, 233–242.

N. N. Abdelmalek (1971), 'Round off error analysis for Gram–Schmidt method and solution of linear least squares problems', *BIT Numer. Math.* **11**, 345–367.

R. Agarwal, S. Balle, F. Gustavson, M. Joshi and P. Palkar (1995), 'A three-dimensional approach to parallel matrix multiplication', *IBM J. Res. Dev.* **39**, 575–582.

R. Agarwal, F. Gustavson and M. Zubair (1994), 'A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication', *IBM J. Res. Dev.* **38**, 673–681.

A. Aggarwal and J. Vitter (1988), 'The input/output complexity of sorting and related problems', *Comm. Assoc. Comput. Mach.* **31**, 1116–1127.

A. Aggarwal, A. K. Chandra and M. Snir (1990), 'Communication complexity of PRAMs', *Theoret. Comput. Sci.* **71**, 3–28.

N. Ahmed and K. Pingali (2000), Automatic generation of block-recursive codes. In *Proc. 6th International Euro-Par Conference on Parallel Processing*, Springer, pp. 368–378.

E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen (1992), *LAPACK Users' Guide*, SIAM.
Also available from http://www.netlib.org/lapack/.

M. Anderson, G. Ballard, J. Demmel and K. Keutzer (2011), Communication-avoiding QR decomposition for GPUs. In *Proc. 2011 IEEE International Parallel and Distributed Processing Symposium: IPDPS '11*, pp. 48–58.

W. E. Arnoldi (1951), 'The principle of minimized iterations in the solution of the matrix eigenvalue problem', *Quart. Appl. Math.* **9**, 17–29.

Z. Bai and D. Day (2000), Block Arnoldi method. In *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide* (Z. Bai, J. W. Demmel, J. J. Dongarra, A. Ruhe and H. van der Vorst, eds), SIAM, pp. 196–204.

Z. Bai, D. Day, J. Demmel and J. Dongarra (1997*a*), A test matrix collection for non-Hermitian eigenvalue problems. Technical report CS-97-355, Department of CS, University of Tennessee.

Z. Bai, J. Demmel and M. Gu (1997*b*), 'An inverse free parallel spectral divide and conquer algorithm for nonsymmetric eigenproblems', *Numer. Math.* **76**, 279–308.

Z. Bai, D. Hu and L. Reichel (1991), An implementation of the GMRES method using QR factorization. In *Proc. Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pp. 84–91.

Z. Bai, D. Hu and L. Reichel (1994), 'A Newton basis GMRES implementation', *IMA J. Numer. Anal.* **14**, 563–581.

G. Ballard (2013), Avoiding communication in dense linear algebra. PhD thesis, EECS Department, UC Berkeley.

[12] The URLs cited in this work were correct at the time of going to press, but the publisher and the authors make no undertaking that the citations remain live or are accurate or appropriate.

G. Ballard, D. Becker, J. Demmel, J. Dongarra, A. Druinsky, I. Peled, O. Schwartz, S. Toledo and I. Yamazaki (2013*a*), Communication-avoiding symmetric-indefinite factorization. Technical report UCB/EECS-2013-127, EECS Department, UC Berkeley.

G. Ballard, D. Becker, J. Demmel, J. Dongarra, A. Druinsky, I. Peled, O. Schwartz, S. Toledo and I. Yamazaki (2013*b*), Implementing a blocked Aasen's algorithm with a dynamic scheduler on multicore architectures. In *Proc. 27th IEEE International Parallel Distributed Processing Symposium: IPDPS '13*, pp. 895–907.

G. Ballard, A. Buluç, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz and S. Toledo (2013*c*), Communication optimal parallel multiplication of sparse random matrices. In *Proc. 25th ACM Symposium on Parallelism in Algorithms and Architectures: SPAA '13*, ACM, pp. 222–231.

G. Ballard, J. Demmel and I. Dumitriu (2011*a*), Communication-optimal parallel and sequential eigenvalue and singular value algorithms. Technical Report EECS-2011-14, UC Berkeley.

G. Ballard, J. Demmel and A. Gearhart (2011*b*), Brief announcement: Communication bounds for heterogeneous architectures. In *Proc. 23rd ACM Symposium on Parallelism in Algorithms and Architectures: SPAA '11*, ACM, pp. 257–258.

G. Ballard, J. Demmel and N. Knight (2012*a*), Communication avoiding successive band reduction. In *Proc. 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming: PPoPP '12*, ACM, pp. 35–44.

G. Ballard, J. Demmel and N. Knight (2013*d*), Avoiding communication in successive band reduction. Technical report UCB/EECS-2013-131, EECS Department, UC Berkeley.

G. Ballard, J. Demmel, L. Grigori, M. Jacquelin, H. D. Nguyen and E. Solomonik (2014), Reconstructing Householder vectors from Tall-Skinny QR. In *Proc. 2014 IEEE International Parallel and Distributed Processing Symposium: IPDPS '14*, to appear.

G. Ballard, J. Demmel, O. Holtz and O. Schwartz (2010), 'Communication-optimal parallel and sequential Cholesky decomposition', *SIAM J. Sci. Comput.* **32**, 3495–3523.

G. Ballard, J. Demmel, O. Holtz and O. Schwartz (2011*c*), Graph expansion and communication costs of fast matrix multiplication. In *Proc. 23rd ACM Symposium on Parallelism in Algorithms and Architectures: SPAA '11*, ACM, pp. 1–12.

G. Ballard, J. Demmel, O. Holtz and O. Schwartz (2011*d*), 'Minimizing communication in numerical linear algebra', *SIAM J. Matrix Anal. Appl.* **32**, 866–901.

G. Ballard, J. Demmel, O. Holtz and O. Schwartz (2012*b*), 'Graph expansion and communication costs of fast matrix multiplication', *J. Assoc. Comput. Mach.* **59**, #32.

G. Ballard, J. Demmel, O. Holtz and O. Schwartz (2012*c*), Sequential communication bounds for fast linear algebra. Technical report EECS-2012-36, UC Berkeley.

G. Ballard, J. Demmel, O. Holtz, B. Lipshitz and O. Schwartz (2012*d*), Brief announcement: Strong scaling of matrix multiplication algorithms and memory-

independent communication lower bounds. In *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures: SPAA '12*, ACM, pp. 77–79.

G. Ballard, J. Demmel, O. Holtz, B. Lipshitz and O. Schwartz (2012*e*), Communication-optimal parallel algorithm for Strassen's matrix multiplication. In *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures: SPAA '12*, ACM, pp. 193–204.

G. Ballard, J. Demmel, O. Holtz, B. Lipshitz and O. Schwartz (2012*f*), Graph expansion analysis for communication costs of fast rectangular matrix multiplication. In *Design and Analysis of Algorithms* (G. Even and D. Rawitz, eds), Vol. 7659 of *Lecture Notes in Computer Science*, Springer, pp. 13–36.

G. Ballard, J. Demmel, B. Lipshitz, O. Schwartz and S. Toledo (2013*f*), Communication efficient Gaussian elimination with partial pivoting using a shape morphing data layout. In *Proc. 25th ACM Symposium on Parallelism in Algorithms and Architectures: SPAA '13*, ACM, pp. 232–240.

R. Barrett, M. Berry, T. F. Chan, J. W. Demmel, J. Donato, J. J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. van der Vorst (1994), *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, second edition, SIAM.

M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob and E. Vicari (2010), 'Optimal sparse matrix dense vector multiplication in the I/O-model', *Theory Comput. Syst.* **47**, 934–962.

J. Bennett, A. Carbery, M. Christ and T. Tao (2010), 'Finite bounds for Hölder–Brascamp–Lieb multilinear inequalities', *Math. Res. Lett.* **17**, 647–666.

J. Berntsen (1989), 'Communication efficient matrix multiplication on hypercubes', *Parallel Comput.* **12**, 335–342.

G. Bilardi and F. P. Preparata (1999), 'Processor–time tradeoffs under bounded-speed message propagation II: Lower bounds', *Theory Comput. Syst.* **32**, 531–559.

G. Bilardi, A. Pietracaprina and P. D'Alberto (2000), On the space and access complexity of computation DAGs. In *Graph-Theoretic Concepts in Computer Science: 26th International Workshop* (U. Brandes and D. Wagner, eds), Vol. 1928 of *Lecture Notes in Computer Science*, Springer, pp. 47–58.

C. Bischof and C. Van Loan (1987), 'The WY representation for products of Householder matrices', *SIAM J. Sci. Statist. Comput.* **8**, 2–13.

C. H. Bischof, B. Lang and X. Sun (2000*a*), 'Algorithm 807: The SBR Toolbox, Software for successive band reduction', *ACM Trans. Math. Softw.* **26**, 602–616.

C. Bischof, B. Lang and X. Sun (2000*b*), 'A framework for symmetric band reduction', *ACM Trans. Math. Softw.* **26**, 581–601.

Å. Björck (1967), 'Solving linear least squares problems by Gram–Schmidt orthogonalization', *BIT Numer. Math.* **7**, 1–21.

L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker and R. C. Whaley (1997), *ScaLAPACK Users' Guide*, SIAM. Also available from http://www.netlib.org/scalapack/.

L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington and

R. C. Whaley (2002), 'An updated set of basic linear algebra subroutines (BLAS)', *J. ACM Trans. Math. Softw.* **28**, 135–151.

S. Börm and L. Grasedyck (2006), HLib package. http://www.hlib.org/hlib.html

S. Börm, L. Grasedyck and W. Hackbusch (2004), Hierarchical matrices. http://www.mis.mpg.de/scicomp/Fulltext/WS_HMatrices.pdf

K. Braman, R. Byers and R. Mathias (2002*a*), 'The multishift QR algorithm I: Maintaining well-focused shifts and level 3 performance', *SIAM J. Matrix Anal. Appl.* **23**, 929–947.

K. Braman, R. Byers and R. Mathias (2002*b*), 'The multishift QR algorithm II: Aggressive early deflation', *SIAM J. Matrix Anal. Appl.* **23**, 948–973.

J. Bunch and L. Kaufman (1977), 'Some stable methods for calculating inertia and solving symmetric linear systems', *Math. Comp.* **31**, 163–179.

A. Buttari, J. Langou, J. Kurzak and J. J. Dongarra (2007), A class of parallel tiled linear algebra algorithms for multicore architectures. LAPACK Working Note 191.

J. Byun, R. Lin, K. Yelick and J. Demmel (2012), Autotuning sparse matrix–vector multiplication for multicore. Technical report UCB/EECS-2012-215, EECS Department, UC Berkeley.

L. Cannon (1969), A cellular computer to implement the Kalman filter algorithm. PhD thesis, Montana State University.

E. Carson and J. Demmel (2014), 'A residual replacement strategy for improving the maximum attainable accuracy of s-step Krylov subspace methods', *SIAM J. Matrix Anal. Appl.* **35**, 22–43.

E. Carson, N. Knight and J. Demmel (2013), 'Avoiding communication in non-symmetric Lanczos-based Krylov subspace methods', *SIAM J. Sci. Comput.* **35**, S42–S61.

U. V. Catalyurek and C. Aykanat (1999), 'Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication', *IEEE Trans. Parallel Distributed Systems* **10**, 673–693.

Ü. V. Çatalyürek and C. Aykanat (2001), A fine-grain hypergraph model for 2D decomposition of sparse matrices. In *Proc. 15th IEEE International Parallel and Distributed Processing Symposium: IPDPS '01.*

E. Chan, M. Heimlich, A. Purkayastha and R. van de Geijn (2007), 'Collective communication: Theory, practice, and experience', *Concurrency and Computation: Pract. Exp.* **19**, 1749–1783.

E. Chow (2000), '*A priori* sparsity patterns for parallel sparse approximate inverse preconditioners', *SIAM J. Sci. Comput.* **21**, 1804–1822.

E. Chow (2001), 'Parallel implementation and practical use of sparse approximate inverses with *a priori* sparsity patterns', *Internat. J. High Perf. Comput. Appl.* **15**, 56–74.

M. Christ, J. Demmel, N. Knight, T. Scanlon and K. Yelick (2013), Communication lower bounds and optimal algorithms for programs that reference arrays, part 1. Technical report UCB/EECS-2013-61, EECS Department, UC Berkeley.

A. Chronopoulos and C. Gear (1989*a*), 'On the efficient implementation of preconditioned s-step conjugate gradient methods on multiprocessors with memory hierarchy', *Parallel Comput.* **11**, 37–53.

A. Chronopoulos and C. Gear (1989*b*), '*s*-step iterative methods for symmetric linear systems', *J. Comput. Appl. Math.* **25**, 153–168.

A. Chronopoulos and C. Swanson (1996), 'Parallel iterative *s*-step methods for unsymmetric linear systems', *Parallel Comput.* **22**, 623–641.

E. Cohen (1994), Estimating the size of the transitive closure in linear time, in *Proc. 35th Ann. Symp. Found. Comp. Sci.*, IEEE, pp. 190–200.

E. Cohen (1997), 'Size-estimation framework with applications to transitive closure and reachability', *J. Comput. System Sci.* **55**, 441–453.

Committee on the Analysis of Massive Data; Committee on Applied and Theoretical Statistics; Board on Mathematical Sciences and Their Applications; Division on Engineering and Physical Sciences; National Research Council (2013), *Frontiers in Massive Data Analysis*, The National Academies Press.

R. D. da Cunha, D. Becker and J. C. Patterson (2002), New parallel (rank-revealing) QR factorization algorithms. In *Euro-Par 2002 Parallel Processing*, Springer, pp. 677–686.

K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf and K. Yelick (2008), Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In *Proc. 2008 ACM/IEEE Conference on Supercomputing*, IEEE Press, p. 4.

T. Davis and Y. Hu (2011), 'The University of Florida sparse matrix collection', *ACM Trans. Math. Softw.* **38**, 1–25.

E. Dekel, D. Nassimi and S. Sahni (1981), 'Parallel matrix and graph algorithms', *SIAM J. Comput.* **10**, 657–675.

J. Demmel (1997), *Applied Numerical Linear Algebra*, SIAM.

J. Demmel (2013), An arithmetic complexity lower bound for computing rational functions, with applications to linear algebra. Technical report UCB/EECS-2013-126, EECS Department, UC Berkeley.

J. Demmel, I. Dumitriu and O. Holtz (2007*a*), 'Fast linear algebra is stable', *Numer. Math.* **108**, 59–91.

J. Demmel, I. Dumitriu, O. Holtz and R. Kleinberg (2007*b*), 'Fast matrix multiplication is stable', *Numer. Math.* **106**, 199–224.

J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz and O. Spillinger (2013*a*), Communication-optimal parallel recursive rectangular matrix multiplication. In *Proc. 27th IEEE International Parallel and Distributed Processing Symposium: IPDPS '13*, pp. 261–272.

J. Demmel, A. Gearhart, B. Lipshitz and O. Schwartz (2013*b*), Perfect strong scaling using no additional energy. In *Proc. 27th IEEE International Parallel and Distributed Processing Symposium: IPDPS '13*, pp. 649–660.

J. Demmel, L. Grigori, M. Gu and H. Xiang (2013*c*), Communication avoiding rank revealing QR factorization with column pivoting. Technical report UCB/EECS-2013-46, EECS Department, UC Berkeley.

J. Demmel, L. Grigori, M. Hoemmen and J. Langou (2008*a*), Communication-avoiding parallel and sequential QR and LU factorizations: Theory and practice. LAPACK Working Note.

J. Demmel, L. Grigori, M. Hoemmen and J. Langou (2012), 'Communication-optimal parallel and sequential QR and LU factorizations', *SIAM J. Sci. Comput.* **34**, A206–A239.

J. Demmel, M. Hoemmen, M. Mohiyuddin and K. Yelick (2007*c*), Avoiding communication in computing Krylov subspaces. Technical report UCB/EECS-2007-123, EECS Department, UC Berkeley.

J. Demmel, M. Hoemmen, M. Mohiyuddin and K. Yelick (2008*b*), Avoiding communication in sparse matrix computations. In *Proc. 2008 IEEE International Parallel and Distributed Processing Symposium: IPDPS 2008*, pp. 1–12.

J. Demmel, O. Marques, B. Parlett and C. Vömel (2008*c*), 'Performance and accuracy of LAPACK's symmetric tridiagonal eigensolvers', *SIAM J. Sci. Comput.* **30**, 1508–1526.

K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling and U. V. Catalyurek (2006), Parallel hypergraph partitioning for scientific computing. In *Proc. 20th IEEE International Parallel and Distributed Processing Symposium: IPDPS 2006*.

J. J. Dongarra, J. D. Croz, I. S. Duff and S. Hammarling (1990*a*), 'Algorithm 679: A set of level 3 basic linear algebra subprograms', *ACM Trans. Math. Softw.* **16**, 18–28.

J. J. Dongarra, J. D. Croz, I. S. Duff and S. Hammarling (1990*b*), 'A set of level 3 basic linear algebra subprograms', *ACM Trans. Math. Softw.* **16**, 1–17.

J. J. Dongarra, J. D. Croz, S. Hammarling and R. J. Hanson (1988*a*), 'Algorithm 656: An extended set of Fortran basic linear algebra subprograms', *ACM Trans. Math. Softw.* **14**, 18–32.

J. J. Dongarra, J. D. Croz, S. Hammarling and R. J. Hanson (1988*b*), 'An extended set of Fortran basic linear algebra subprograms', *ACM Trans. Math. Softw.* **14**, 1–17.

J. J. Dongarra, C. B. Moler, J. R. Bunch and G. W. Stewart (1979), *LINPACK Users' Guide*, SIAM.

C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde and C. Weiß (2000), 'Cache optimization for structured and unstructured grid multigrid', *Electron. Trans. Numer. Anal.* **10**, 21–40.

M. Driscoll, E. Georganas, P. Koanantakool, E. Solomonik and K. Yelick (2013), A communication-optimal N-body algorithm for direct interactions. In *Proc. 27th IEEE International Parallel and Distributed Processing Symposium: IPDPS '13*, pp. 1075–1084.

H. Dursun, K.-I. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano and P. Vashishta (2009), A multilevel parallelization framework for high-order stencil computations. In *Euro-Par 2009 Parallel Processing*, Springer, pp. 642–653.

E. Elmroth and F. Gustavson (1998), New serial and parallel recursive QR factorization algorithms for SMP systems. In *Applied Parallel Computing: Large Scale Scientific and Industrial Problems* (B. Kågström *et al.*, eds), Vol. 1541 of *Lecture Notes in Computer Science*, Springer, pp. 120–128.

R. Floyd (1962), 'Algorithm 97: Shortest path', *Commun. Assoc. Comput. Mach.* **5**, 345.

J. D. Frens and D. S. Wise (2003), 'QR factorization with Morton-ordered quadtree matrices for memory re-use and parallelism', *ACM SIGPLAN Notices* **38**, 144–154.

M. Frigo and V. Strumpen (2005), Cache oblivious stencil computations. In *Proc. 19th Annual International Conference on Supercomputing*, ACM, pp. 361–366.

M. Frigo and V. Strumpen (2009), 'The cache complexity of multithreaded cache oblivious algorithms', *Theory Comput. Syst.* **45**, 203–233.

M. Frigo, C. E. Leiserson, H. Prokop and S. Ramachandran (1999), Cache-oblivious algorithms. In *Proc. 40th Annual Symposium on Foundations of Computer Science: FOCS '99*, IEEE Computer Society, pp. 285–297.

S. H. Fuller and L. I. Millett, eds (2011), *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press. http://www.nap.edu

D. Gannon and J. Van Rosendale (1984), 'On the impact of communication complexity on the design of parallel numerical algorithms', *Trans. Comput.* **100**, 1180–1194.

E. Georganas, J. Gonzalez-Dominguez, E. Solomonik, Y. Zheng, J. Tourino and K. Yelick (2012), Communication avoiding and overlapping for numerical linear algebra. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis: SC '12*, pp. 1–11.

A. George (1973), 'Nested dissection of a regular finite element mesh', *SIAM J. Numer. Anal.* **10**, 345–363.

J. R. Gilbert and R. E. Tarjan (1987), 'The analysis of a nested dissection algorithm', *Numer. Math.* pp. 377–404.

L. Giraud and J. Langou (2003), 'A robust criterion for the modified Gram–Schmidt algorithm with selective reorthogonalization', *SIAM J. Sci. Comput.* **25**, 417–441.

L. Giraud, J. Langou and M. Rozloznik (2005), 'The loss of orthogonality in the Gram–Schmidt orthogonalization process', *Comput. Math. Appl.* **50**, 1069–1075.

G. Golub and C. Van Loan (1996), *Matrix Computations*, third edition, Johns Hopkins University Press.

G. H. Golub, R. J. Plemmons and A. Sameh (1988), Parallel block schemes for large-scale least-squares computations. In *High-Speed Computing: Scientific Applications and Algorithm Design*, University of Illinois Press, pp. 171–179.

S. L. Graham, M. Snir and C. A. Patterson, eds (2004), *Getting up to Speed: The Future of Supercomputing*, Report of National Research Council of the National Academies Sciences, The National Academies Press.

R. Granat, B. Kågström, D. Kressner and M. Shao (2012), Parallel library software for the multishift QR algorithm with aggressive early deflation. Report UMINF 12.06, Department of Computing Science, Umeå University, SE-901.

A. Greenbaum (1997*a*), 'Estimating the attainable accuracy of recursively computed residual methods', *SIAM J. Matrix Anal. Appl.* **18**, 535–551.

A. Greenbaum (1997*b*), *Iterative Methods for Solving Linear Systems*, SIAM.

A. Greenbaum, M. Rozložník and Z. Strakoš (1997), 'Numerical behavior of the modified Gram–Schmidt GMRES implementation', *BIT Numer. Math.* **37**, 706–719.

G. Greiner and R. Jacob (2010*a*), Evaluating non-square sparse bilinear forms on multiple vector pairs in the I/O-model. In *Mathematical Foundations of Computer Science 2010*, Springer, pp. 393–404.

G. Greiner and R. Jacob (2010*b*), The I/O complexity of sparse matrix dense matrix multiplication. In *LATIN 2010: Theoretical Informatics*, Springer, pp. 143–156.

L. Grigori and S. Moufawad (2013), Communication avoiding ILU(0) preconditioner. Research report RR-8266, INRIA.

L. Grigori, P.-Y. David, J. Demmel and S. Peyronnet (2010), Brief announcement: Lower bounds on communication for sparse Cholesky factorization of a model problem. In *Proc. 22nd ACM Symposium on Parallelism in Algorithms and Architectures: SPAA '10*, ACM, pp. 79–81.

L. Grigori, J. Demmel and H. Xiang (2011), 'CALU: A communication optimal LU factorization algorithm', *SIAM J. Matrix Anal. Appl.* **32**, 1317–1350.

M. Gu and S. Eisenstat (1996), 'Efficient algorithms for computing a strong rank-revealing QR factorization', *SIAM J. Sci. Comput.* **17**, 848–869.

B. C. Gunter and R. A. van de Geijn (2005), 'Parallel out-of-core computation and updating of the QR factorization', *ACM Trans. Math. Softw.* **31**, 60–78.

F. G. Gustavson (1997), 'Recursion leads to automatic variable blocking for dense linear-algebra algorithms', *IBM J. Res. Dev.* **41**, 737–756.

M. Gutknecht (1997), Lanczos-type solvers for nonsymmetric linear systems of equations. In *Acta Numerica*, Vol. 6, Cambridge University Press, pp. 271–398.

M. Gutknecht and K. Ressel (2000), 'Look-ahead procedures for Lanczos-type product methods based on three-term Lanczos recurrences', *SIAM J. Matrix Anal. Appl.* **21**, 1051–1078.

M. Gutknecht and Z. Strakoš (2000), 'Accuracy of two three-term and three two-term recurrences for Krylov space solvers', *SIAM J. Matrix Anal. Appl.* **22**, 213–229.

W. Hackbusch (2006), Hierarchische Matrizen: Algorithmen und Analysis. http://www.mis.mpg.de/scicomp/Fulltext/hmvorlesung.ps

A. Haidar, P. Luszczek, J. Kurzak and J. Dongarra (2013), An improved parallel singular value algorithm and its implementation for multicore hardware. LAPACK Working Note 283.

M. R. Hestenes and E. Stiefel (1952), 'Methods of conjugate gradients for solving linear systems', *J. Res. Nat. Bur. Standards* **49**, 409–436.

A. Hindmarsh and H. Walker (1986), Note on a Householder implementation of the GMRES method. Technical report UCID-20899, Lawrence Livermore National Laboratory.

M. Hoemmen (2010), Communication-avoiding Krylov subspace methods. PhD thesis, EECS Department, UC Berkeley.

A. J. Hoffman, M. S. Martin and D. J. Rose (1973), 'Complexity bounds for regular finite difference and finite element grids', *SIAM J. Numer. Anal.* **10**, 364–369.

J. W. Hong and H. T. Kung (1981), I/O complexity: The red–blue pebble game. In *Proc. 13th Annual ACM Symposium on Theory of Computing: STOC '81*, ACM, pp. 326–333.

G. W. Howell, J. Demmel, C. T. Fulton, S. Hammarling and K. Marmol (2008), 'Cache efficient bidiagonalization using BLAS 2.5 operators', *ACM Trans. Math. Softw.* **34**, #14.

P. Hupp and R. Jacob (2013), Tight bounds for low dimensional star stencils in the external memory model. In *Algorithms and Data Structures* (F. Dehne, R. Solis-Oba and J.-R. Sack, eds), Vol. 8037 of *Lecture Notes in Computer Science*, Springer, pp. 415–426.

F. Irigoin and R. Triolet (1988), Supernode partitioning. In *Proc. 15th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, ACM, pp. 319–329.

D. Irony, S. Toledo and A. Tiskin (2004), 'Communication lower bounds for distributed-memory matrix multiplication', *J. Parallel Distrib. Comput.* **64**, 1017–1026.

W. Jalby and B. Philippe (1991), 'Stability analysis and improvement of the block Gram–Schmidt algorithm', *SIAM J. Sci. Statist. Comput.* **12**, 1058–1073.

S. L. Johnsson (1992), 'Minimizing the communication time for matrix multiplication on multiprocessors', *Parallel Comput.*

W. Joubert and G. Carey (1992), 'Parallelizable restarted iterative methods for nonsymmetric linear systems I: Theory', *Internat. J. Comput. Math.* **44**, 243–267.

B. Kågström, D. Kressner and M. Shao (2012), On aggressive early deflation in parallel variants of the QR algorithm. In *Applied Parallel and Scientific Computing*, Springer, pp. 1–10.

L. Karlsson and B. Kågström (2011), 'Parallel two-stage reduction to Hessenberg form using dynamic scheduling on shared-memory architectures', *Parallel Comput.* **37**, 771–782.

J. Kepner and J. Gilbert (2011), *Graph Algorithms in the Language of Linear Algebra*, Vol. 22, SIAM.

A. Kiełbasinski (1974), 'Numerical analysis of the Gram–Schmidt orthogonalization algorithm (analiza numeryczna algorytmu ortogonalizacji Grama–Schmidta)', *Roczniki Polskiego Towarzystwa Matematycznego, Seria III: Matematyka Stosowana II*, pp. 15–35.

S. Kim and A. Chronopoulos (1992), 'An efficient nonsymmetric Lanczos method on parallel vector computers', *J. Comput. Appl. Math.* **42**, 357–374.

N. Knight, E. Carson and J. Demmel (2014), Exploiting data sparsity in parallel matrix powers computations. In *Proc. PPAM '13*, Vol. 8384 of *Lecture Notes in Computer Science*, Springer, to appear.

T. G. Kolda and B. W. Bader (2009), 'Tensor decompositions and applications', *SIAM Review* **51**, 455–500.

A. LaMielle and M. Strout (2010), Enabling code generation within the sparse polyhedral framework. Technical report CS-10-102, Colorado State University.

C. Lanczos (1950), *An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators*, United States Government Press Office.

C. L. Lawson, R. J. Hanson, D. Kincaid and F. T. Krogh (1979), 'Basic linear algebra subprograms for Fortran usage', *ACM Trans. Math. Softw.* **5**, 308–323.

C. E. Leiserson, S. Rao and S. Toledo (1997), 'Efficient out-of-core algorithms for linear relaxation using blocking covers', *J. Comput. System Sci.* **54**, 332–344.

G. Lev and L. Valiant (1983), 'Size bounds for superconcentrators', *Theor. Comput. Sci.* **22**, 233–251.

B. Lipshitz (2013), Communication-avoiding parallel recursive algorithms for matrix multiplication. Master's thesis, EECS Department, UC Berkeley.

B. Lipshitz, G. Ballard, J. Demmel and O. Schwartz (2012), Communication-avoiding parallel Strassen: Implementation and performance. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis: SC '12*, #101.

L. H. Loomis and H. Whitney (1949), 'An inequality related to the isoperimetric inequality', *Bull. Amer. Math. Soc.* **55**, 961–962.

W. McColl and A. Tiskin (1999), 'Memory-efficient matrix multiplication in the BSP model', *Algorithmica* **24**, 287–297.

G. Meurant (2006), *The Lanczos and Conjugate Gradient Algorithms: From Theory to Finite Precision Computations*, SIAM.

G. Meurant and Z. Strakoš (2006), The Lanczos and conjugate gradient algorithms in finite precision arithmetic. In *Acta Numerica*, Vol. 15, Cambridge University Press, pp. 471–542.

S. Mohanty and S. Gopalan (2012), I/O efficient QR and QZ algorithms, in *2012 19th International Conference on High Performance Computing: HiPC*, pp. 1–9.

M. Mohiyuddin (2012), Tuning hardware and software for multiprocessors. PhD thesis, EECS Department, UC Berkeley.

M. Mohiyuddin, M. Hoemmen, J. Demmel and K. Yelick (2009), Minimizing communication in sparse matrix solvers. In *Proc. International Conference on High Performance Computing Networking, Storage and Analysis: SC '09*.

Y. Nakatsukasa and N. Higham (2012), Stable and efficient spectral divide and conquer algorithms for the symmetric eigenvalue decomposition and the SVD. MIMS EPrint 2012.52, University of Manchester.

R. Nishtala, R. W. Vuduc, J. W. Demmel and K. A. Yelick (2007), 'When cache blocking of sparse matrix vector multiply works and why', *Applicable Algebra in Engineering, Communication and Computing* **18**, 297–311.

J.-S. Park, M. Penner and V. Prasanna (2004), 'Optimizing graph algorithms for improved cache performance', *IEEE Trans. Parallel Distributed Systems* **15**, 769–782.

B. Parlett (1995), The new QD algorithms. In *Acta Numerica*, Vol. 4, Cambridge University Press, pp. 459–491.

B. Parlett and J. Reid (1970), 'On the solution of a system of linear equations whose matrix is symmetric but not definite', *BIT Numer. Math.* **10**, 386–397.

B. Parlett, D. Taylor and Z. Liu (1985), 'A look-ahead Lanczos algorithm for unsymmetric matrices', *Math. Comp.* **44**, 105–124.

C. Pfeifer (1963), Data flow and storage allocation for the PDQ-5 program on the Philco-2000. Technical report, Westinghouse Electric Corp. Bettis Atomic Power Lab., Pittsburgh.
B. Philippe and L. Reichel (2012), 'On the generation of Krylov subspace bases', *Appl. Numer. Math.* **62**, 1171–1186.
C. Puglisi (1992), 'Modification of the Householder method based on compact WY representation', *SIAM J. Sci. Statist. Comput.* **13**, 723–726.
L. Reichel (1990), 'Newton interpolation at Leja points', *BIT* **30**, 332–346.
M. Rozložník, G. Shklarski and S. Toledo (2011), 'Partitioned triangular tridiagonalization', *ACM Trans. Math. Softw.* **37**, #38.
Y. Saad (1985), 'Practical use of polynomial preconditionings for the conjugate gradient method', *SIAM J. Sci. Statist. Comput.* **6**, 865–881.
Y. Saad (2003), *Iterative Methods for Sparse Linear Systems*, second edition, SIAM.
Y. Saad and M. H. Schultz (1986), 'GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems', *SIAM J. Sci. Statist. Comput.* **7**, 856–869.
Y. Saad, M. Yeung, J. Erhel and F. Guyomarc'h (2000), 'A deflated version of the conjugate gradient algorithm', *SIAM J. Sci. Comput.* **21**, 1909–1926.
J. E. Savage (1995), Extending the Hong–Kung model to memory hierarchies. In *Computing and Combinatorics*, Vol. 959, Springer, pp. 270–281.
M. Schatz, J. Poulson and R. van de Geijn (2013), Scalable universal matrix multiplication algorithms: 2D and 3D variations on a theme. Technical report, University of Texas.
R. Schreiber and C. Van Loan (1989), 'A storage-efficient WY representation for products of Householder transformations', *SIAM J. Sci. Statist. Comput.* **10**, 53–57.
H. A. Schwarz (1870), 'Über einen Grenzübergang durch alternierendes Verfahren', *Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich* **15**, 272–286.
M. Scquizzato and F. Silvestri (2014), Communication lower bounds for distributed-memory computations. In *31st International Symposium on Theoretical Aspects of Computer Science: STACS 2014* (E. W. Mayr and N. Portier, eds), Vol. 25 of *Leibniz International Proceedings in Informatics: LIPIcs*, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 627–638.
G. Sleijpen and H. van der Vorst (1996), 'Reliable updated residuals in hybrid Bi-CG methods', *Computing* **56**, 141–163.
B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema and C. B. Moler (1976), *Matrix Eigensystem Routines: EISPACK Guide*, second edition, Springer.
A. Smoktunowicz, J. L. Barlow and J. Langou (2006), 'A note on the error analysis of classical Gram–Schmidt', *Numer. Math.* **105**, 299–313.
E. Solomonik and J. Demmel (2011), Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In *Euro-Par 2011 Parallel Processing* (E. Jeannot, R. Namyst and J. Roman, eds), Vol. 6853 of *Lecture Notes in Computer Science*, Springer, pp. 90–109.
E. Solomonik, A. Buluç and J. Demmel (2013), Minimizing communication in all-pairs shortest-paths. In *Proc. 27th IEEE International Parallel Distributed Processing Symposium: IPDPS '13*, pp. 548–559.

E. Solomonik, E. Carson, N. Knight and J. Demmel (2014), Tradeoffs between synchronization, communication, and work in parallel linear algebra computations. Technical report UCB/EECS-2014-8, EECS Department, UC Berkeley.

E. Solomonik, D. Matthews, J. Hammond and J. Demmel (2013*c*), Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *Proc. 27th IEEE International Parallel and Distributed Processing Symposium: IPDPS '13*, pp. 813–824.

D. Sorensen (1985), 'Analysis of pairwise pivoting in Gaussian elimination', *IEEE Trans. Computers* **C-34**, 274–278.

D. Sorensen (1992), 'Implicit application of polynomial filters in a k-step Arnoldi method', *SIAM J. Matrix Anal. Appl.* **13**, 357–385.

A. Stathopoulos and K. Wu (2002), 'A block orthogonalization procedure with constant synchronization requirements', *SIAM J. Sci. Comput.* **23**, 2165–2182.

G. Stewart (2008), 'Block Gram–Schmidt orthogonalization', *SIAM J. Sci. Comput.* **31**, 761–775.

V. Strassen (1969), 'Gaussian elimination is not optimal', *Numer. Math.* **13**, 354–356.

M. M. Strout, L. Carter and J. Ferrante (2001), Rescheduling for locality in sparse matrix computations. In *Computational Science: ICCS 2001*, Springer, pp. 137–146.

E. Sturler (1996), 'A performance model for Krylov subspace methods on mesh-based parallel computers', *Parallel Comput.* **22**, 57–74.

Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk and C. E. Leiserson (2011), The Pochoir stencil compiler. In *Proc. 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, ACM, pp. 117–128.

R. Thakur, R. Rabenseifner and W. Gropp (2005), 'Optimization of collective communication operations in MPICH', *Internat. J. High Performance Comput. Appl.* **19**, 49–66.

A. Tiskin (2002), 'Bulk-synchronous parallel Gaussian elimination', *J. Math. Sci.* **108**, 977–991.

A. Tiskin (2007), 'Communication-efficient parallel generic pairwise elimination', *Future Generation Computer Systems* **23**, 179–188.

S. Toledo (1995), Quantitative performance modeling of scientific computations and creating locality in numerical algorithms. PhD thesis, MIT.

S. Toledo (1997), 'Locality of reference in LU decomposition with partial pivoting', *SIAM J. Matrix Anal. Appl.* **18**, 1065–1081.

C. Tong and Q. Ye (2000), 'Analysis of the finite precision bi-conjugate gradient algorithm for nonsymmetric linear systems', *Math. Comp.* **69**, 1559–1576.

L. Trefethen and R. Schreiber (1990), 'Average-case stability of Gaussian elimination', *SIAM J. Matrix Anal. Appl.* **11**, 335–360.

R. A. van de Geijn and J. Watts (1997), 'SUMMA: Scalable universal matrix multiplication algorithm', *Concurrency: Pract. Exp.* **9**, 255–274.

H. Van der Vorst and Q. Ye (1999), 'Residual replacement strategies for Krylov subspace iterative methods for the convergence of true residuals', *SIAM J. Sci. Comput.* **22**, 835–852.

J. Van Rosendale (1983), Minimizing inner product data dependencies in conjugate gradient iteration. Technical report 172178, ICASE-NASA.

D. Vanderstraeten (1999), A stable and efficient parallel block Gram–Schmidt algorithm. In *Euro-Par99 Parallel Processing*, Springer, pp. 1128–1135.

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

R. W. Vuduc (2003), Automatic performance tuning of sparse matrix kernels. PhD thesis, EECS Department, UC Berkeley.

H. Walker (1988), 'Implementation of the GMRES method using Householder transformations', *SIAM J. Sci. Statist. Comput.* **9**, 152–163.

S. Warshall (1962), 'A theorem on Boolean matrices', *J. Assoc. Comput. Mach.* **9**, 11–12.

S. Williams, M. Lijewski, A. Almgren, B. Van Straalen, E. Carson, N. Knight and J. Demmel (2014), s-step Krylov subspace methods as bottom solvers for geometric multigrid. In *Proc. 2014 IEEE International Parallel and Distributed Processing Symposium: IPDPS '14*, to appear.

S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick and J. Demmel (2009), 'Optimization of sparse matrix–vector multiplication on emerging multicore platforms', *Parallel Comput.* **35**, 178–194.

V. Williams (2012), Multiplying matrices faster than Coppersmith–Winograd. In *Proc. 44th Annual Symposium on Theory of Computing: STOC '12*, ACM, pp. 887–898.

D. Wise (2000), Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free. In *Euro-Par 2000 Parallel Processing* (A. Bode, T. Ludwig, W. Karl and R. Wismöller, eds), Vol. 1900 of *Lecture Notes in Computer Science*, Springer, pp. 774–783.

M. M. Wolf, E. G. Boman and B. Hendrickson (2008), 'Optimizing parallel sparse matrix–vector multiplication by corner partitioning', *PARA08, Trondheim, Norway.*

J. Xia, S. Chandrasekaran, M. Gu and X. S. Li (2010), 'Fast algorithms for hierarchically semiseparable matrices', *Numer. Linear Algebra Appl.* **17**, 953–976.

K. Yotov, T. Roeder, K. Pingali, J. Gunnels and F. Gustavson (2007), An experimental comparison of cache-oblivious and cache-conscious programs. In *Proc. 19th Annual ACM Symposium on Parallel Algorithms and Architectures: SPAA '07*, ACM, pp. 93–104.

A. Yzelman and R. H. Bisseling (2011), 'Two-dimensional cache-oblivious sparse matrix–vector multiplication', *Parallel Comput.* **37**, 806–819.