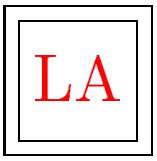

Numerical Linear Algebra

1 Introduction

We begin with an example to motivate our study of parallel and vector algorithms for linear algebra problems. Consider solving a system of linear equations $Ax = b$, where A is a given dense n-by-n matrix, b is a given n-by-1 vector, and x is an n-by-1 vector of unknowns we want to compute. The algorithm we will use is *Cholesky*, which is a variation of Gaussian elimination suitable for matrices A with the special (but common) properties of *symmetry* and *positive definiteness* (these are discussed in more detail below). Cholesky, as well as Gaussian elimination, consists almost entirely of simple, regular operations which are in principle easy to parallelize or vectorize: adding multiples of one column of the matrix to other columns. Since vector processors like the Cray Y-MP are designed to perform this operation particularly well, we expect to get good performance running Cholesky on a Cray Y-MP.

The second column of Table 1 gives the speed on a single processor of a Cray Y-MP; here vectorization is feature to be exploited. The third column gives the speed using all 8 processors; here parallelization as well as vectorization is possible. The first row of the table gives the maximum possible speed in megaflops of the machine; no algorithm for any problem can go faster. The next three rows give the speed of various Basic Linear Algebra Subroutines (or BLAS), such as multiplying two large, square (dimension $n = 500$) matrices, multiplying a large square matrix by a vector, and solving a triangular system of equations $Tx = b$ by substitution. These BLAS have been written in Cray assembly language and fully exploit the Cray Y-MP architecture; we see that they generally come quite close to the maximum performance given in the first row.

The fifth row of the table gives the performance of the Cholesky subroutine `spofa` from the LINPACK subroutine library [1]. LINPACK is a well-known and widely used library of

Table 1: Performance in Megaflops of Cholesky on a Cray Y-MP

	1 Proc.	8 Procs.
Maximum speed	330	2640
Matrix-matrix multiply	312	2425
Matrix-vector multiply	311	2285
Solve $Tx = b$	272	584
LINPACK (Cholesky, $n = 500$)	72	72
LAPACK (Cholesky, $n = 500$)	290	1414
LAPACK (Cholesky, $n = 1000$)	301	2115

Fortran 77 linear algebra routines, but was written before the advent of vector and parallel computers. Its performance is very poor, over 4.5 times slower than the maximum machine speed on a single processor and over 36 times slower than maximum on 8 processors. This is very disappointing, since at first glance it is hard to imagine an algorithm better suited to the Cray than Cholesky.

The last two rows of Table 1 give the performance of a different version of Cholesky, `sposv`, taken from the newer LAPACK library of linear algebra routines [2]. LAPACK was written to exploit architectures like the Cray. It is also written in Fortran 77, but includes calls to the BLAS, which are written in Cray assembler. It performs the same floating point operations as LINPACK, yet goes 4 times faster on a single processor and almost 20 times faster on 8 processors. Increasing the matrix dimension n from 500 to 1000 increases LAPACK's speed even more, to within 80% of the peak machine speed.

The purpose of this example is to show that an "obviously" parallel or vector algorithm may work much more poorly than expected, but by applying certain (systematic) transformations to the algorithm, it can be made to work quite well. We understand these algorithmic transformations most completely in the case of simple algorithms like Cholesky, on simple (to the user) architectures like the Cray Y-MP. We understand them rather less well for more complicated algorithms on more complicated architectures. Our goal in this chapter is to discuss these transformations so the reader can learn to use them, and give pointers to existing software (like LAPACK) where these transformations have already been performed.

In order to describe and solve the basic linear algebra problems of this chapter, we need some notation. We will refer frequently to *matrices*, *vectors* and *scalars*. A matrix will be denoted by an upper case letter like A, and its (i, j)-th element by a_{ij}. Occasionally in detailed algorithmic descriptions we will instead write $A(i, j)$. The submatrix of A occupying rows i though j and columns k through l will be denoted $A(i : j, k : l)$. A lower case letter like x will denote a vector, and its i-th element will be written x_i. Vectors will almost always be column vectors, which are the same as matrices with one column. Lower case Greek letters (and occasionally lower case letters) will denote scalars. $\mathbb{R}$ will denote the set of real numbers, $\mathbb{R}^n$ the set of n-dimensional real vectors, and $\mathbb{R}^{m\times n}$ the set of m-by-n real matrices.

A^T will denote the *transpose* of the matrix A: $(A^T)_{ij} = a_{ji}$. A matrix is called *symmetric* if $A = A^T$. A matrix A is *positive definite* if $x^T Ax > 0$ for all nonzero vectors x. Other notation will be introduced as needed.

Computational science problems can often be reduced to solving one or a sequence of the following **standard linear algebra problems**:

- **Solving linear systems of equations**: $Ax = b$. Here A is a given n-by-n nonsingular real or complex matrix, b is a given column vector with n entries, and x is a column vector with n entries we wish to compute.

- **Least squares problems**: compute the x which minimizes $\|Ax - b\|_2$ where A is m-by-n, b is m-by-1, and x is n-by-1. $\|y\|_2 \equiv \sqrt{\sum |y_i|^2}$ is called the *2-norm* of the vector y. If $m > n$, so we have more equations than unknowns, the system is called *overdetermined*. In this case we cannot generally solve $Ax = b$ exactly. If $m < n$, the system is called *underdetermined*, and we will have infinitely many solutions.

- **Eigenvalue problems**: Given an n-by-n matrix A, find an n-by-1 nonzero vector x and a scalar λ so that $Ax = \lambda x$.

For example, linear equations often arise when solving ordinary or partial differential equations numerically; the linear system must be solved to advance the solution by a time step. Least squares problems arise in fitting curves or surfaces to experimental data. Eigenvalue problems arise when analyzing vibrations. There are also many important variations on these basic problems,[1] but to keep this chapter to a reasonable length, we will concentrate on algorithms for solving systems of linear equations, and refer elsewhere for least squares and eigenvalue problems [3, 4, 5, 6].

A great many algorithms are available for all these problems. The choice of algorithm depends on three things:

- The **structure of** A has a large influence on the algorithm. For example, solving $Ax = b$ using Gaussian elimination takes $2/3 \cdot n^3 + O(n^2)$ flops (floating point operations) if A is a dense matrix. If A is *symmetric* and *positive definite*, we can use the Cholesky algorithm which costs half as much: $1/3 \cdot n^3 + O(n^2)$ flops. If A is *triangular* (i.e. either zero above the diagonal or zero below the diagonal), then we can solve $Ax = b$ in only $n^2 + O(n)$ flops by simple substitution, which is much faster. Recognizing triangularity is easy, but other structures are more subtle. For example, if A arises from solving certain elliptic partial differential equations (like Poisson's equation), then $Ax = b$ can actually be solved in $O(n)$ flops using *multigrid*, which is essentially as little work as possible.

[1] For example, there is the as the *generalized eigenvalue problem* $Ax = \lambda Bx$, where A and B are given square matrices and we want to find the eigenvalue λ and eigenvector x, and the *singular value decomposition*, which can be described as finding the eigenvalues and eigenvectors of $A^T A$ and AA^T.

- The **desired accuracy** limits which algorithms are suitable. For example, if one needs to solve $Ax = b$ to 16 decimals of accuracy (more on what this means below), one might use a careful and expensive implementation of Gaussian elimination, but if one only needs 3 decimal digits, a few iterations of a much cheaper iterative method like *conjugate gradients* might suffice.

- The **computer system** on which one wants to solve the problem influences the algorithm choice greatly, and is the main topic of this chapter. Differences in the computer architecture, available programming languages, and compiler quality can make large differences in the way one implements a given algorithm; we saw this in Table 1 above, and will see it later as well. The computer system can influence algorithm performance so much that one would choose very different algorithms on different machines. New parallel architectures have also lead to the invention of new and interesting parallel algorithms.

Since the three standard problems listed above have been studied for so long, one might hope that good numerical software libraries would be available to solve them. This is partly true, as we illustrated by LAPACK [2, 7]. However, there are several reasons one cannot depend on libraries entirely. First, it is impossible to anticipate all possible structures of A that can arise in practice and write corresponding libraries (although extensive libraries for various specialized A do exist [7]). Second, high performance computer architectures, programming languages and compilers have been changing so much lately that library writers have not been able to keep up. There is no clear cut rule on when to use a library routine. In some cases an unoptimized or serial implementation of an algorithm which fully exploits a matrix structure may be much faster than a highly tuned, parallel implementation of a more general algorithm, and in other cases the opposite may be true. If the user's priority is ease of programming and reliability, a library routine is a good choice. If, instead, the user's priority is maximum performance on one specialized problem, and software development time is secondary, custom software is the likely choice. We will try to address the concerns of both classes of users in this chapter.

The rest of this chapter is organized as follows. Section 2 discusses the cost models we will use to understand algorithm performance. Section 3 discusses the BLAS, which we use to construct many high performance algorithms. Section 4 discusses how to implement matrix multiplication quickly on several different parallel architectures; these techniques serve as models for implementing more complicated algorithms. Section 5 discusses how to layout matrices on distributed memory machines. Section 6 discusses parallelizing Gaussian elimination on dense matrices, and gives pointers to other related algorithms. Section 7 discusses parallelizing Gaussian elimination on dense matrices, and gives pointers to other related algorithms. Section 8 discusses parallelizing Gaussian elimination on sparse matrices; this is a much harder problem, and much less software is available. Section 9 discusses iterative methods for solving $Ax = b$, which in contrast to Gaussian elimination take an approximate solution and iteratively improves it, rather than providing the final answer after a fixed number of steps. Section 10 shows how to bound the error in a computed

solution to $Ax = b$, for any of the methods discussed in this chapter. Section 11 discusses how to use netlib to retrieve existing library software electronically. Finally, section 12 gives a quick references guide to the BLAS.

This is a necessarily partial survey of a large and active field. For further reading in numerical linear algebra, see the textbooks by Golub and Van Loan [4], Watkins [5] or Demmel [6]. For more details on parallel numerical linear algebra in particular, see [3, 8, 9], the last of which includes a bibliography of over 2000 references.

2 Memory Hierarchies

To understand the speedups of LAPACK over LINPACK in Table 1, and in general to be able to understand which algorithms are fast and which are slow, we need to understand memory hierarchies. Memory hierarchies were discussed in some detail in the chapter on Computer Architecture, and we just review them briefly here, emphasizing their importance to algorithm design.

Computer memories are built as *hierarchies*, with a series of different kinds of memories ranging from very fast, expensive, and therefore small memory at the top of the hierarchy, down to slow, cheap and very large memory at the bottom. For example, registers typically form the fastest memory, then cache, main memory, disks, and finally tape as the slowest, largest and cheapest. Useful floating point operations can only be done on data at the top of the hierarchy, in the registers. Since an entire large matrix cannot fit in the registers, it must be moved up and down through the hierarchy, transferred up to the registers when work needs to be done, and transferred back down to the main memory (or disk or tape) when it is no longer needed. It takes time to move between levels in the memory hierarchy, and moving is slower the farther down in the hierarchy one goes. Indeed, one such data movement takes far longer than performing a floating point operation. So the limiting factor for many algorithms is *not* the time needed for floating point operations, but the time to move data in the memory hierarchy. Therefore, a clever algorithm will attempt to minimize the number of these memory moves (even if it might mean doing a few more floating point operations).

LINPACK's Cholesky performed so poorly in Table 1, because it was *not* designed to minimize memory movement on machines such as the Cray YMP (it was designed to minimize another kind of memory movement, *page faults* between main memory and disk). In contrast, the matrix-matrix multiplication and other BLAS on the Cray YMP in Table 1 were written (in assembly language) just for the Cray YMP to minimize data movement.

Since it is expensive and time-consuming to write every routine like Cholesky in assembly language for every new computer, we would like a better approach. Here is the most successful approach we have discovered so far. Since operations like matrix-matrix multiplication are so common, computer manufacturers have standardized them as the *Basic Linear Algebra Subroutines* or *BLAS* [10, 11, 12], and optimized them for their machines. In other words, a library of subroutines for matrix-matrix multiplication, matrix-vector multiplication, and other routines is available as a standard Fortran (or C) callable library on most

high performance machines, and underneath they have been optimized for each machine. If we can reorganize our algorithms to use these optimized BLAS to perform all or most of the work, then our algorithms will go as fast as the manufacturer's optimized BLAS. This was the approach taken in LAPACK: the algorithms in LINPACK (and the corresponding library for eigenvalue problems, EISPACK [13, 14]) were reorganized to call the BLAS in their innermost loops, where most of the work is done. This led to the speedups shown in Table 1.

This approach was very successful on machines like the Cray, i.e. parallel vector processors using fast shared memory with relatively few processors. On newer architectures, especially distributed memory machines like the Intel Paragon and CM-5, this approach cannot be used as straightforwardly, although many of the same ideas still apply. The difficulty with distributed memory machines is twofold. First, the memory hierarchy is deeper, including both "local memory" and "remote memory" layers at the bottom (remote memory means memory physically on another processor). Individual nodes may also be more complicated; for example, the CM-5 has 5 discernible levels of memory hierarchy on a single processor node; other machines are also complicated. Second, languages and compilers are still evolving, so that there are many more possible ways to store a matrix on a machine, and "obviously" parallelizable or vectorizable loops may or may not be compiled well.

In the remaining sections, we will discuss the successful BLAS-based approach to numerical software, and outline current activities on distributed memory machines.

3 The BLAS

Let us examine the BLAS more carefully. For a more complete list of the BLAS, see section 12. Table 2 counts the number of memory references and floating points operations performed by three related BLAS. The last column gives the ratio q of flops to memory references. The significance of q is that it tells us roughly how many flops we can perform per memory reference, or how much useful work we can do compared to the time moving data; therefore, the algorithms with the larger q values are better building blocks for other algorithms.

Table 2 reflects a hierarchy of operations: Operations like `saxpy` operate on vectors and offer the worst q values; these are called Level 1 BLAS [10], and include inner products and other simple operations. Operations like matrix–vector multiplication operate on matrices and vectors, and offer slightly better q values; these are called Level 2 BLAS [11], and include solving triangular systems of equations and rank-1 updates of matrices ($A + xy^T$, x and y column vectors). Operations like matrix–matrix multiplication operate on pairs of matrices, and offer the best q values; these are called Level 3 BLAS [12], and include solving triangular systems of equations with many right hand sides.

Since the Level 3 BLAS have the highest q values, we endeavor to reorganize our algorithms in terms of operations like matrix-matrix multiplication, rather than saxpy (the LINPACK Cholesky is already constructed in terms of calls to saxpy).

Table 2: Basic Linear Algebra Subroutines (BLAS)

Operation	Definition	Floating point operations	Memory references	q
`saxpy`	$y_i = \alpha x_i + y_i$, $i = 1, ..., n$	$2n$	$3n+1$	$2/3$
Matrix–vector mult	$y_i = \sum_{j=1}^{n} A_{ij} x_j + y_i$	$2n^2$	$n^2 + 3n$	2
Matrix–matrix mult	$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} + C_{ij}$	$2n^3$	$4n^2$	$n/2$

4 Matrix Multiplication

4.1 Matrix Multiplication on a Shared Memory Machine

Let us examine in detail how to implement matrix multiplication to minimize the number of memory moves. Suppose we have two levels of memory hierarchy, fast and slow, where the slow memory is large enough to contain the $n \times n$ matrices A, B and C, but the fast memory contains only M words where $n < M \ll n^2$. Further assume the data are reused optimally (which may be optimistic if the decisions are made automatically by hardware).

The simplest algorithm one might try consists of three nested loops:

Algorithm 4.1 *Unblocked Matrix Multiplication*

for $i = 1 : n$
  for $j = 1 : n$
    for $k = 1 : n$
      $C_{ij} = C_{ij} + A_{ik} \cdot B_{kj}$

We count the number of references to the slow memory as follows: n^3 for reading B n times, n^2 for reading A one row at a time and keeping it in fast memory until it is no longer needed, and $2n^2$ for reading one entry of C at a time, keeping it in fast memory until it is completely computed. This comes to $n^3 + 3n^2$ for a q of about 2, which is no better than the Level 2 BLAS and far from the maximum possible $n/2$. If $M \ll n$, so that we cannot keep a full row of A in fast memory, q further decreases to 1, since the algorithm reduces to a sequence of inner products, which are Level 1 BLAS. For every permutation of the three loops on i, j and k, one gets another algorithm with q about the same.

The next possibility is dividing B and C into *column blocks*, and computing C block by block. Recall that we use the notation $B(i : j, k : l)$ to mean the submatrix in rows i through j and columns k through l. We partition $B = [B^{(1)}, B^{(2)}, ..., B^{(N)}]$ where each $B^{(i)}$ is $n \times n/N$, and similarly for C. Our column block algorithm is then

Algorithm 4.2 *Column-blocked Matrix Multiplication*

for $j = 1 : N$
  for $k = 1 : n$
    $C^{(j)} = C^{(j)} + A(1 : n, k) \cdot B^{(j)}(k, 1 : n/N)$

Assuming $M \geq 2n^2/N + n$, so that fast memory can accommodate $B^{(j)}$, $C^{(j)}$ and one column of A simultaneously, our memory reference count is as follows: $2n^2$ for reading and writing each block of C once, n^2 for reading each block of B once, and Nn^2 for reading A N times. This yields $q \approx M/n$, so that M needs to grow with n to keep q large.

Finally, we consider *rectangular blocking*, where A is broken into an $N \times N$ block matrix with $n/N \times n/N$ blocks $A^{(ij)}$, and B and C are similarly partitioned. The algorithm becomes

Algorithm 4.3 *Rectangular-blocked Matrix Multiplication*

for $i = 1 : N$
  for $j = 1 : N$
    for $k = 1, N$
      $C^{(ij)} = C^{(ij)} + A^{(ik)} \cdot B^{(kj)}$

Assuming $M \geq 3(n/N)^2$ so that one block each from A, B and C fit in memory simultaneously, our memory reference count is as follows: $2n^2$ for reading and writing each block of C once, Nn^2 for reading A N times, and Nn^2 for reading B N times. This yields $q \approx \sqrt{M/3}$, which is much better than the previous algorithms. In [15] an analysis of this problem leads to an upper bound for q near $\sqrt{M}$, so we cannot expect to improve much on this algorithm for square matrices.

See exercises 1 and 2.

4.2 Matrix Multiplication on a Distributed Memory Machine

In this section it will be convenient to number matrix entries (or subblocks) and processors from 0 to $n - 1$ instead of 1 to n.

On distributed memory machines the cost model is more complicated than on shared memory machines, because we will need to worry about the *data layout*, or how the matrices are partitioned across the machine. This will determine both the amount of parallelism and the cost of communication. Recall from the chapter on Computer Architecture that the cost of sending a message of k words from one processor to another is $\alpha + k\beta$, where α is the *start-up cost* or *latency*, and β is the per-word cost, or reciprocal of *bandwidth*. Therefore to assess the cost of an algorithm we need to count the number of floating point

operations, the number of messages sent (at a cost of α per message), and the total length of messages sent (at a cost of β per word). We begin by showing how best to implement matrix multiplication without regard to the layout's suitability for other matrix operations, and return to the question of layouts in the next section.

The algorithm is due to Cannon [20] and is well suited for computers laid out in a square $N \times N$ mesh, i.e. where each processor communicates most efficiently with the four other processors immediately north, east, south and west of itself. We also assume the processors at the edges of the grid are directly connected to the processors on the opposite edge; this makes the topology that of a two-dimensional torus. Let A be partitioned into square subblocks $A^{(ij)}$ as before, with $A^{(ij)}$ stored on processor (i, j). Let B and C be partitioned similarly. The algorithm is given below. It is easily seen that whenever $A^{(ik)}$ and $B^{(kj)}$ 'meet' in processor i, j, they are multiplied and accumulated in $C^{(ij)}$; the products for the different $C^{(ij)}$ are accumulated in different orders.

Algorithm 4.4 *Cannon's matrix multiplication algorithm*

forall $i = 0 : N-1$
  Left circular shift row i *by* i*, so that* $A^{(i,j)}$
    is assigned $A^{(i,(j+i) \bmod N)}$.
forall $j = 0 : N-1$
  Upward circular shift column j *by* j*, so that* $B^{(i,j)}$
    is assigned $B^{((j+i) \bmod N, j)}$.
for $k = 1 : N$
  forall $i = 0 : N-1$, *forall* $j = 0 : N-1$
    $C^{(ij)} = C^{(ij)} + A^{(ij)} \cdot B^{(ij)}$
    Left circular shift each row of A *by 1, so* $A^{(i,j)}$
      is assigned $A^{(i,(j+1) \bmod N)}$.
    Upward circular shift each column of B *by 1, so* $B^{(i,j)}$
      is assigned $B^{((i+1) \bmod N, j)}$.

Figure 1 illustrates the functioning of this algorithm for $N = 3$.

The time spent by this algorithm is computed as follows. Assume we multiply n-by-n matrices on an N-by-N processor mesh, where for simplicity N divides n evenly. Assuming messages are sent between nearest neighbors only, and that a processor can only send a single message at a time, the total number of messages sent (in parallel) is $4N - 2$, the total number of words sent (in parallel) is $(n/N)^2 \cdot (4N - 2)$, and the total number of flops (in parallel) is $2n^3/N^2$.

See exercises 3, 4, 5, and 6.

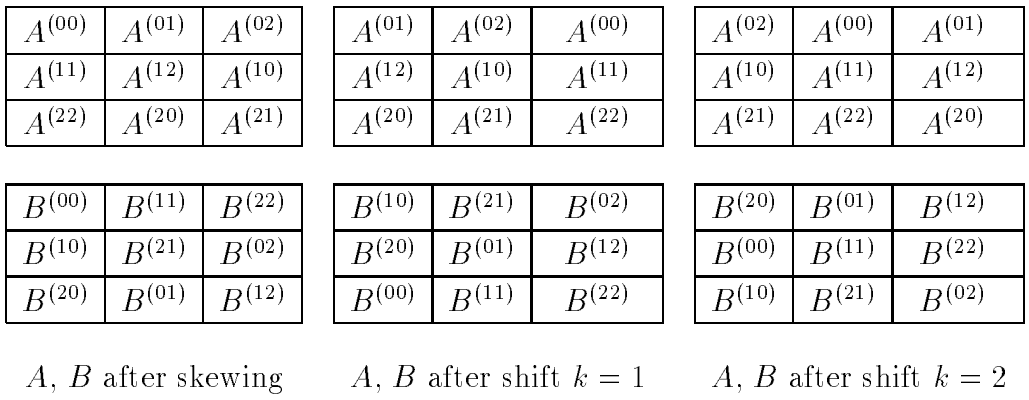

Figure 1: Cannon's algorithm for $N = 3$.

A variation of this algorithm suitable for machines that are efficient at *spreading* subblocks across rows (or down columns) is to do this instead of the preshifting and rotation of A (or B) [21].

Cannon's algorithm may also be easily adapted to a hypercube [3]. The simplest way is to embed a grid (or two-dimensional torus) in a hypercube, i.e. map the processors in a grid to the processors in a hypercube, and the connections in a grid to a subset of the connections in a hypercube [22, 23]. This approach (which is useful for more than matrix multiplication) uses only a subset of the connections in a hypercube, which makes the communication slower than it need be. Several sophisticated improvements on this basic algorithm have been developed [24, 25], the latter of which fully utilizes the available bandwidth of the hypercube to reduce the number of messages sent by a factor of 2 and the number of words sent by a factor of nearly $2n/N$. This assumes each processor in the hypercube can send messages to all its neighbors simultaneously, as was the case on the CM-2. If the architecture permits us to overlap communication and computation, we can save up to another factor of two in speed.

In this section we have shown one can optimize matrix multiplication in a series of steps tuning it ever more highly for a particular computer architecture, until essentially every communication link and floating point unit is utilized. The algorithms are scalable, in that they continue to run efficiently on larger machines and larger problems, with communication costs becoming ever smaller with respect to computation. On the other hand, let us ask what we lose by optimizing so heavily for one architecture. Our high performance depends on the matrices having just the right dimensions, being laid out just right in memory, and leaving them in a scrambled final position at the end (although a modest amount of extra communication could repair this). It is unreasonable to expect users, who want to do several computations of which this is but one, to satisfy all these requirements. Therefore a practical algorithm will have to deal with many irregularities, and be quite complicated. Our ability to do this extreme optimization is limited to a few simple and regular problems like matrix

multiplication on a hypercube, as well as other heavily used kernels like the BLAS, which have indeed been highly optimized for many architectures. We do not expect equal success for more complicated algorithms on all architectures of interest, at least within a reasonable amount of time.[1] Also, the algorithm is highly tuned to a particular interconnection network topology, which may require redesign for another machine. In view of this, a number of recent machines try to make communication time appear as independent of topology as possible, so the user sees essentially a completely connected topology.

5 Data Layouts on Distributed Memory Machines

Choosing a data layout may be described as choosing a mapping $f(i,j)$ from location (i,j) in a matrix to the processor on which it is stored. As discussed previously, we hope to design f so that it permits highly parallel implementation of a variety of matrix algorithms, limits communication cost as much as possible, and retains these attractive properties as we scale to larger matrices and larger machines. For example, the algorithm of the previous section uses the map $f(i,j) = (\lfloor i/r \rfloor, \lfloor j/r \rfloor)$, where we subscript matrices starting at 0, number processors by their coordinates in a grid (also starting at (0,0)), and store an $r \times r$ submatrix on each processor, where $r = n/N$.

There is an emerging consensus about data layouts for distributed memory machines. This is being implemented in several programming languages [26, 27], that will be available to programmers in the near future. We describe these layouts here.

High Performance Fortran (HPF) [27] permits the user to define a virtual array of processors, align actual data structures like matrices and arrays with this virtual array (and so with respect to each other), and then to layout the virtual processor array on an actual machine. We describe the layout functions f offered for this last step. The range of f is a rectangular array of processors numbered from $(0,0)$ up to $(p_1 - 1, p_2 - 1)$. Then all f can be parameterized by two integer parameters b_1 and b_2 as follows:

$$f_{b_1,b_2}(i,j) = \left(\left\lfloor \frac{i}{b_1} \right\rfloor \bmod p_1, \left\lfloor \frac{j}{b_2} \right\rfloor \bmod p_2 \right)$$

Suppose the matrix A (or virtual processor array) is $m \times n$. Then choosing $b_2 = n$ yields a column of processors, each containing some number of complete rows of A. Choosing $b_1 = m$ yields a row of processors. Choosing $b_1 = m/p_1$ and $b_2 = n/p_2$ yields a *blocked layout*, where A is broken into $b_1 \times b_2$ subblocks, each of which resides on a single processor. This is the simplest two-dimensional layout one could imagine (we used it in the previous section), and by having large subblocks stored on each processor it makes using the BLAS on each processor attractive. However, for straightforward matrix algorithms that process the matrix from left to right (including Gaussian elimination, QR decomposition, reduction to tridiagonal form, and so on), the leftmost processors will become idle early in the computation and make load balance poor. Choosing $b_1 = b_2 = 1$ is called *scatter mapping* (also known as *wrapped* or *cyclic* or *interleaved mapping*), and optimizes load balance, since the matrix entries stored on a single processor are as nearly as possible uniformly distributed throughout the matrix.

0,0	0,0	0,0	0,0	0,1	0,1	0,1	0,1	0,2	0,2	0,2	0,2	0,3	0,3	0,3	0,3
0,0	0,0	0,0	0,0	0,1	0,1	0,1	0,1	0,2	0,2	0,2	0,2	0,3	0,3	0,3	0,3
0,0	0,0	0,0	0,0	0,1	0,1	0,1	0,1	0,2	0,2	0,2	0,2	0,3	0,3	0,3	0,3
0,0	0,0	0,0	0,0	0,1	0,1	0,1	0,1	0,2	0,2	0,2	0,2	0,3	0,3	0,3	0,3
1,0	1,0	1,0	1,0	1,1	1,1	1,1	1,1	1,2	1,2	1,2	1,2	1,3	1,3	1,3	1,3
1,0	1,0	1,0	1,0	1,1	1,1	1,1	1,1	1,2	1,2	1,2	1,2	1,3	1,3	1,3	1,3
1,0	1,0	1,0	1,0	1,1	1,1	1,1	1,1	1,2	1,2	1,2	1,2	1,3	1,3	1,3	1,3
1,0	1,0	1,0	1,0	1,1	1,1	1,1	1,1	1,2	1,2	1,2	1,2	1,3	1,3	1,3	1,3
2,0	2,0	2,0	2,0	2,1	2,1	2,1	2,1	2,2	2,2	2,2	2,2	2,3	2,3	2,3	2,3
2,0	2,0	2,0	2,0	2,1	2,1	2,1	2,1	2,2	2,2	2,2	2,2	2,3	2,3	2,3	2,3
2,0	2,0	2,0	2,0	2,1	2,1	2,1	2,1	2,2	2,2	2,2	2,2	2,3	2,3	2,3	2,3
2,0	2,0	2,0	2,0	2,1	2,1	2,1	2,1	2,2	2,2	2,2	2,2	2,3	2,3	2,3	2,3
3,0	3,0	3,0	3,0	3,1	3,1	3,1	3,1	3,2	3,2	3,2	3,2	3,3	3,3	3,3	3,3
3,0	3,0	3,0	3,0	3,1	3,1	3,1	3,1	3,2	3,2	3,2	3,2	3,3	3,3	3,3	3,3
3,0	3,0	3,0	3,0	3,1	3,1	3,1	3,1	3,2	3,2	3,2	3,2	3,3	3,3	3,3	3,3
3,0	3,0	3,0	3,0	3,1	3,1	3,1	3,1	3,2	3,2	3,2	3,2	3,3	3,3	3,3	3,3

Figure 2: Block layout of a 16×16 matrix on a 4×4 processor grid.

On the other hand, this appears to inhibit the use of the BLAS locally in each processor, since the data owned by a processor are not contiguous from the point of view of the matrix. Finally, by choosing $1 < b_1 < m/p_1$ and $1 < b_2 < n/p_2$, we get a *block-scatter mapping* which trades off load balance and applicability of the BLAS. These layouts are shown in Figures 2 through 4 for a 16×16 matrix laid out on a 4×4 processor grid; each array entry is labeled by the number of the processor that stores it.

See exercises 7 and 8.

A different approach is to write algorithms that work independently of the location of the data, and rely on the underlying language or run-time system to optimize the necessary communications. This makes code easier to write, but puts a large burden on compiler and run-time system writers [28]. Even though compiler and run-time system writers aspire to make programming this easy, they have not yet succeeded (at least without large performance penalties), so the computational scientist interested in top performance must currently pay attention to details of data layout and communication.

6 Gaussian Elimination on Dense Matrices

We begin with a brief review of Gaussian elimination, Then we show how to reorganize it for shared memory parallel machines (section 6.1) and distributed memory parallel machines (section 6.2).

To solve $Ax = b$, we first use Gaussian elimination to factor the nonsingular matrix A as

0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3

Figure 3: Scatter layout of a 16×16 matrix on a 4×4 processor grid.

0,0	0,0	0,1	0,1	0,2	0,2	0,3	0,3	0,0	0,0	0,1	0,1	0,2	0,2	0,3	0,3
0,0	0,0	0,1	0,1	0,2	0,2	0,3	0,3	0,0	0,0	0,1	0,1	0,2	0,2	0,3	0,3
1,0	1,0	1,1	1,1	1,2	1,2	1,3	1,3	1,0	1,0	1,1	1,1	1,2	1,2	1,3	1,3
1,0	1,0	1,1	1,1	1,2	1,2	1,3	1,3	1,0	1,0	1,1	1,1	1,2	1,2	1,3	1,3
2,0	2,0	2,1	2,1	2,2	2,2	2,3	2,3	2,0	2,0	2,1	2,1	2,2	2,2	2,3	2,3
2,0	2,0	2,1	2,1	2,2	2,2	2,3	2,3	2,0	2,0	2,1	2,1	2,2	2,2	2,3	2,3
3,0	3,0	3,1	3,1	3,2	3,2	3,3	3,3	3,0	3,0	3,1	3,1	3,2	3,2	3,3	3,3
3,0	3,0	3,1	3,1	3,2	3,2	3,3	3,3	3,0	3,0	3,1	3,1	3,2	3,2	3,3	3,3
0,0	0,0	0,1	0,1	0,2	0,2	0,3	0,3	0,0	0,0	0,1	0,1	0,2	0,2	0,3	0,3
0,0	0,0	0,1	0,1	0,2	0,2	0,3	0,3	0,0	0,0	0,1	0,1	0,2	0,2	0,3	0,3
1,0	1,0	1,1	1,1	1,2	1,2	1,3	1,3	1,0	1,0	1,1	1,1	1,2	1,2	1,3	1,3
1,0	1,0	1,1	1,1	1,2	1,2	1,3	1,3	1,0	1,0	1,1	1,1	1,2	1,2	1,3	1,3
2,0	2,0	2,1	2,1	2,2	2,2	2,3	2,3	2,0	2,0	2,1	2,1	2,2	2,2	2,3	2,3
2,0	2,0	2,1	2,1	2,2	2,2	2,3	2,3	2,0	2,0	2,1	2,1	2,2	2,2	2,3	2,3
3,0	3,0	3,1	3,1	3,2	3,2	3,3	3,3	3,0	3,0	3,1	3,1	3,2	3,2	3,3	3,3
3,0	3,0	3,1	3,1	3,2	3,2	3,3	3,3	3,0	3,0	3,1	3,1	3,2	3,2	3,3	3,3

Figure 4: Block scatter layout of a 16×16 matrix on a 4×4 processor grid with 2×2 blocks.

$PA = LU$, where L is lower triangular, U is upper triangular, and P is a permutation matrix, i.e. PA is the same as A but with its rows in a permuted order. Then we solve the triangular systems $Ly = Pb$ and $Ux = y$ for the solution x. These last two operations are already Level 2 BLAS, so we concentrate on factoring $PA = LU$, which has the dominant number of floating point operations, $2n^3/3 + \mathcal{O}(n^2)$. Reordering the rows of A with P is called *pivoting*, and is necessary for numerical stability (more on this in section 10). We use the standard *partial pivoting* scheme [4]: this means L has ones on its diagonal and other entries bounded in absolute value by one. The simplest version of Gaussian elimination algorithm involves adding multiples of one row of A to others to zero out subdiagonal entries, and overwriting A with L and U:

Algorithm 6.1 *Row oriented Gaussian elimination (kij-LU decomposition)*

for $k = 1 : n-1$
  { *Pivot by choosing* l *so* $|A(l,k)| = \max_{k \le i \le n} |A(i,k)|$,
    and swapping rows l *and* k *of* A }
  { *Exit if* $A(k,k) = 0$ }
  for $i = k+1 : n$
    $A(i,k) = A(i,k)/A(k,k)$
  for $i = k+1 : n$
    for $j = k+1 : n$
      $A(i,j) = A(i,j) - A(i,k) \cdot A(k,j)$

When this algorithm terminates, the diagonal and upper triangle of A contain U, the part below the diagonal contains the corresponding part of L (the diagonal of L is all ones, and so does not need to be stored. The permutation matrix P is determined implicitly by the permutations in the second line of the algorithm.

If A is stored by column, as is the case in Fortran (but not C), then since the inner loop combines rows of A, it accesses memory locations (at least) n locations apart. As described in Chapter CA, this does not respect locality, and so is likely to be slower than an algorithm which accesses A by column (i.e. accesses consecutive memory locations). Algorithm 6.1 is also called kij-LU decomposition, because of the nesting order of its loops. All the rest of the 3! permutations of i, j and k lead to valid algorithms, some of which access columns of A in the innermost loop. The next algorithm is one of these, and is used in the LINPACK routine `sgefa` [1].

Algorithm 6.2 *Column oriented Gaussian elimination (kji-LU decomposition)*

for $k = 1 : n-1$
  { *Pivot by choosing* l *so* $|A(l,k)| = \max_{k \le i \le n} |A(i,k)|$,
    and swapping $A(l,k)$ *and* $A(k,k)$ }
  { *Exit if* $A(k,k) = 0$ }
  for $i = k+1 : n$

$A(i,k) = A(i,k)/A(k,k)$
for $j = k+1 : n$
{ *Pivot by swapping* $A(l,j)$ *and* $A(k,j)$ }
for $i = k+1 : n$
$A(i,j) = A(i,j) - A(i,k) \cdot A(k,j)$

6.1 Gaussian Elimination on Shared Memory Machines

The innermost loop of Algorithm 6.2 can be performed by a single call to the Level 1 BLAS operation `saxpy`; this in done in LINPACK. To achieve higher performance, we modify this code first to use the Level 2 and then the Level 3 BLAS in its innermost loops. Again, 3! versions of these algorithms are possible, but we just describe the ones used in the LAPACK library [2]. There is obvious parallelism in the innermost loop, since each A_{ij} can be updated independently. To make the use of BLAS clear, we use Fortran 90 (or Matlab) notation:

Algorithm 6.3 *Gaussian elimination using Level 2 BLAS*

for $k = 1 : n-1$
{ *Pivot by choosing* l *so* $|A_{lk}| = \max_{k \le i \le n} |A_{ik}|$,
and swapping rows l *and* k *of* A }
{ *Exit if* $A(k,k) = 0$ }
$A(k+1:n, k) = A(k+1:n, k)/A_{kk}$
$A(k+1:n, k+1:n) = A(k+1:n, k+1:n)$
$-A(k+1:n, k) \cdot A(k, k+1:n)$

The parallelism is evident: most work is performed is a single rank-1 update of the trailing $n-k \times n-k$ submatrix $A(k+1:n, k+1:n)$, where each entry of $A(k+1:n, k+1:n)$ can be updated in parallel. Other permutations of the nested loops lead to different algorithms, which depend on the BLAS for matrix–vector multiplication and solving a triangular system instead of rank-1 updating [29, 30]; which is faster depends on the relative speed of these on a given machine.

To convert to the Level 3 BLAS involves column blocking

$$A = [A^{(1)}, \ldots, A^{(m)}]$$

into $n \times n_b$ blocks, where n_b is the *block size* and $m = n/n_b$. The optimal choice of n_b depends on the memory hierarchy of the machine in question: our approach is to compute the LU decomposition of each $n \times n_b$ subblock of A using Algorithm 6.3 in the fast memory, and then use Level 3 BLAS to update the rest of the matrix:

Algorithm 6.4 *Gaussian elimination using Level 3 BLAS (we assume* n_b *divides* n*)*

for $l = 1 : m$

$k = (l-1) \cdot n_b + 1$
Use Algorithm 6.3 to factorize $PA^{(l)} = LU$ *in place*
Apply P *to prior columns* $A(1:n, 1:k-1)$ *and later columns*
$A(1:n, k+n_b:n)$
Update block row of U*:*
Replace $A(k:k+n_b-1, k+n_b:n)$ *by the solution* X *of*
$TX = A(k:k+n_b-1, k+n_b:n)$*, where* T *is the lower*
triangular matrix in $A(k:k+n_b-1, k:k+n_b-1)$
$A(k+n_b:n, k+n_b:n) = A(k+n_b:n, k+n_b:n) -$
$A(k+n_b:n, k:k+n_b-1) \cdot A(k:k+n_b-1, k+n_b:n)$

Most of the work is performed in the last two lines, solving a triangular system with many right-hand sides, and matrix multiplication. Other similar algorithms may be derived by conformally partitioning L, U and A, and equating partitions in $A = LU$. Algorithms 6.3 and 6.4 are available as, respectively, subroutines `sgetf2` and `sgetrf` in LAPACK [2].

See exercise 9.

The LAPACK routine implementing Cholesky is `sposv`. Other matrix structures handled by LAPACK include

- *packed symmetric positive definite matrices*, i.e. stored in half the space of a full dense matrix (`sppsv`)
- *symmetric indefinite matrices*, i.e. symmetric but not necessarily positive definite matrices (`ssysv`),
- *packed symmetric indefinite matrices*, i.e. stored in half the space of a full dense matrix (`sspsv`),
- *band matrices*, i.e. those nonzero only within a fixed distance from the main diagonal (`sgbsv`),
- *tridiagonal matrices*, i.e. a special case of band matrices which are nonzero only on the diagonal, immediately above it and immediately below it (`sgtsv`),
- *symmetric positive definite band matrices* (`spbsv`), and
- *symmetric positive definite tridiagonal matrices* (`sptsv`).

There are also double precision, complex, and double precision complex counterparts of all LAPACK routines (replace the leading 's' in the subroutine name by 'd', 'c', and 'z', respectively).

6.2 Gaussian Elimination on Distributed Memory Machines

As described earlier, data layout influences the algorithm. We show the algorithm for a block scatter mapping in both dimensions, and then discuss how other layouts may be handled. The algorithm is essentially the same as Algorithm 6.4 with interprocessor communication inserted as necessary. The block size n_b equals b_2, which determines the layout in the horizontal direction.

Communication is required in Algorithm 6.3 to find the pivot entry at each step and swap rows if necessary; then each processor can perform the scaling and rank-1 updates independently. The pivot search is a *reduction* operation, meaning that values from all processors must be reduced to a single value, a pointer to the row containing the largest pivot. After the block column is fully factorized, the pivot information must be *broadcast* so other processors can permute their own data, as well as permute among different processors.

In Algorithm 6.4, the $n_b \times n_b$ L matrix stored on the diagonal must be *spread* rightward to other processors in the same row, so they can compute their entries of U. Finally, the processors holding the rest of L below the diagonal must *spread* their submatrices to the right, and the processors holding the new entries of U just computed must *spread* their submatrices downward, before the final rank-n_b update in the last line of Algorithm 6.4 can take place.

The optimal choice of block sizes b_1 and b_2 depends on the cost of communication *versus* computation. For example, if the communication required to do pivot search and swapping of rows is expensive, b_1 should be large. The execution time is a function of dimension n, block sizes b_1 and b_2, processor counts p_1 and p_2, and the cost of computation and communication (from Chapter CA, we know how to model these). Given this function, it may be minimized as a function of b_1, b_2, p_1 and p_2. Some theoretical analyses of this sort for special cases may be found in [30] and the references therein. See also [31] and [32]. As an example of the performance that can be attained in practice, on an Intel Delta with 512 processors the speed of LU ranged from a little over 1 gigaflop for $n = 2000$ to nearly 12 gigaflops for $n = 25000$.

Even if the layout is not block scatter as described so far, essentially the same algorithm may be used. As described in Section 5, many possible layouts are related by permutation matrices. So simply performing the algorithm just described with (optimal) block sizes b_1 and b_2 on the matrix A as stored is equivalent to performing the LU decomposition of $P_1 A P_2$ where P_1 and P_2 are permutation matrices. Thus at the cost of keeping track of these permutations (a possibly nontrivial software issue), a single algorithm suffices for a wide variety of layouts.

Finally, we need to solve the triangular systems $Ly = b$ and $Ux = y$ arising from the LU decomposition. On a shared memory machine, this is accomplished by two calls to the Level 2 BLAS. Designing such an algorithm on a distributed memory machine is harder, because the fewer floating point operations performed ($\mathcal{O}(n^2)$ instead of $\mathcal{O}(n^3)$) make it harder to mask the communication [33, 34, 35, 36].

7 Gaussian Elimination on Band Matrices

Recall that a *band matrix* is one which is nonzero only on certain diagonals. We say a matrix A has *lower bandwidth* lb if $a_{ij} = 0$ for $i > j + lb$, and *upper bandwidth* ub if $a_{ij} = 0$ for $i < j - ub$. If lb and/or ub are large, then the techniques of the Section 6 are still applicable, and the LAPACK routines for band matrices (`sgbsv` and `spbsv`) have been optimized for this situation [37, 21]. Different techniques are needed when the bandwidth is small. The reason is that proportionately less and less parallel work is available in updating the trailing submatrix, which is where we use the Level 3 BLAS. In the limiting case of tridiagonal matrices, the parallel algorithm derived as in section 6.1 and the standard serial algorithm are nearly identical.

The problem of solving banded linear systems with a narrow band has been studied by many authors, see for instance the references in [9, 38]. We will only sketch some of the main ideas and we will do so for rather simple problems. The reader should keep in mind that these ideas can easily be generalized for more complicated situations, and many have appeared in the literature.

Most of the parallel approaches perform more arithmetic operations than standard (sequential) Gaussian elimination (typically 2.5 times as many), *twisted factorization* being the only exception. In twisted factorization the Gaussian elimination process is carried out in parallel from both sides. This method was first proposed in [39] for tridiagonal systems $Tx = b$ as a means to compute a specified component of x more accurately. For a tridiagonal matrix twisted factorization leads to the following decomposition of T:

$$\begin{pmatrix} \star & \star & & & & & & \\ \star & \star & \star & & & & & \\ & \star & \star & \star & & & & \\ & & \star & \ddots & \ddots & & & \\ & & & \ddots & \ddots & \ddots & & \\ & & & & \ddots & \star & \star & \\ & & & & & \star & \star & \star \\ & & & & & & \star & \star \end{pmatrix} \tag{1}$$

$$= \begin{pmatrix} \star & & & & & & & \\ \star & \star & & & & & & \\ & \star & \ddots & & & & & \\ & & \ddots & \ddots & & & & \\ & & & \star & \star & \star & & \\ & & & & \ddots & \ddots & & \\ & & & & & \star & \star & \\ & & & & & & \star & \end{pmatrix} \begin{pmatrix} \star & \star & & & & & & \\ & \star & \star & & & & & \\ & & \star & \ddots & & & & \\ & & & \ddots & \star & & & \\ & & & & \star & & & \\ & & & & \star & \ddots & & \\ & & & & & \ddots & \star & \\ & & & & & & \star & \star \end{pmatrix},$$

or $T = PQ$, where we have assumed that no zero diagonal element is created in P or Q. Such decompositions exist if A is symmetric positive definite, or if A is an M-matrix, or when A is diagonally dominant. The twisted factorization and subsequent forward and back substitutions with P and Q take as many arithmetic operations as the standard factorization, and can be carried out with twofold parallelism by working from both ends of the matrix simultaneously. For an analysis of this process for tridiagonal systems, see [40]. Twisted factorization can be combined with any of the following techniques, often doubling the parallelism.

The other techniques we will discuss can all be applied to general banded systems, for which most were originally proposed, but for ease of exposition we will illustrate them just with a lower unit bidiagonal system $Lx = b$. A straight forward parallelization approach is to eliminate the unknown x_{i-1} from equation i using equation $i-1$, for all i in parallel. This leads to a new system in which each x_i is coupled only with x_{i-2}. Thus, the original system splits in two independent lower bidiagonal systems of half the size, one for the odd-numbered unknowns, and one for the even-numbered unknowns. This process can be repeated recursively for both new systems, leading to an algorithm known as *recursive doubling* [41]. It has been analyzed and generalized for banded systems in [42]. Its significance for modern parallel computers is limited, which we illustrate with the following examples.

Suppose we perform a single step of recursive doubling. This step can be done in parallel, but it involves slightly more arithmetic than the serial elimination process for solving $Lx = b$. The two resulting lower bidiagonal systems can be solved in parallel. This implies that on a two-processor system the time for a single step of recursive doubling will be slightly more than the time for solving the original system with only one processor. If we have n processors (where n is the dimension of L), then the elimination step can be done in very few time steps, and the two resulting systems can be solved in parallel, so that we have a speedup of about 2. However, this is not very practical, since during most of the time $n-2$ processors are idle, or formulated differently, the efficiency of the processors is rather low.

If we use n processors to apply this algorithm recursively instead of splitting into just two systems, we can solve in $\mathcal{O}(\log n)$ steps, a speedup of $\mathcal{O}(n/\log n)$, but the efficiency decreases like $\mathcal{O}(1/\log n)$. This is theoretically attractive but inefficient. Because of the data movement required, it is unlikely to be fast without system support for this communication pattern.

A related approach, which avoids the two subsystems, is to eliminate only the odd-numbered unknowns x_{i-1} from the even-numbered equations i. Again, this can be done in parallel, or in vector mode, and it results in a new system in which only the even-numbered unknowns are coupled. After having solved this reduced system, the odd-numbered unknowns can be computed in parallel from the odd-numbered equations. Of course, the trick can be repeated for the subsystem of half size, and this process is known as *cyclic reduction* [43, 44]. Since the amount of serial work is halved in each step by completely parallel (or vectorizable) operations, this approach has been successfully applied on vector supercomputers, especially when the vector speed of the machine is significantly greater than the scalar speed [38, 45, 46]. For distributed memory computers the method requires too much data

movement for the reduced system to be practical.

However, the method is easily generalized to one with more parallelism. Cyclic reduction can be viewed as an approach in which the given matrix L is written as a lower block bidiagonal matrix with 2×2 blocks along the diagonal. In the elimination process all $(2,1)$ positions in the diagonal blocks are eliminated in parallel. An obvious idea is to subdivide the matrix into larger blocks, i.e. we write L as a block bidiagonal matrix with $k \times k$ blocks along the diagonal (for simplicity we assume that n is a multiple of k). In practical cases k is chosen so large that the process is not repeated for the resulting subsystems, as for cyclic reduction (where $k = 2$). This approach is referred to as a *divide-and-conquer* approach. For banded triangular systems it was first suggested in [47], for tridiagonal systems it was proposed in [48].

To illustrate, let us apply one parallel elimination step to the lower bidiagonal system $Lx = b$ to eliminate all subdiagonal elements in all diagonal blocks. This yields a system $\tilde{L}x = \tilde{b}$, where for $k = 4$ and $n = 16$ we get

$$\tilde{L} = \begin{pmatrix} 1 & & & & & & & & & & & & & & & \\ & 1 & & & & & & & & & & & & & & \\ & & 1 & & & & & & & & & & & & & \\ & & & 1 & & & & & & & & & & & & \\ & & & \star & 1 & & & & & & & & & & & \\ & & & \star & & 1 & & & & & & & & & & \\ & & & \star & & & 1 & & & & & & & & & \\ & & & \star & & & & 1 & & & & & & & & \\ & & & & & & & \star & 1 & & & & & & & \\ & & & & & & & \star & & 1 & & & & & & \\ & & & & & & & \star & & & 1 & & & & & \\ & & & & & & & \star & & & & 1 & & & & \\ & & & & & & & \star & & & & & 1 & & & \\ & & & & & & & \star & & & & & & 1 & & \\ & & & & & & & \star & & & & & & & 1 & \\ & & & & & & & \star & & & & & & & & 1 \end{pmatrix}. \tag{2}$$

There are two possibilities for the next step. In the original approach [48], the fill-in in the subdiagonal blocks is eliminated in parallel, or vector mode, for each subdiagonal block (note that each subdiagonal block has only one column with nonzero elements). It has been shown in [49] that this leads to very efficient vectorized code for machines such as Cray, Fujitsu, etc.

For parallel computers, the parallelism in eliminating these subdiagonal blocks is relatively fine-grained compared with the more coarse-grained parallelism in the first step of the algorithm. Furthermore, on distributed memory machines the data for each subdiagonal block has to be spread over all processors. In [50] it has been shown that this limits the performance of the algorithm, the speedup being bounded by the ratio of computational speed and communication speed. This ratio is often very low [50].

The other approach is first to eliminate successively the last nonzero elements in the subdiagonal blocks $\tilde{L}_{j,j-1}$. This can be done with a short recurrence of length $n/k - 1$, after which all fill-in can be eliminated in parallel. For the recurrence we need some data communication between processors. However, for k large enough with respect to n/k, one can attain speedups close to $2k/5$ for this algorithm on a k processor system [51]. For a generalization of the divide-and-conquer approach for banded systems, see [52]; the data transport aspects for distributed memory machines have been discussed in [50].

There are other variants of the divide-and-conquer approach that move the fill-in into other columns of the subblocks or are more stable numerically. For example, in [53] the matrix is split into a block diagonal matrix and a remainder via rank-1 updates.

8 Gaussian Elimination on Sparse Matrices

8.1 Cholesky Factorization

In this section we discuss parallel algorithms for solving sparse systems of linear equations by direct methods. Paradoxically, sparse matrix factorization offers additional opportunities for exploiting parallelism beyond those available with dense matrices, yet it is usually more difficult to attain good efficiency in the sparse case. We examine both sides of this paradox: the additional parallelism induced by sparsity, and the difficulty in achieving high efficiency in spite of it. We will see that regularity and locality play a similar role in determining performance in the sparse case as they do for dense matrices.

We couch most of our discussion in terms of the Cholesky factorization, $A = LL^T$, where A is symmetric positive definite (SPD) and L is lower triangular with positive diagonal entries. We focus on Cholesky factorization primarily because this allows us to discuss parallelism in relative isolation, without the additional complications of pivoting for numerical stability. Most of the lessons learned are also applicable to other matrix factorizations, such as LU and QR. We do not try to give an exhaustive survey of research in this area, which is currently very active, instead referring the reader to existing surveys, such as [54]. Our main point in the current discussion is to explain how the sparse case differs from the dense case, and examine the performance implications of those differences.

We begin by considering the main features of sparse Cholesky factorization that affect its performance on serial machines. Algorithm 8.1 gives a standard, column-oriented formulation in which the Cholesky factor L overwrites the initial matrix A, and only the lower triangle is accessed:

Algorithm 8.1 *Column Oriented Cholesky factorization*

for $j = 1,\ n$
  for $k = 1,\ j-1$
    for $i = j,\ n$ $\{\mathrm{cmod}(j,k)\}$
      $a_{ij} = a_{ij} - a_{ik} \cdot a_{jk}$

$$
\begin{array}{l}
a_{jj} = \sqrt{a_{jj}} \\
\textit{for } k = j+1,\ n \qquad \{\mathrm{cdiv}(j)\} \\
\quad a_{kj} = a_{kj}/a_{jj}
\end{array}
$$

The outer loop in Algorithm 8.1 is over successive columns of A. The current column (indexed by j) is modified by a multiple of each prior column (indexed by k); we refer to such an operation as $\mathrm{cmod}(j,k)$. The computation performed by the inner loop (indexed by i) is a `saxpy`. After all its modifications have been completed, column j is then scaled by the reciprocal of the square root of its diagonal element; we refer to this operation as $\mathrm{cdiv}(j)$. As usual, this is but one of the 3! ways of ordering the triple-nested loop that embodies the factorization.

The inner loop in Algorithm 8.1 has no effect, and thus may as well be skipped, if $a_{jk} = 0$. For a dense matrix A, such an event is too unlikely to offer significant advantage. The fundamental difference with a sparse matrix is that a_{jk} is in fact very often zero, and computational efficiency demands that we recognize this situation and take advantage of it. Another way of expressing this condition is that column j of the Cholesky factor L does not depend on prior column k if $\ell_{jk} = 0$, which not only provides a computational shortcut, but also suggests an additional source of parallelism that we will explore in detail later.

8.2 Sparse Matrices

Thus far we have not said what we mean by a 'sparse' matrix. A good operational definition is that a matrix is sparse if it contains enough zero entries to be worth taking advantage of them to reduce both the storage and work required in solving a linear system. Ideally, we would like to store and operate on only the nonzero entries of the matrix, but such a policy is not necessarily a clear win in either storage or work. The difficulty is that sparse data structures include more overhead (to store indices as well as numerical values of nonzero matrix entries) than the simple arrays used for dense matrices, and arithmetic operations on the data stored in them usually cannot be performed as rapidly either (due to indirect addressing of operands). There is therefore a tradeoff in memory requirements between sparse and dense representations and a tradeoff in performance between the algorithms that use them. For this reason, a practical requirement for a family of matrices to be 'usefully' sparse is that they have only $\mathcal{O}(n)$ nonzero entries, i.e. a (small) constant number of nonzeros per row or column, independent of the matrix dimension. For example, most matrices arising from finite difference or finite element discretizations of PDEs satisfy this condition. In addition to the number of nonzeros, their particular locations, or pattern, in the matrix also has a major effect on how well sparsity can be exploited. Sparsity arising from physical problems usually exhibits some systematic pattern that can be exploited effectively, whereas the same number of nonzeros located randomly might offer relatively little advantage.

In Algorithm 8.1, the modification of a given column of the matrix by a prior column not only changes the existing nonzero entries in the target column, but may also introduce new nonzero entries in the target column. Thus, the Cholesky factor L may have additional nonzeros, called *fill*, in locations that were zero in the original matrix A. In determining

the storage requirements and computational work, these new nonzeros that the matrix gains during the factorization are equally as important as the nonzeros with which the matrix starts out.

The amount of such fill is dramatically affected by the order in which the columns of the matrix are processed. For example, if the first column of the matrix A is completely dense, then all of the remaining columns, no matter how sparse they start out, will completely fill in with nonzeros during the factorization. On the other hand, if a single such dense column is permuted (symmetrically) to become the last column in the matrix, then it will cause no fill at all. Thus, a critical part of the solution process for sparse systems is to determine an ordering for the rows and columns of the input matrix that limits fill to preserve sparsity. Unfortunately, finding an ordering that minimizes fill is a very hard combinatorial problem (NP-complete), but heuristics are available that do a good job of reducing, if not exactly minimizing, fill. These techniques include minimum degree, nested dissection, and various schemes for reducing the bandwidth or profile of a matrix (see, e.g., [55, 56]). for details on these and many other concepts used in sparse matrix computations).

One of the key advantages of SPD matrices is that such a sparsity preserving ordering can be selected in advance of the numeric factorization, independent of the particular values of the nonzero entries: only the pattern of the nonzeros matters, not their numerical values. This would not be the case, in general, if we also had to take into account pivoting for numerical stability, which obviously would require knowledge of the nonzero values, and would introduce a potential conflict between preserving sparsity and preserving stability. For the SPD case, once the ordering is selected, the locations of all fill elements in L can be anticipated prior to the numeric factorization, and thus an efficient static data structure can be set up in advance to accommodate them (this process is usually called *symbolic factorization*). This feature also stands in contrast to general sparse linear systems, which usually require dynamic data structures to accommodate fill entries as they occur, since their locations depend on numerical information that becomes known only as the numeric factorization process unfolds. Thus, modern algorithms and software for solving sparse SPD systems include a symbolic preprocessing phase in which a sparsity-preserving ordering is computed and a static data structure is set up for storing the entries of L before any floating point computation takes place.

We introduce some concepts and notation that will be useful in our subsequent discussion of parallel sparse Cholesky factorization. An important tool in understanding the combinatorial aspects of sparse Cholesky factorization is the notion of the *graph* of a symmetric $n \times n$ matrix A, which is an undirected graph having n vertices, with an edge between two vertices i and j if the corresponding entry a_{ij} of the matrix is nonzero. We denote the graph of A by $G(A)$. The structural effect of the factorization process can then be described by observing that the elimination of a variable adds fill edges to the corresponding graph so that the neighbors of the eliminated vertex become a clique (i.e. a fully connected subgraph). We also define the *filled graph*, denoted by $F(A)$, as having an edge between vertices i and j, with $i > j$, if $\ell_{ij} \neq 0$ in the Cholesky factor L (i.e. $F(A)$ is simply $G(A)$ with all fill edges added).

We use the notation M_{i*} to denote the ith row, and M_{*j} to denote the jth column, of a matrix M. For a given sparse matrix M, we define

$$\text{Struct}(M_{i*}) = \{k < i \mid m_{ik} \neq 0\}$$

and

$$\text{Struct}(M_{*j}) = \{k > j \mid m_{kj} \neq 0\}.$$

In other words, $\text{Struct}(M_{i*})$ is the sparsity structure of row i of the strict lower triangle of M, while $\text{Struct}(M_{*j})$ is the sparsity structure of column j of the strict lower triangle of M. For the Cholesky factor L, we define the *parent* function as follows:

$$\text{parent}(j) = \begin{cases} \min\{i \in \text{Struct}(L_{*j})\}, & \text{if } \text{Struct}(L_{*j}) \neq \emptyset, \\ j & \text{otherwise.} \end{cases}$$

Thus, $\text{parent}(j)$ is the row index of the first offdiagonal nonzero in column j of L, if any, and has the value j otherwise. Using the parent function, we define the *elimination tree* as a graph having n vertices, with an edge between vertices i and j, for $i > j$, if $i = \text{parent}(j)$. If the matrix is irreducible, then the elimination tree is indeed a single tree with its root at vertex n (otherwise it is more accurately termed an *elimination forest*). The elimination tree, which we denote by $T(A)$, is a spanning tree for the filled graph $F(A)$. The many uses of the elimination tree in analyzing and organizing sparse Cholesky factorization are surveyed in [57]. We will illustrate these concepts pictorially in several examples below.

8.3 Sparse Factorization

There are three basic types of algorithms for Cholesky factorization, depending on which of the three indices is placed in the outer loop:

(1) *Row-Cholesky:* Taking i in the outer loop, successive rows of L are computed one by one, with the inner loops solving a triangular system for each new row in terms of the previously computed rows.

(2) *Column-Cholesky:* Taking j in the outer loop, successive columns of L are computed one by one, with the inner loops computing a matrix–vector product that gives the effect of previously computed columns on the column currently being computed.

(3) *Submatrix-Cholesky:* Taking k in the outer loop, successive columns of L are computed one by one, with the inner loops applying the current column as a rank-1 update to the remaining unreduced submatrix.

These three families of algorithms have markedly different memory reference patterns in terms of which parts of the matrix are accessed and modified at each stage of the factorization, as illustrated in Figure 5, and each has its advantages and disadvantages in a given context.

For sparse Cholesky factorization, row-Cholesky is seldom used for a number of reasons, including the difficulty in providing a row-oriented data structure that can be accessed

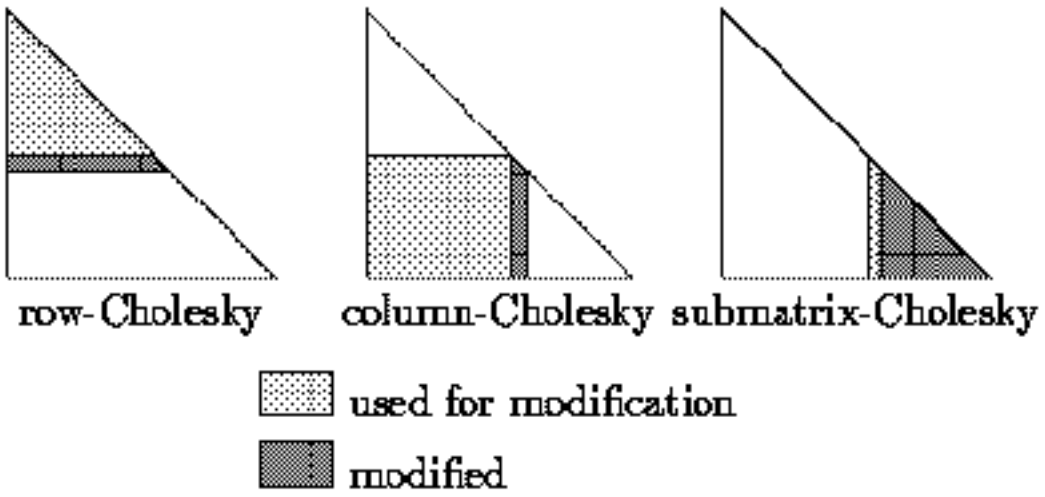

Figure 5: Three forms of Cholesky factorization.

efficiently during the factorization, and the difficulty in vectorizing or parallelizing the triangular solutions required. We will therefore focus our attention on the column-oriented methods, column-Cholesky and submatrix-Cholesky. Expressed in terms of the column operations cmod and cdiv and the Struct notation defined earlier, sparse column-Cholesky can be stated as follows:

Algorithm 8.2 *Sparse column-Cholesky factorization*

for $j = 1,\ n$
  for $k \in \text{Struct}(L_{j*})$
    $\text{cmod}(j, k)$
  $\text{cdiv}(j)$

In column-Cholesky, a given column j of A remains unchanged until the outer loop index reaches that value of j. At that point column j is updated by a nonzero multiple of each column $k < j$ of L for which $\ell_{jk} \neq 0$. After all column modifications have been applied to column j, the diagonal entry ℓ_{jj} is computed and used to scale the completely updated column to obtain the remaining nonzero entries of L_{*j}. Column-Cholesky is sometimes said to be a 'left-looking' algorithm, since at each stage it accesses needed columns to the left of the current column in the matrix. It can also be viewed as a 'demand-driven' algorithm, since the inner products that affect a given column are not accumulated until actually needed to modify and complete that column. For this reason, Ortega [38] terms column-Cholesky a 'delayed-update' algorithm. It is also sometimes referred to as a 'fan-in' algorithm, since the basic operation is to combine the effects of multiple previous columns on a single target column. The column-Cholesky algorithm is the most commonly used method in commercially available sparse matrix packages.

Similarly, sparse submatrix-Cholesky can be expressed as follows.

Algorithm 8.3 *Sparse submatrix-Cholesky factorization*

```
for k = 1, n
    cdiv(k)
    for j ∈ Struct(L_*k)
        cmod(j, k)
```

In submatrix-Cholesky, as soon as column k has been computed, its effects on all subsequent columns are computed immediately. Thus, submatrix-Cholesky is sometimes said to be a 'right-looking' algorithm, since at each stage columns to the right of the current column are modified. It can also be viewed as a 'data-driven' algorithm, since each new column is used as soon as it is completed to make all modifications to all the subsequent columns it affects. For this reason, Ortega [38] terms submatrix-Cholesky an 'immediate-update' algorithm. It is also sometimes referred to as a 'fan-out' algorithm, since the basic operation is for a single column to affect multiple subsequent columns. We will see that these characterizations of the column-Cholesky and submatrix-Cholesky algorithms have important implications for parallel implementations.

We note that many variations and hybrid implementations that lie somewhere between pure column-Cholesky and pure submatrix-Cholesky are possible. Perhaps the most important of these are the multi-frontal methods (see, e.g., [55]), in which updating operations are accumulated in and propagated through a series of *front matrices* until finally being incorporated into the ultimate target columns. Multifrontal methods have a number of attractive advantages, most of which accrue from the localization of memory references in the front matrices, thereby facilitating the effective use of memory hierarchies, including cache, virtual memory with paging, or explicit out-of-core solutions (the latter was the original motivation for these methods [58]. In addition, since the front matrices are essentially dense, the operations on them can be done using optimized kernels, such as the BLAS, to take advantage of vectorization or any other available architectural features. For example, such techniques have been used to attain very high performance for sparse factorization on conventional vector supercomputers [59] and on RISC workstations [60].

8.4 Parallelism in Sparse Factorization

We now examine in greater detail the opportunities for parallelism in sparse Cholesky factorization and various algorithms for exploiting it. One of the most important issues in designing any parallel algorithm is selecting an appropriate level of *granularity*, by which we mean the size of the computational subtasks that are assigned to individual processors. The optimal choice of task size depends on the tradeoff between communication costs and the load balance across processors. We follow Liu [61] in identifying three potential levels of granularity in a parallel implementation of Cholesky factorization:

(1) *fine-grain*, in which each task consists of only one or two floating point operations, such as a multiply-add pair,

(2) *medium-grain*, in which each task is a single column operation, such as cmod or cdiv,

(3) *large-grain*, in which each task is the computation of an entire group of columns in a subtree of the elimination tree.

Fine-grain parallelism, at the level of individual floating point operations, is available in either the dense or sparse case. It can be exploited effectively by a vector processing unit or a systolic array, but would incur far too much communication overhead to be exploited profitably on most current generation parallel computers. In particular, the communication latency of these machines is too great for such frequent communication of small messages to be feasible.

Medium-grain parallelism, at the level of operations on entire columns, is also available in either the dense or the sparse case. This level of granularity accounts for essentially all of the parallel speedup in dense factorization on current generation parallel machines, and it is an extremely important source of parallelism for sparse factorization as well. This parallelism is due primarily to the fact that many cmod operations can be computed simultaneously by different processors. For many problems, such a level of granularity provides a good balance between communication and computation, but scaling up to very large problems and/or very large numbers of processors may necessitate that the tasks be further broken up into chunks based on a two-dimensional partitioning of the columns. One must keep in mind, however, that in the sparse case an entire column operation may require only a few floating point operations involving the sparsely populated nonzero elements in the column. For a matrix of order n having a planar graph, for example, the largest embedded dense submatrix to be factored is roughly of order $\sqrt{n}$, and thus a sparse problem must be extremely large before a two-dimensional partitioning becomes essential.

Large-grain parallelism, at the level of subtrees of the elimination tree, is available only in the sparse case. If T_i and T_j are disjoint subtrees of the elimination tree, with neither root node a descendant of the other, then all of the columns corresponding to nodes in T_i can be computed completely independently of the columns corresponding to nodes in T_j, and *vice versa*, and hence these computations can be done simultaneously by separate processors with no communication between them. For example, each leaf node of the elimination tree corresponds to a column of L that depends on no prior columns, and hence all of the leaf node columns can be completed immediately merely by performing the corresponding cdiv operation on each of them. Furthermore, all such cdiv operations can be performed simultaneously by separate processors (assuming enough processors are available). By contrast, in the dense case all cdiv operations must be performed sequentially (at least at this level of granularity), since there is never more than one leaf node at any given time.

We see from this discussion that the elimination tree serves to characterize the parallelism that is unique to sparse factorization. In particular, the height of the elimination tree gives a rough measure of the parallel computation time, and the width of the elimination tree gives

a rough measure of the degree or multiplicity of large-grain parallelism. These measures are only very rough, however, since the medium level parallelism also plays a major role in determining overall performance. Still, we can see that short, bushy elimination trees are more advantageous then tall, slender ones in terms of the large-grain parallelism available. And just as the fill in the Cholesky factor is very sensitive to the ordering of the matrix, so is the structure of the elimination tree. This suggests that we should choose an ordering to enhance parallelism, and indeed this is possible (see, e.g., [62, 63, 64]) but such an objective may conflict to some degree with preservation of sparsity. Roughly speaking, sparsity and parallelism are largely compatible, since the large-grain parallelism is due to sparsity in the first place. However, these two criteria are by no means coincident, as we will see by example below.

We now illustrate these concepts using a series of simple examples. Figure 6 shows a small one-dimensional mesh with a 'natural' ordering of the nodes, the nonzero patterns of the corresponding tridiagonal matrix A and its Cholesky factor L, and the resulting elimination tree $T(A)$. On the positive side, the Cholesky factor suffers no fill at all and the total work required for the factorization is minimal. However, we see that the elimination tree is simply a chain, and therefore there is no large-grain parallelism available. Each column of L depends on the immediately preceding one, and thus they must be computed sequentially. This behavior is typical of orderings that minimize the bandwidth of a sparse matrix: they tend to inhibit rather than enhance large-grain parallelism in the factorization. (As previously discussed in Section 7, there is in fact little parallelism of any kind to be exploited in solving a tridiagonal system in this natural order. The cmod operations involve only a couple of flops each, so that even the 'medium-grain' tasks are actually rather small in this case.)

Figure 7 shows the same one-dimensional mesh with the nodes reordered by a minimum degree algorithm. Minimum degree is the most effective general purpose heuristic known for limiting fill in sparse factorization [65]. In its simplest form, this algorithm begins by selecting a node of minimum degree (i.e. one having fewest incident edges) in $G(A)$ and numbering it first. The selected node is then deleted and new edges are added, if necessary, to make its former neighbors into a clique. The process is then repeated on the updated graph, and so on, until all nodes have been numbered. We see in Figure 7 that L suffers no fill in the new ordering, and the elimination tree now shows some large-grain parallelism. In particular, columns 1 and 2 can be computed simultaneously, then columns 3 and 4, and so on. This twofold parallelism reduces the tree height (roughly the parallel completion time) by approximately a factor of two.

At any stage of the minimum degree algorithm, there may be more than one node with the same minimum degree, and the quality of the ordering produced may be affected by the tie breaking strategy used. In the example of Figure 7, we have deliberately broken ties in the most favorable way (with respect to parallelism); the least favorable tie breaking would have reproduced the original ordering of Figure 6, resulting in no parallelism. Breaking ties randomly (which in general is about all one can do) could produce anything in between these two extremes, yielding an elimination tree that reveals some large-grain parallelism,

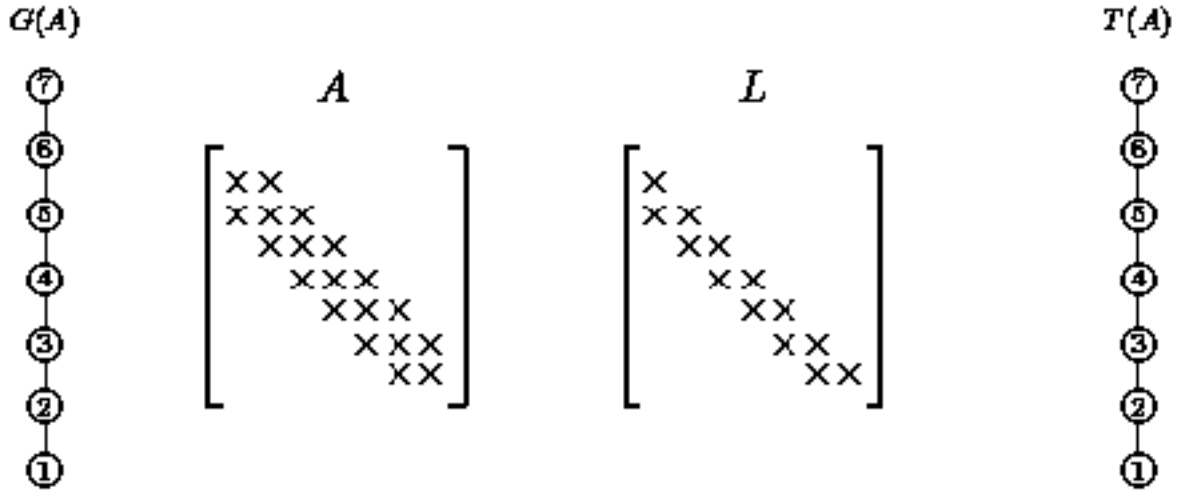

Figure 6: One-dimensional grid and corresponding tridiagonal matrix (left), with Cholesky factor and elimination tree (right).

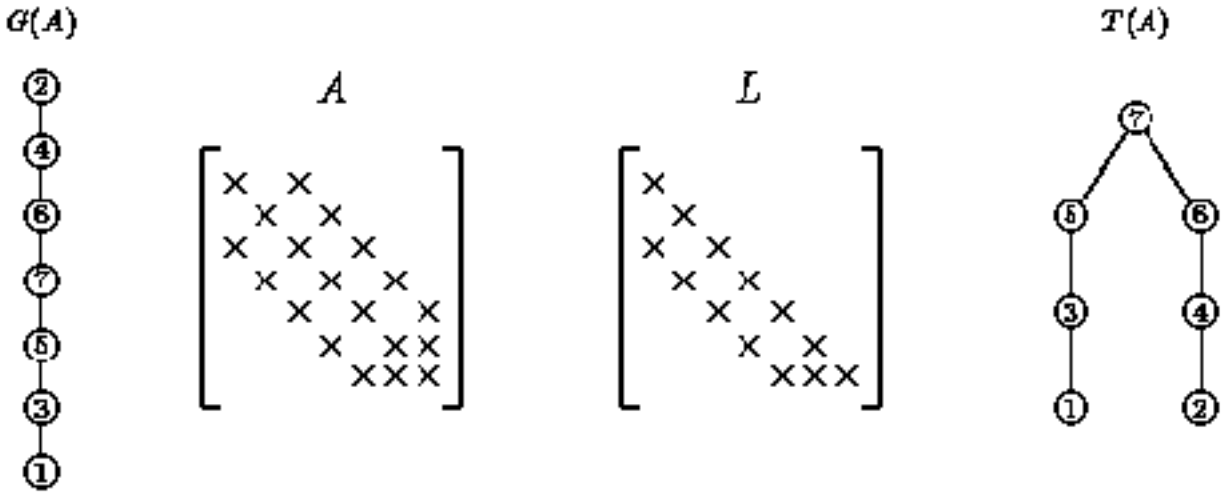

Figure 7: Graph and matrix reordered by minimum degree (left), with corresponding Cholesky factor and elimination tree (right).

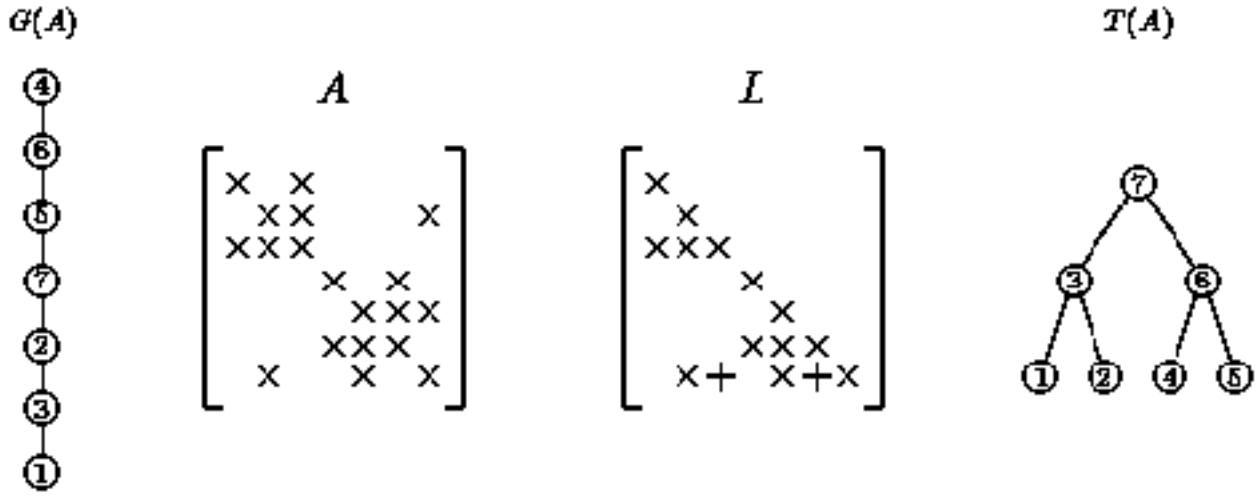

Figure 8: Graph and matrix reordered by nested dissection (left), with corresponding Cholesky factor and elimination tree (right).

but which is taller and less well balanced than our example in Figure 7. Again, this is typical of minimum degree orderings. In view of this property, Liu [64] has developed an interesting strategy for further reordering of an initial minimum degree ordering that preserves fill while reducing the height of the elimination tree.

Figure 8 shows the same mesh again, this time ordered by nested dissection, a divide-and-conquer strategy [66]. Let S be a set of nodes, called a *separator*, whose removal, along with all edges incident upon nodes in S, disconnects $G(A)$ into two remaining subgraphs. The nodes in each of the two remaining subgraphs are numbered contiguously and the nodes in the separator S are numbered last. This procedure is then applied recursively to split each of the remaining subgraphs, and so on, until all nodes have been numbered. If sufficiently small separators can be found, then nested dissection tends to do a good job of limiting fill, and if the pieces into which the graph is split are of about the same size, then the elimination tree tends to be well balanced. We see in Figure 8 that for our example, with this ordering, the Cholesky factor L suffers fill in two matrix entries (indicated by $+$), but the elimination tree now shows a fourfold large-grain parallelism, and its height has been reduced further. This behavior is again typical of nested dissection orderings: they tend to be somewhat less successful at limiting fill than minimum degree, but their divide-and-conquer nature tends to identify parallelism more systematically and produce better balanced elimination trees.

Finally, Figure 9 shows the same problem reordered by odd–even reduction. This is not a general purpose strategy for sparse matrices, but it is often used to enhance parallelism in tridiagonal and related systems, so we illustrate it for the sake of comparison with more general purpose methods. In odd–even reduction (see, e.g., [55]), odd node numbers come before even node numbers, and then this same renumbering is applied recursively within each

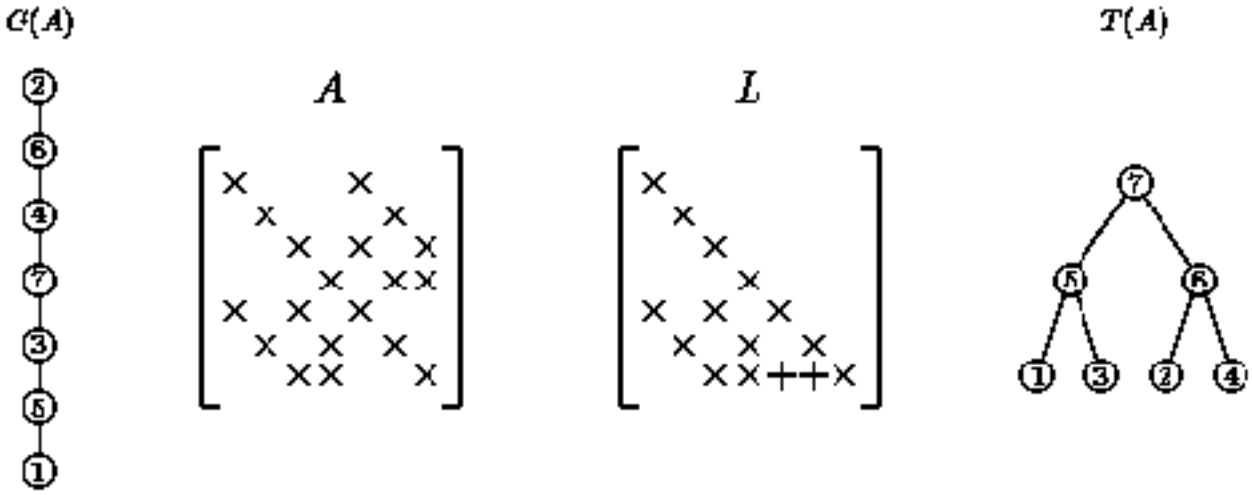

Figure 9: Graph and matrix reordered by odd–even reduction (left), with corresponding Cholesky factor and elimination tree (right).

resulting subset, and so on until all nodes are numbered. Although the resulting nonzero pattern of A looks superficially different, we can see from the elimination tree that this method is essentially equivalent to nested dissection for this type of problem.

8.5 Parallel Algorithms for Sparse Factorization

Having developed some understanding of the sources of parallelism in sparse Cholesky factorization, we now consider some algorithms for exploiting it. In designing any parallel algorithm, one of the most important decisions is how tasks are to be assigned to processors. In a shared memory parallel architecture, the tasks can easily be assigned to processors dynamically by maintaining a common pool of tasks from which available processors claim work to do. This approach has the additional advantage of providing automatic load balancing to whatever degree is permitted by the chosen task granularity. An implementation of this approach for parallel sparse factorization is given in [67].

In a distributed memory environment, communication costs often prohibit dynamic task assignment or load balancing, and thus we seek a static mapping of tasks to processors. In the case of column-oriented factorization algorithms, this amounts to assigning the columns of the matrix to processors according to some mapping procedure determined in advance. Such an assignment could be made using the block or wrap mappings, or combinations thereof, often used for dense matrices. However, such simple mappings risk wasting much of the large-grain parallelism identified by means of the elimination tree, and may also incur unnecessary communication. For example, the leaf nodes of the elimination tree can be processed in parallel if they are assigned to different processors, but the latter is not necessarily ensured by a simple block or wrap mapping.

A better approach for sparse factorization is to preserve locality by assigning subtrees of the elimination tree to contiguous subsets of neighboring processors. A good example of this technique is the 'subtree-to-subcube' mapping often used with hypercube multicomputers [68]. Of course, the same idea applies to other network topologies, such as submeshes of a larger mesh. We will assume that some such mapping is used, and we will comment further on its implications later. Whatever the mapping, we will denote the processor containing column j by map$[j]$, or, more generally, if J is a set of column numbers, map$[J]$ will denote the set of processors containing the given columns.

One of the earliest and simplest parallel algorithms for sparse Cholesky factorization is the following version of submatrix-Cholesky [69]. Algorithm 8.4 runs on each processor, with each responsible for its own subset, *mycols*, of columns.

Algorithm 8.4 *Distributed fan-out sparse Cholesky factorization*

for $j \in mycols$
  if j *is a leaf node in* $T(A)$
    cdiv(j)
    send L_{*j} *to processors in* map(Struct(L_{*j}))
    $mycols = mycols - \{j\}$
while $mycols \neq \emptyset$
  receive any column of L*, say* L_{*k}
  for $j \in mycols \cap$ Struct(L_{*k})
    cmod(j,k)
    if column j *requires no more* cmods
      cdiv(j)
      send L_{*j} *to processors in* map(Struct(L_{*j}))
      $mycols = mycols - \{j\}$

In Algorithm 8.4, any processor that owns a column of L corresponding to a leaf node of the elimination tree can complete it immediately merely by performing the necessary cdiv operation, since such a column depends on no prior columns. The resulting factor columns are then broadcast (fanned-out) to all other processors that will need them to update columns that they own. The remainder of the algorithm is then driven by the arrival of factor columns, as each processor goes into a loop in which it receives and applies successive factor columns, in whatever order they may arrive, to whatever columns remain to be processed. When the modifications of a given column have been completed, then the cdiv operation is done, the resulting factor column is broadcast as before, and the process continues until all columns of L have been computed.

Algorithm 8.4 potentially exploits both the large-grain parallelism characterized by concurrent cdivs and the medium-grain parallelism characterized by concurrent cmods, but this data-driven approach also has a number of drawbacks that severely limit its efficiency. In particular, performing the column updates one at a time by the receiving processors results in

unnecessarily high communication frequency and volume, and in a relatively inefficient computational inner loop. The communication requirements can be reduced by careful mapping and by aggregating updating information over subtrees (see, e.g., [70, 71, 72]) but even with this improvement, the fan-out algorithm is usually not competitive with other algorithms presented later. The shortcomings of the fan-out algorithm motivated the formulation of the following fan-in algorithm for sparse factorization, which is a parallel implementation of column-Cholesky [73].

Algorithm 8.5 *Distributed fan-in sparse Cholesky factorization*

```
for j = 1, n
    if j ∈ mycols or mycols ∩ Struct(L_j*) ≠ ∅
        u = 0
        for k ∈ mycols ∩ Struct(L_j*)
            u = u + ℓ_jk * L_*k        {aggregate column update s}
        if j ∈ mycols
            incorporate u into the factor column j
            while any aggregated update column for
                column j remains, receive in u another
                aggregated update column for column j, and
                incorporate it into the factor column j
            cdiv(j)
        else
            send u to processor map[j]
```

Algorithm 8.5 takes a demand-driven approach: the updates for a given column j are not computed until needed to complete that column, and they are computed by the sending processors rather than the receiving processor. As a result, all of a given processor's contributions to the updating of the column in question can be combined into a single aggregate update column, which is then transmitted in a single message to the processor containing the target column. This approach not only decreases communication frequency and volume, but it also facilitates a more efficient computational inner loop. In particular, no communication is required to complete the columns corresponding to any subtree that is assigned entirely to a single processor. Thus, with an appropriate locality-preserving and load-balanced subtree mapping, Algorithm 8.5 has a perfectly parallel, communication-free initial phase that is followed by a second phase in which communication takes place over increasingly larger subsets of processors as the computation proceeds up the elimination tree, encountering larger subtrees. This perfectly parallel phase, which is due entirely to sparsity, tends to constitute a larger proportion of the overall computation as the size of the problem grows for a fixed number of processors, and thus the algorithm enjoys relatively high efficiencies for sufficiently large problems.

In the fan-out and fan-in factorization algorithms, the necessary information flow between columns is mediated by factor columns or aggregate update columns, respectively. Another

alternative is a *multi-frontal* method, in which update information is mediated through a series of front matrices. In a sense, this represents an intermediate strategy, since the effect of each factor column is incorporated immediately into a front matrix, but its eventual incorporation into the ultimate target column is delayed until actually needed. The principal computational advantage of multi-frontal methods is that the frontal matrices are treated as dense matrices, and hence updating operations on them are much more efficient than the corresponding operations on sparse data structures that require indirect addressing. Unfortunately, although the updating computations employ simple dense arrays, the overall management of the front matrices is relatively complicated. As a consequence, multi-frontal methods are difficult to specify succinctly, so we will not attempt to do so here, but note that multi-frontal methods have been implemented for both shared-memory (e.g., [74, 75]) and distributed-memory (e.g., [76, 77]) parallel computers, and are among the most effective methods known for sparse factorization in all types of computational environments. For a unified description and comparison of parallel fan-in, fan-out and multi-frontal methods, see [78].

In this brief section on parallel direct methods for sparse systems, we have concentrated on numeric Cholesky factorization for SPD matrices. We have omitted many other aspects of the computation, even for the SPD case: computing the ordering in parallel, symbolic factorization, and triangular solution. More generally, we have omitted any discussion of LU factorization for general sparse square matrices or QR factorization for sparse rectangular matrices. Instead we have concentrated on identifying the major features that distinguish parallel sparse factorization from the dense case and examining the performance implications of those differences.

9 Iterative Methods for Linear Systems

In this section we discuss parallel aspects of iterative methods for solving large linear systems. For a good mathematical introduction to a class of successful and popular methods, the so-called Krylov subspace methods, see [79]. There are many such methods and new ones are frequently proposed. Fortunately, they share enough properties that to understand how to implement them in parallel it suffices to examine carefully just a few.

For the purposes of parallel implementation there are two classes of methods: those with short recurrences, i.e. methods that maintain only a very limited number of search direction vectors, and those with long recurrences. The first class includes CG (Conjugate Gradients), CR (Conjugate Residuals), Bi-CG, CGS (CG squared), QMR (Quasi Minimum Residual), GMRES(m) for small m (Generalized Minimum Residual), truncated ORTHOMIN (Orthogonal Minimum Residual), Chebychev iteration, and so on. We could further distinguish between methods with fixed iteration parameters and methods with dynamical parameters, but we will not do so; the effects of this aspect will be clear from our discussion. As the archetype for this class we will consider CG; the parallel implementation issues for this method apply to most other short recurrence methods. The second class of methods includes GMRES, GMRES(m) with larger m, ORTHOMIN, ORTHODIR (Orthogonal Directions),

ORTHORES (Orthogonal Residuals), and EN (Eirola–Nevanlinna's Rank-1 update method). We consider GMRES in detail, which is a popular method in this class.

This section is organized as follows. In Section 9.1 we will discuss the parallel aspects of important computational kernels in iterative schemes. From the discussions it should be clear how to combine coarse-grained and fine-grained approaches, for example when implementing a method on a parallel machine with vector processors. The implementation for such machines, in particular those with shared memory, is given much attention in [8]. In Section 9.2, coarse-grained parallel and data-locality issues of CG will be discussed, while in Section 9.3 the same will be done for GMRES.

9.1 Parallelism in the Kernels of Iterative Methods

The basic time-consuming computational kernels of iterative schemes are usually:

(1) inner products,

(2) vector updates,

(3) matrix–vector products, like Ap_i (for some methods also $A^T p_i$),

(4) preconditioning (e.g., solve for w in $Kw = r$).

The inner products can be easily parallelized; each processor computes the inner product of two segments of each vector (local inner products or LIPs). On distributed memory machines the LIPs have to be sent to other processors in order to be reduced to the required global inner product. This step requires communication. For shared memory machines the inner products can be computed in parallel without difficulty. If the distributed memory system supports overlap of communication with computation, then we seek opportunities in the algorithm to do so. In the standard formulation of most iterative schemes this is usually a major problem. We will come back to this in the next two sections. Vector updates are trivially parallelizable: each processor updates its 'own' segment. The matrix–vector products are often easily parallelized on shared memory machines by splitting the matrix into strips corresponding to the vector segments. Each processor takes care of the matrix–vector product of one strip.

For distributed memory machines there may be a problem if each processor has only a segment of the vector in its memory. Depending on the bandwidth of the matrix we may need communication for other elements of the vector, which may lead to communication problems. However, many sparse matrix problems are related to a network in which only nearby nodes are connected. In such a case it seems natural to subdivide the network, or grid, in suitable blocks and to distribute these blocks over the processors. When computing Ap_i each processor needs at most the values of p_i at some nodes in neighboring blocks. If the number of connections to these neighboring blocks is small compared to the number of internal nodes, then the communication time can be overlapped with computational work.

For more detailed discussions on implementation aspects on distributed memory systems, see [80] and [81].

Preconditioning is often the most problematic part in a parallel environment. Incomplete decompositions of A form a popular class of preconditionings in the context of solving discretized PDEs. In this case the preconditioner $K = LU$, where L and U have a sparsity pattern equal or close to the sparsity pattern of the corresponding parts of A (L is lower triangular, U is upper triangular). For details see [4], [82] and [83]. Solving $Kw = r$ leads to solving successively $Lz = r$ and $Uw = z$. These triangular solves lead to recurrence relations that are not easily parallelized. We will now discuss a number of approaches to obtain parallelism in the preconditioning part.

(1) *Reordering the computations.* Depending on the structure of the matrix a *frontal approach* may lead to successful parallelism. By inspecting the dependency graph one can select those elements that can be computed in parallel. For instance, if a second order PDE is discretized by the usual five-point star over a rectangular grid, then the triangular solves can be parallelized if the computation is carried out along diagonals of the grid, instead of the usual lexicographical order. For vector computers this leads to a vectorizable preconditioner (see [84, 8, 85, 86]). For coarse-grained parallelism this approach is insufficient. By a similar approach more parallelism can be obtained in three-dimensional situations: the so-called *hyperplane approach* [87, 85, 86]. The disadvantage is that the data need to be redistributed over the processors, since the grid points, which correspond to a hyperplane in the grid, are located quite irregularly in the array. For shared memory machines this also leads to reduced performance because of indirect addressing. In general one concludes that the data dependency approach is not adequate for obtaining a suitable degree of parallelism.

(2) *Reordering the unknowns.* One may also use a *coloring scheme* for reordering the unknowns, so that unknowns with the same color are not explicitly coupled. This means that the triangular solves can be parallelized for each color. Of course, communication is required for couplings between groups of different colors. Simple coloring schemes, like red-black ordering for the five-point discretized Poisson operator, seem to have a negative effect on the convergence behavior. Duff and Meurant [88] have carried out numerical experiments for many different orderings, which show that the numbers of iterations may increase significantly for other than lexicographical ordering. Some modest degree of parallelism can be obtained, however, with so-called incomplete twisted factorizations [8, 89, 85]. Multi-color schemes with a large number of colors (e.g., 20 to 100) may lead to little or no degradation in convergence behavior [90]. but also to less parallelism. Moreover, the ratio of computation to communication may be more unfavorable.

(3) *Forced parallelism.* Parallelism can also be forced by simply neglecting couplings to unknowns residing in other processors. This is like block Jacobi preconditioning, in which the blocks may be decomposed in incomplete form [91]. Again, this may not always reduce the overall solution time, since the effects of increased parallelism are more

than undone by an increased number of iteration steps. In order to reduce this effect, it is suggested in [92] to construct incomplete decompositions on slightly overlapping domains. This requires communication similar to that of matrix–vector products. In [92] results are reported on a six-processor shared memory system (IBM3090), and speedups close to 6 have been observed.

The problems with parallelism in the preconditioner have led to searches for other preconditioners. Often simple diagonal scaling is an adequate preconditioner and this is trivially parallelizable. For results on a Connection Machine, see [93]. Often this approach leads to a significant increase in iteration steps. Still another approach is to use polynomial preconditioning: $w = p_j(A)r$, i.e. $K^{-1} = p_j(A)$, for some suitable jth degree polynomial. This preconditioner can be implemented by forming only matrix–vector products, which, depending on the structure of A, are easier to parallelize [94]. For p_j one often selects a Chebychev polynomial, which requires some information on the spectrum of A.

Finally we point out the possibility of using the truncated Neumann series for the approximate inverse of A, or parts of L and U. Madsen *et al.* [95] discuss approximate inversion of A, which from the implementation point of view is equivalent to polynomial preconditioning. In [96] the use of truncated Neumann series for removing some of the recurrences in the triangular solves is discussed. This approach leads to only fine-grained parallelism (vectorization).

9.2 Parallelism and Data Locality in Preconditioned CG

To use CG to solve $Ax = b$, A must be symmetric and positive definite. In other short recurrence methods, other properties of A may be required or desirable, but we will not exploit these properties explicitly here.

Most often, CG is used in combination with some kind of preconditioning [79, 4, 97]. This means that the matrix A is implicitly multiplied by an approximation K^{-1} of A^{-1}. Usually, K is constructed to be an approximation of A, and so that $Ky = z$ is easier to solve than $Ax = b$. Unfortunately, a popular class of preconditioners, those based on incomplete factorizations of A, are hard to parallelize. We have discussed some efforts to obtain more parallelism in the preconditioner in Section 9.1. Here we will assume the preconditioner is chosen such that the time to solve $Ky = z$ in parallel is comparable with the time to compute Ap. For CG we also require that the preconditioner K be symmetric positive definite. We exploit this to implement the preconditioner more efficiently.

The preconditioned CG algorithm is as follows.

Algorithm 9.1 *Preconditioned Conjugate Gradients – variant 1*

x_0= *initial guess;* $r_0 = b - Ax_0$;
$p_{-1} = 0; \beta_{-1} = 0$;
Solve for w_0 *in* $Kw_0 = r_0$;
$\rho_0 = (r_0, w_0)$

for $i = 0, 1, 2, ...$
$p_i = w_i + \beta_{i-1} p_{i-1}$;
$q_i = Ap_i$;
$\alpha_i = \rho_i / (p_i, q_i)$
$x_{i+1} = x_i + \alpha_i p_i$;
$r_{i+1} = r_i - \alpha_i q_i$;
if x_{i+1} *accurate enough then quit;*
Solve for w_{i+1} *in* $K w_{i+1} = r_{i+1}$;
$\rho_{i+1} = (r_{i+1}, w_{i+1})$;
$\beta_i = \frac{\rho_{i+1}}{\rho_i}$;
$\beta_i = \rho_{i+1} / \rho_i$;
end;

If A or K is not very sparse, most work is done in multiplying $q_i = Ap_i$ or solving $K w_{i+1} = r_{i+1}$, and this is where parallelism is most beneficial. It is also completely dependent on the structures of A and K.

Now we consider parallelizing the rest of the algorithm. Note that updating x_{i+1} and r_{i+1} can only begin after completing the inner product for α_i. Since on a distributed memory machine communication is needed for the inner product, we cannot overlap this communication with useful computation. The same observation applies to updating p_i, which can only begin after completing the inner product for β_{i-1}. Apart from computing Ap_i and solving $K w_{i+1} = r_{i+1}$, we need to load 7 vectors for 10 vector floating point operations. This means that for this part of the computation only 10/7 floating point operation can be carried out per memory reference on average.

Several authors [98, 99, 100, 101] have attempted to improve this ratio, and to reduce the number of synchronization points (the points at which computation must wait for communication). In Algorithm 9.1 there are two such synchronization points, namely the computation of both inner products. Meurant [100] (see also [94]) has proposed a variant in which there is only one synchronization point, however at the cost of possibly reduced numerical stability, and one additional inner product. In this scheme the ratio between computations and memory references is about 2. We show here yet another variant, proposed by Chronopoulos and Gear [98].

Algorithm 9.2 *Preconditioned conjugate gradients – variant 2*

x_0= *initial guess;* $r_0 = b - Ax_0$;
$q_{-1} = p_{-1} = 0; \beta_{-1} = 0$;
Solve for w_0 *in* $K w_0 = r_0$;
$s_0 = Aw_0$;
$\rho_0 = (r_0, w_0); \mu_0 = (s_0, w_0)$;
$\alpha_0 = \rho_0 / \mu_0$;
for $i = 0, 1, 2, ...$
$p_i = w_i + \beta_{i-1} p_{i-1}$;

$q_i = s_i + \beta_{i-1} q_{i-1}$;
$x_{i+1} = x_i + \alpha_i p_i$;
$r_{i+1} = r_i - \alpha_i q_i$;
if x_{i+1} *accurate enough then quit;*
Solve for w_{i+1} *in* $K w_{i+1} = r_{i+1}$;
$s_{i+1} = A w_{i+1}$;
$\rho_{i+1} = (r_{i+1}, w_{i+1})$; $\mu_{i+1} = (s_{i+1}, w_{i+1})$;
$\beta_i = \rho_{i+1}/\rho_i$;
$\alpha_{i+1} = \rho_{i+1}/(\mu_{i+1} - \rho_{i+1}\beta_i/\alpha_i)$;
end;

In this scheme all vectors need be loaded only once per pass of the loop, which leads to improved data locality. However, the price is $2n$ extra flops per iteration step. Chronopoulos and Gear [98] claim the method is stable, based on their numerical experiments. Instead of two synchronization points, as in the standard version of CG, we have now only one such synchronization point, as the next loop can be started only when the inner products at the end of the previous loop have been completed. Another slight advantage is that these inner products can be computed in parallel.

Chronopoulos and Gear [98] propose to improve further the data locality and parallelism in CG by combining s successive steps. Their algorithm is based upon the following property of CG. The residual vectors $r_0, ..., r_i$ form an orthogonal basis (assuming exact arithmetic) for the Krylov subspace spanned by $r_0, Ar_0, ..., A^{i-1}r_0$. Given r_0 through r_j, the vectors $r_0, r_1, ..., r_j,\ Ar_j, ..., A^{i-j-1}r_j$ also span this subspace. Chronopoulos and Gear propose to combine s successive steps by generating $r_j, Ar_j, ..., A^{s-1}r_j$ first, and then to orthogonalize and update the current solution with this blockwise extended subspace. Their approach leads to slightly more flops than s successive steps of standard CG, and also one additional matrix–vector product every s steps. The implementation issues for vector register computers and distributed memory machines are discussed in great detail in [102].

The main drawback in this approach is potential numerical instability: depending on the spectrum of A, the set $r_j, ..., A^{s-1}r_j$ may converge to a vector in the direction of a dominant eigenvector, or in other words, may become dependent for large values of s. The authors claim success in using this approach without serious stability problems for small values of s. Nevertheless, it seems that s-step CG still has a bad reputation [103] because of these problems. However, a similar approach, suggested by Chronopoulos and Kim [104] for other processes such as GMRES, seems to be more promising. Several authors have pursued this direction, and we will come back to this in Section 9.3.

We consider another variant of CG, in which we may overlap all communication time with useful computations. This is just a reorganized version of the original CG scheme, and is therefore precisely as stable. The key trick is to delay the updating of the solution vector. Another advantage over the previous scheme is that no additional operations are required. We will assume that the preconditioner K can be written as $K = LL^T$. Furthermore, L has a block structure, corresponding to the grid blocks, so that any communication can again be overlapped with computation.

Algorithm 9.3 *Preconditioned conjugate gradients – variant 3*

$x_{-1} = x_0$= *initial guess;* $r_0 = b - Ax_0$;
$p_{-1} = 0; \beta_{-1} = 0; \alpha_{-1} = 0$;
$s = L^{-1} r_0$;
$\rho_0 = (s, s)$
for $i = 0, 1, 2, ...$
$w_i = L^{-T} s$; *(0)*
$p_i = w_i + \beta_{i-1} p_{i-1}$; *(1)*
$q_i = A p_i$; *(2)*
$\gamma = (p_i, q_i)$; *(3)*
$x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$; *(4)*
$\alpha_i = \rho_i / \gamma$; *(5)*
$r_{i+1} = r_i - \alpha_i q_i$; *(6)*
$s = L^{-1} r_{i+1}$; *(7)*
$\rho_{i+1} = (s, s)$; *(8)*
if r_{i+1} *small enough then* *(9)*
$x_{i+1} = x_i + \alpha_i p_i$
quit;
$\beta_i = \rho_{i+1} / \rho_i$;
end;

Under the assumptions that we have made, CG can be efficiently parallelized as follows.

(1) All compute intensive operations can be done in parallel. Only operations (2), (3), (7), (8), (9), and (0) require communication. We have assumed that the communication in (2), (7), and (0) can be overlapped with computation.

(2) The communication required for the reduction of the inner product in (3) can be overlapped with the update for x_i in (4), (which could in fact have been done in the previous iteration step).

(3) The reduction of the inner product in (8) can be overlapped with the computation in (0). Also step (9) usually requires information such as the norm of the residual, which can be overlapped with (0).

(4) Steps (1), (2), and (3) can be combined: the computation of a segment of p_i can be followed immediately by the computation of a segment of q_i in (2), and this can be followed by the computation of a part of the inner product in (3). This saves on load operations for segments of p_i and q_i.

(5) Depending on the structure of L, the computation of segments of r_{i+1} in (6) can be followed by operations in (7), which can be followed by the computation of parts of the inner product in (8), and the computation of the norm of r_{i+1}, required for (9).

(6) The computation of β_i can be done as soon as the computation in (8) has been completed. At that moment, the computation for (1) can be started if the requested parts of w_i have been completed in (0).

(7) If no preconditioner is used, then $w_i = r_i$, and steps (7) and (0) are skipped. Step (8) is replaced by $\rho_{i+1} = (r_{i+1}, r_{i+1})$. Now we need some computation to overlap the communication for this inner product. To this end, one might split the computation in (4) in two parts. The first part would be computed in parallel with (3), and the second part with ρ_{i+1}.

More recent work on removing synchronization points in CG while retaining numerical stability appears in [105, 106].

9.3 Parallelism and Data Locality in GMRES

GMRES, proposed by Saad and Schultz [107], is a CG-like method for solving general non-singular linear systems $Ax = b$. GMRES minimizes the residual over the Krylov subspace $\text{span}[r_0, Ar_0, A^2r_0, ..., A^ir_0]$, with $r_0 = b - Ax_0$. This requires, as with CG, the construction of an orthogonal basis of this space. Since we do not require A to be symmetric, we need long recurrences: each new vector must be explicitly orthogonalized against all previously generated basis vectors. In its most common form GMRES orthogonalizes using Modified Gram–Schmidt [4]. In order to limit memory requirements (since all basis vectors must be stored), GMRES is restarted after each cycle of m iteration steps; this is called GMRES(m). A slightly simplified version of GMRES(m) with preconditioning K is as follows (for details, see [108]):

Algorithm 9.4 *GMRES(m)*

x_0 is an initial guess; $r = b - Ax_0$;
for $j = 1, 2, ...$
  Solve for w in $Kw = r$;
  $v_1 = w/\|w\|_2$;
  for $i = 1, 2, ..., m$
    Solve for w in $Kw = Av_i$;
    for $k = 1, ..., i$ *orthogonalization of w*
      $h_{k,i} = (w, v_k)$; *against vs, by modified*
      $w = w - h_{k,i}v_k$; *Gram–Schmidt process*
    end k;
    $h_{i+1,i} = \|w\|_2$;
    $v_{i+1} = w/h_{i+1,i}$;
    apply $J_1, ..., J_{i-1}$ on $(h_{1,i}, ..., h_{i+1,i})$;
    construct J_i, acting on i–th and $(i+1)$–st component
    of $h_{.,i}$, such that $(i+1)$–st component of $J_ih_{.,i}$ is 0;

$s := J_i s$;
if $s(i+1)$ is small enough then (UPDATE($\tilde{x}, i$); quit);
end i;
Compute x_m using the $h_{k,i}$ and v_i;
$r = b - Ax_m$;
if residual r is small enough then quit
else ($x_0 := x_m$;);
end j;

Another scheme for GMRES, based upon Householder orthogonalization instead of modified Gram–Schmidt, has been proposed in [109]. For some applications the additional computation required by Householder orthogonalization is compensated by improved numerical properties: the better orthogonality saves iteration steps. In [110] a variant of GMRES is proposed in which the preconditioner itself may be an iterative process, which may help to increase parallel efficiency.

Similar to CG and other iterative schemes, the major computations are matrix–vector computations (with A and K), inner products and vector updates. All of these operations are easily parallelizable, although on distributed memory machines the inner products in the orthogonalization act as synchronization points. In this part of the algorithm, one new vector, $K^{-1}Av_j$, is orthogonalized against the previously built orthogonal set v_1, v_2, ... , v_j. In Algorithm 9.4, this is done using Level 1 BLAS, which may be quite inefficient. To incorporate Level 2 BLAS we can do either Householder orthogonalization or classical Gram–Schmidt twice (which mitigates classical Gram–Schmidt's potential instability [103]. Both approaches significantly increase the computational work and do not remove the synchronization and data-locality problems completely. Note that we cannot, as in CG, overlap the inner product computation with updating the approximate solution, since in GMRES this updating can be done only after completing a cycle of m orthogonalization steps.

The obvious way to extract more parallelism and data locality is to generate a basis v_1, Av_1, ..., $A^m v_1$ for the Krylov subspace first, and to orthogonalize this set afterwards; this is called m-step GMRES(m) [104]. This approach does not increase the computational work and, in contrast to CG, the numerical instability due to generating a possibly near-dependent set is not necessarily a drawback. One reason is that error cannot build up as in CG, because the method is restarted every m steps. In any case, the resulting set, after orthogonalization, is the basis of some subspace, and the residual is then minimized over that subspace. If, however, one wants to mimic standard GMRES(m) as closely as possible, one could generate a better (more independent) starting set of basis vectors v_1, $y_2 = p_1(A)v_1$, ..., $y_{m+1} = p_m(A)v_1$, where the p_j are suitable degree j polynomials. Newton polynomials are suggested in [111] and and Chebychev polynomials in [80].

After generating a suitable starting set, we still have to orthogonalize it. In [80] modified Gram–Schmidt is used while avoiding communication times that cannot be overlapped. We outline this approach, since it may be of value for other orthogonalization methods. Given a basis for the Krylov subspace, we orthogonalize by

for $k = 2, ..., m+1$:
  /* orthogonalize $y_k, ..., y_{m+1}$ w.r.t. v_{k-1} */
  for $j = k, ..., m+1$
    $y_j = y_j - (y_j, v_{k-1})v_{k-1}$
  $v_k = y_k / \|y_k\|_2$

In order to overlap the communication costs of the inner products, we split the j-loop into two parts. Then for each k we proceed as follows.

1. compute in parallel the local parts of the inner products for the first group
2. assemble the local inner products to global inner products
3. compute in parallel the local parts of the inner products for the second group
4. update y_k; compute the local inner products required for $\|y_k\|_2$
5. assemble the local inner products of the second group to global inner products
6. update the vectors $y_{k+1}, ..., y_{m+1}$
7. compute $v_k = y_k / \|y_k\|_2$

From this scheme it is obvious that if the length of the vector segments per processor are not too small, in principle all communication time can be overlapped by useful computations.

For a 150 processor MEIKO system, configured as a 15×10 torus, de Sturler [80] reports speedups of about 120 for typical discretized PDE systems with $60,000$ unknowns (i.e. 400 unknowns per processor). For larger systems, the speedup increases to 150 (or more if more processors are involved) as expected. Calvetti *et al.* [112] report results for an implementation of m-step GMRES, using BLAS2 Householder orthogonalization, for a four-processor IBM 6000 distributed memory system. For larger linear systems, they observed speedups close to 2.5.

10 Error Bounds for Solving $Ax = b$

How accurately can we expect to solve $Ax = b$? In this section we will outline the answer to this question, and refer to standard textbooks for details [4, 5, 6]. The answer depends both on the matrix A, and on the algorithm we use. To derive a useful error bound, the property of the matrix we need to know is its *condition number* $\kappa(A)$, and the property we require of the algorithm is *numerical stability.*

Library software also exists for evaluating all the error bounds described here, for dense and band matrices. We list the LAPACK routines for computing these error bounds below.

To define the condition number $\kappa(A)$ of A, we need to introduce some more notation. Let $\hat{x}$ denote solution of the linear system $(A+E)\hat{x} = b$, where E is a small perturbation of A. We want to measure the difference between $\hat{x} = (A+E)^{-1}b$ and $x = A^{-1}b$. Let $\|x\|_\infty \equiv \max_i |x_i|$; this is called the *infinity-norm of* x. We will derive a bound on

$$\frac{\|\hat{x} - x\|_\infty}{\|\hat{x}\|_\infty} \quad .$$

When this quantity is significantly less than 1, we say that $\hat{x}$ is a good approximation to x. For example, if it is 10^{-6}, we say $\hat{x}$ and x agree to six decimal places (note that only the largest components of $\hat{x}$ and x agree to six decimal places; the smaller components may agree to fewer). Supposing that X is an n-by-n matrix, define

$$\|X\|_\infty \equiv \max_{1 \le i \le n} \sum_{1 \le j \le n} |x_{ij}| \quad .$$

$\|X\|_\infty$ is called the *infinity-norm* of the matrix X. Note that $\|X\|_\infty$ is always within a factor of n of the absolute value of the largest entry of X. Using this notation, we can now define the condition number of a matrix as

$$\kappa(A) \equiv \|A\|_\infty \cdot \|A^{-1}\|_\infty \quad .$$

Using $\kappa(A)$, we can bound the difference between $\hat{x} = (A+E)^{-1}b$ and $x = A^{-1}b$ as follows:

$$\frac{\|x - \hat{x}\|_\infty}{\|\hat{x}\|_\infty} \le \kappa(A) \cdot \frac{\|E\|_\infty}{\|A\|_\infty} \quad .$$

In other words, to guarantee a small error we need $\kappa(A)$ not to be too large (one can show it is always at least 1), and for $\frac{\|E\|_\infty}{\|A\|_\infty}$ to be small. When $\kappa(A)$ is not large, we say A is *well conditioned*. If $\kappa(A)$ is large, we say A is *ill conditioned*. (The exact definition of these terms depends on the context, but their use is widespread.) $\kappa(A)$ depends only on the matrix A, not the algorithm we use; we will describe cheap ways to estimate it below.

The other property we need to guarantee a small error bound, $\frac{\|E\|_\infty}{\|A\|_\infty}$ being small, is called *numerical stability*, and is a property of the algorithm. A well designed algorithm will be numerically stable, and a poor algorithm will not be. More formally, we say an algorithm to solve $Ax = b$ is numerically stable if the solution $\hat{x}$ it computes satisfies $(A+E)\hat{x} = b$ (or equivalently $\hat{x} = (A+E)^{-1}b$), where $\frac{\|E\|_\infty}{\|A\|_\infty}$ is close to the machine precision ϵ. Recall from Chapter CA that the machine precision measures the roundoff error in individual floating point operations. In IEEE standard single precision floating point arithmetic, which has 23 bits to store the significant bits of a floating point number, $\epsilon = 2^{-24} \approx 10^{-7}$. In IEEE standard double precision floating point arithmetic, which has 52 bits to store the significant bits of a floating point number, $\epsilon = 2^{-53} \approx 10^{-16}$.

Gaussian elimination with partial pivoting is *almost always* numerically stable, so the error bound one expects from solving $Ax = b$ this way is

$$\frac{\|x - \hat{x}\|_\infty}{\|\hat{x}\|_\infty} \le \kappa(A) \cdot O(\epsilon) \quad .$$

The notation $O(\epsilon)$ means something close to ϵ. In practice, one sees errors as large as $\kappa(A) \cdot n\epsilon$ or so, where n is the dimension of A. So for example, if $\kappa(A) \approx 10^4$ and we compute in IEEE standard single precision, we can expect to get 3 correct decimal places in the answer, since $\kappa(A)\epsilon \approx 10^{-3}$.

We can only say that Gaussian elimination with partial pivoting is almost always numerically stable, because matrices A do exist where $\frac{\|E\|_\infty}{\|A\|_\infty}$ is as large as $2^n\epsilon$. Since 2^n grows very quickly with n, the error bound rapidly becomes enormous (and the actual solution also becomes poor). These examples are found in numerical analysis textbooks [4, 5, 6], but not in practice.

Rather than formally proving that Gaussian elimination with partial pivoting is generally stable, let us instead illustrate how omitting pivoting can destroy stability. Let us apply Gaussian elimination without pivoting to compute the factorization $A = LU$ of

$$A = \begin{bmatrix} .0001 & 1 \\ 1 & 1 \end{bmatrix}. \tag{3}$$

To keep the example simple, we will use 3 decimal digit floating point arithmetic. Note that $\kappa(A) = \|A\|_\infty \cdot \|A^{-1}\|_\infty \approx 4$, so A is well conditioned and we should expect to be able to solve $Ax = b$ accurately. We will use the notation $\mathrm{fl}(a\ op\ b)$ to mean the rounded, floating point result of $a\ op\ b$, where op is one of the operations $+$, $-$, $\times$ and $/$.

$$\begin{aligned} L &= \begin{bmatrix} 1 & 0 \\ \mathrm{fl}(1/10^{-4}) & 1 \end{bmatrix}, \quad \mathrm{fl}(1/10^{-4}) \text{ rounds to } 10^4 \text{ without error} \\ U &= \begin{bmatrix} 10^{-4} & 1 \\ & \mathrm{fl}(1 - 10^4 \cdot 1) \end{bmatrix}, \quad \mathrm{fl}(1 - 10^4 \cdot 1) \text{ rounds to } -10^4 \\ \text{so } LU &= \begin{bmatrix} 1 & 0 \\ 10^4 & 1 \end{bmatrix} \begin{bmatrix} 10^{-4} & 1 \\ & -10^4 \end{bmatrix} = \begin{bmatrix} 10^{-4} & 1 \\ 1 & 0 \end{bmatrix} \end{aligned}$$

Note that the original a_{22} has been entirely "lost" from the computation by subtracting 10^4 from it. We would have gotten the same LU factors whether a_{22} had been 1, 0, -2, or any number such that $\mathrm{fl}(a_{22} - 10^4) = -10^4$. Since the algorithm proceeds to work only with L and U, it will get the same answer for all these different a_{22}, which correspond to completely different A and so completely different $x = A^{-1}b$; there is no way to guarantee an accurate answer. This is called *numerical instability*, since L and U are *not* the exact factors of a matrix $A + E$ close to A (another way to say this is that $\|E\|_\infty = \|A - LU\|_\infty$ is about as large as $\|A\|_\infty$).

Let us see what happens when we go on to solve $Ax = [1, 2]^T$ for x using this LU factorization. The correct answer is $x \approx [1, 1]^T$. Instead we get the following. Solving $Ly = [1, 2]^T$ yields $y_1 = \mathrm{fl}(1/1) = 1$ and $y_2 = \mathrm{fl}(2 - 10^4 \cdot 1) = -10^4$; note that the value 2 has been "lost" by subtracting 10^4 from it. Solving $U\hat{x} = y$ yields $\hat{x}_2 = \mathrm{fl}((-10^4)/(-10^4)) = 1$ and $\hat{x}_1 = \mathrm{fl}((1 - 1)/10^{-4}) = 0$, a completely erroneous solution.

The intuition behind this is that if intermediate quantities in computing the product $L \cdot U$ are very large compared to $\|A\|_\infty$, then this can lead to loss of accuracy in the factorization of A, because large entries of L and U get added to small entries of A, and these small entries of A get rounded away. If the intermediate quantities in the product $L \cdot U$ were comparable to those of A, however, we would expect a tiny error $E = A - LU$ in the factorization. This is what pivoting tries to guarantee.

No matter what algorithm is used to solve $Ax = b$, one can compute $\frac{\|E\|_\infty}{\|A\|_\infty}$ very cheaply given the approximate solution $\hat{x}$. Just use the following formula

$$\frac{\|E\|_\infty}{\|A\|_\infty} = \frac{\|A\hat{x} - b\|_\infty}{\|A\|_\infty \cdot \|\hat{x}\|_\infty} \quad .$$

In the case of dense matrices, this formula costs just $O(n^2)$ flops to evaluate, must less than the $2n^3/3$ required by Gaussian elimination. With many iterative methods, $A\hat{x} - b$ is available anyway. To summarize, one can bound the error in the computed solution $\hat{x}$ by

$$\frac{\|x - \hat{x}\|_\infty}{\|\hat{x}\|_\infty} \leq \kappa(A) \cdot \frac{\|A\hat{x} - b\|_\infty}{\|A\|_\infty \|\hat{x}\|_\infty} \quad .$$

It remains to describe how to estimate the condition number $\kappa(A) = \|A\|_\infty \cdot \|A^{-1}\|_\infty$. $\|A\|_\infty$ is easy to compute from its definition, so we concentrate on $\|A^{-1}\|_\infty$. An obvious approach is to compute A^{-1} explicitly, and then compute its infinity norm. This approach would cost twice again as much as solving $Ax = b$ in the first place, and so is not done. Instead, we settle for approximations to $\|A^{-1}\|_\infty$ which can be computed very cheaply once the factorization $PA = LU$ has been produced by Gaussian elimination. In fact, one can usually estimate $\|A^{-1}\|_\infty$ to within a factor of two with just $O(n^2)$ work, given $PA = LU$. Since computing $PA = LU$ costs $2n^3/3$ using Gaussian elimination, estimating the condition number is a negligible and worthwhile extra cost. Note that computing the condition number to within a factor of two is more than accurate enough for an error bound; indeed, an order-of-magnitude estimate is enough to say how many correct decimal places are in the answer. The algorithm for estimating $\|A^{-1}\|_\infty$ is described in [113, 114, 115], and implemented in LAPACK. The routines which compute error bounds have the same names as above, but with an 'x' appended. For example, the routine that solves $Ax = b$ for general, dense A and computes error bounds is called `sgesvx`.

See exercise 10.

---

11 Using Netlib to Retrieve Software

A large body of numerical software is freely available 24 hours a day via an electronic service called *Netlib*. In addition to LAPACK and LINPACK, there are dozens of other

libraries, technical reports on various parallel computers and software, test data, facilities to automatically translate Fortran programs to C, bibliographies, names and addresses of scientists and mathematicians, and so on. One can communicate with Netlib in one of two ways, by email or (much more easily) via an X-window interface called *Xnetlib*. Using email, one sends messages of the form 'send subroutine_name from library_name' or 'send index for library_name' to the address 'netlib@ornl.gov' or 'netlib@research.att.com'. The message will be automatically read and the corresponding subroutine mailed back. Xnetlib (which can be obtained and installed by sending the message 'send xnetlib.shar from xnetlib' to netlib@ornl.gov) is an X-window interface which lets one point at and click on subroutines, which are then automatically transferred back into the user's directory. There are also index search features to help find the appropriate subroutine.

To get started using netlib, send the one-line message 'send index' to netlib@ornl.gov. A description of the overall library should be sent to you within minutes (providing all the intervening networks as well as netlib server are up).

Here is a brief summary of the contents of netlib. See the index netlib sends you for more details.

Contents of Netlib

Library	Short Description
a	approximation algorithms
alliant	set of programs collected from Alliant users
amos	special functions by D. Amos. = toms/644
apollo	set of programs collected from Apollo users
benchmark	various benchmark programs and a summary of timings
bib	bibliographies
bihar	Bjorstad's biharmonic solver
blas	machine constants, vector and matrix * vector BLAS
bmp	Brent's multiple precision package
c	another "misc" library, for software written in C
cheney	kincaid - programs from the 1985 text
confdb	conference database
conformal	conformal mapping
contin	continuation, limit points
c++	code in the C++ language
dierckx	Spline fitting on various geometries.
domino	communication and scheduling of multiple tasks; Univ. Maryland
eispack	matrix eigenvalues and vectors
elefunt	Cody and Waite's tests for elementary functions
errata	corrections to numerical books
f2c	Fortran to C converter
fishpack	separable elliptic PDEs; Swarztrauber and Sweet
fitpack	Cline's splines under tension
fftpack	Swarztrauber's Fourier transforms
fmm	software from the book by Forsythe, Malcolm, and Moler
fn	Fullerton's special functions
fortran	single-double precision converter, static debugger
fp	floating point arithmetic
gcv	Generalized Cross Validation
gmat	multi-processing Time Line and State Graph tools, Mark Seager
go	"golden oldies" gaussq, zeroin, lowess, ...
graphics	auto color, ray-tracing benchmark
harwell	MA28 sparse linear system
hence	Heterogeneous Network Computing Environment
hompack	nonlinear equations by homotopy method

Contents of Netlib (continued)

Library	Short Description
ieeecss	IEEE / Control Systems Society
itpack	iterative linear system solution by Young and Kincaid
jakef	automatic differentiation of Fortran subroutines
jgraph	tool to create Postscript graphs
kincaid	cheney - programs from the 1990 text
lapack	solving the most common problems in numerical linear algebra
lanczos	Cullum and Willoughby's Lanczos programs
lanz	Large Sparse Symmetric Generalized Eigenproblem, Jones and Patrick
laso	Scott's Lanczos program for eigenvalues of sparse matrices
linpack	Gaussian elimination, QR, SVD by Dongarra, Bunch, Moler, Stewart
lp	linear programming
machines	short descriptions of various computers
matlab	software from the MATLAB user's group
microscope	Alfeld and Harris' system for discontinuity checking
minpack	nonlinear equations and least squares by More, Garbow, Hillstrom
misc	everything else
ml	Standard ML of New Jersey (programming language compiler)
mpi	message passing interface (draft specifications)
na-digest	archive of mailings to NA distribution list
napack	numerical algebra programs
news	Grosse's Netlib News column for na-net, SIAM News, SIGNUM Newsletter
numeralgo	algorithms from the new journal "Numerical Algorithms"
ode	ordinary differential equations
odepack	ordinary differential equations from Hindmarsh
odrpack	orthogonal distance regression, Boggs Byrd Donaldson Schnabel
opt	optimization
p4	portable programs for parallel processors
paragraph	display of algorithms on message-passing multiprocessor
paranoia	Kahan's floating point test
parmacs	parallel programming macros
pascal	another "misc" library, for software written in Pascal
pbwg	parallel benchmark working group
pchip	Hermite cubics Fritsch+Carlson
pdes/madpack	a multigrid package, by Craig Douglas
picl	portable instrumented communication library for multiprocessors

Contents of Netlib (continued)

Library	Short Description
pltmg	Bank's multigrid code; too large for ordinary mail
polyhedra	Hume's database of geometric solids
popi	Digital Darkroom image manipulation software (Holzmann)
port	the public subset of PORT library
posix	draft standards
pppack	subroutines from de Boor's Practical Guide to Splines
pvm	parallel virtual machine
pvm3	parallel virtual machine version 3
quadpack	univariate quadrature by Piessens, de Donker, Kahaner
research	miscellanea from AT&T Bell Labs, Computing Science Research Center
sched	environment for portable parallel algorithms in a Fortran setting.
sciport	portable version of Cray SCILIB, by McBride and Lamson
sequent	software from the Sequent Users Group
slap	Seager + Greenbaum, iterative methods for symmetric and unsymmetric
slatec	comprehensive mathematical and statistical software package
sparse	Kundert + Sangiovanni-Vincentelli, C sparse linear algebra
sparse	blas - BLAS by indirection
sparspak	George + Liu, sparse linear algebra core
specfun	transportable special functions
spin	simulation and validation of communication protocols, G. Holzmann
stringsearch	string matching
toeplitz	linear systems in Toeplitz or circulant form by Garbow
toms	Collected Algorithms of the ACM
typesetting	typesetting macros and preprocessors
uncon/data	optimization test problems
vanhuffel	total least squares, partial SVD by Van Hufell
vfftpk	vectorized FFT; variant of fftpack
voronoi	Voronoi diagrams and Delaunay triangulations
xnetlib	X windows interface to netlib
y12m	sparse linear system (Aarhus)

12 Quick Reference Guide to the BLAS

Level 1 BLAS

```
                  dim scalar vector   vector   scalars             5-element array
SUBROUTINE _ROTG (                                    A, B, C, S )
SUBROUTINE _ROTMG(                                D1, D2, A, B,     PARAM )
SUBROUTINE _ROT  ( N,         X, INCX, Y, INCY,                C, S )
SUBROUTINE _ROTM ( N,         X, INCX, Y, INCY,                   PARAM )
SUBROUTINE _SWAP ( N,         X, INCX, Y, INCY )
SUBROUTINE _SCAL ( N,  ALPHA, X, INCX )
SUBROUTINE _COPY ( N,         X, INCX, Y, INCY )
SUBROUTINE _AXPY ( N,  ALPHA, X, INCX, Y, INCY )
FUNCTION   _DOT  ( N,         X, INCX, Y, INCY )
FUNCTION   _DOTU ( N,         X, INCX, Y, INCY )
FUNCTION   _DOTC ( N,         X, INCX, Y, INCY )
FUNCTION   __DOT ( N,         X, INCX, Y, INCY )
FUNCTION   _NRM2 ( N,         X, INCX )
FUNCTION   _ASUM ( N,         X, INCX )
FUNCTION   I_AMAX( N,         X, INCX )
```

Level 2 BLAS

```
       options            dim   b-width scalar matrix  vector   scalar vector
_GEMV (        TRANS,     M, N,         ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )
_GBMV (        TRANS,     M, N, KL, KU, ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )
_HEMV ( UPLO,                N,         ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )
_HBMV ( UPLO,                N, K,      ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )
_HPMV ( UPLO,                N,         ALPHA, AP,     X, INCX, BETA,  Y, INCY )
_SYMV ( UPLO,                N,         ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )
_SBMV ( UPLO,                N, K,      ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )
_SPMV ( UPLO,                N,         ALPHA, AP,     X, INCX, BETA,  Y, INCY )
_TRMV ( UPLO, TRANS, DIAG,   N,                A, LDA, X, INCX )
_TBMV ( UPLO, TRANS, DIAG,   N, K,             A, LDA, X, INCX )
_TPMV ( UPLO, TRANS, DIAG,   N,                AP,     X, INCX )
_TRSV ( UPLO, TRANS, DIAG,   N,                A, LDA, X, INCX )
_TBSV ( UPLO, TRANS, DIAG,   N, K,             A, LDA, X, INCX )
_TPSV ( UPLO, TRANS, DIAG,   N,                AP,     X, INCX )

       options            dim   scalar vector   vector   matrix
_GER  (                   M, N, ALPHA, X, INCX, Y, INCY, A, LDA )
_GERU (                   M, N, ALPHA, X, INCX, Y, INCY, A, LDA )
_GERC (                   M, N, ALPHA, X, INCX, Y, INCY, A, LDA )
_HER  ( UPLO,                N, ALPHA, X, INCX,          A, LDA )
_HPR  ( UPLO,                N, ALPHA, X, INCX,          AP )
_HER2 ( UPLO,                N, ALPHA, X, INCX, Y, INCY, A, LDA )
_HPR2 ( UPLO,                N, ALPHA, X, INCX, Y, INCY, AP )
_SYR  ( UPLO,                N, ALPHA, X, INCX,          A, LDA )
_SPR  ( UPLO,                N, ALPHA, X, INCX,          AP )
_SYR2 ( UPLO,                N, ALPHA, X, INCX, Y, INCY, A, LDA )
_SPR2 ( UPLO,                N, ALPHA, X, INCX, Y, INCY, AP )
```

Level 3 BLAS

```
       options                        dim      scalar matrix  matrix  scalar matrix
_GEMM (             TRANSA, TRANSB,   M, N, K, ALPHA, A, LDA, B, LDB, BETA,  C, LDC )
_SYMM ( SIDE, UPLO,                   M, N,    ALPHA, A, LDA, B, LDB, BETA,  C, LDC )
_HEMM ( SIDE, UPLO,                   M, N,    ALPHA, A, LDA, B, LDB, BETA,  C, LDC )
_SYRK (       UPLO, TRANS,               N, K, ALPHA, A, LDA,         BETA,  C, LDC )
_HERK (       UPLO, TRANS,               N, K, ALPHA, A, LDA,         BETA,  C, LDC )
_SYR2K(       UPLO, TRANS,               N, K, ALPHA, A, LDA, B, LDB, BETA,  C, LDC )
_HER2K(       UPLO, TRANS,               N, K, ALPHA, A, LDA, B, LDB, BETA,  C, LDC )
_TRMM ( SIDE, UPLO, TRANSA,       DIAG, M, N,    ALPHA, A, LDA, B, LDB )
_TRSM ( SIDE, UPLO, TRANSA,       DIAG, M, N,    ALPHA, A, LDA, B, LDB )
```

	prefixes
Generate plane rotation	S, D
Generate modified plane rotation	S, D
Apply plane rotation	S, D
Apply modified plane rotation	S, D
$x \leftrightarrow y$	S, D, C, Z
$x \leftarrow \alpha x$	S, D, C, Z, CS, ZD
$y \leftarrow x$	S, D, C, Z
$y \leftarrow \alpha x + y$	S, D, C, Z
$dot \leftarrow x^T y$	S, D, DS
$dot \leftarrow x^T y$	C, Z
$dot \leftarrow x^H y$	C, Z
$dot \leftarrow \alpha + x^T y$	SDS
$nrm2 \leftarrow \|\|x\|\|_2$	S, D, SC, DZ
$asum \leftarrow \|\|re(x)\|\|_1 + \|\|im(x)\|\|_1$	S, D, SC, DZ
$amax \leftarrow 1^{st} k \ni \lvert re(x_k)\rvert + \lvert im(x_k)\rvert = max(\lvert re(x_i)\rvert + \lvert im(x_i)\rvert)$	S, D, C, Z

$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
$y \leftarrow \alpha Ax + \beta y$	C, Z
$y \leftarrow \alpha Ax + \beta y$	C, Z
$y \leftarrow \alpha Ax + \beta y$	C, Z
$y \leftarrow \alpha Ax + \beta y$	S, D
$y \leftarrow \alpha Ax + \beta y$	S, D
$y \leftarrow \alpha Ax + \beta y$	S, D
$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
$x \leftarrow A^{-1} x, x \leftarrow A^{-T} x, x \leftarrow A^{-H} x$	S, D, C, Z
$x \leftarrow A^{-1} x, x \leftarrow A^{-T} x, x \leftarrow A^{-H} x$	S, D, C, Z
$x \leftarrow A^{-1} x, x \leftarrow A^{-T} x, x \leftarrow A^{-H} x$	S, D, C, Z

$A \leftarrow \alpha xy^T + A, A - m \times n$	S, D
$A \leftarrow \alpha xy^T + A, A - m \times n$	C, Z
$A \leftarrow \alpha xy^H + A, A - m \times n$	C, Z
$A \leftarrow \alpha xx^H + A$	C, Z
$A \leftarrow \alpha xx^H + A$	C, Z
$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
$A \leftarrow \alpha xx^T + A$	S, D
$A \leftarrow \alpha xx^T + A$	S, D
$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D
$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D

$C \leftarrow \alpha op(A) op(B) + \beta C, op(X) = X, X^T, X^H, C - m \times n$	S, D, C, Z
$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^H$	C, Z
$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^T$	S, D, C, Z
$C \leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C - n \times n$	C, Z
$C \leftarrow \alpha AA^T + \beta C, C \leftarrow \alpha A^T A + \beta C, C - n \times n$	S, D, C, Z
$C \leftarrow \alpha AB^H + \alpha BA^H + \beta C, C \leftarrow \alpha A^H B + \alpha B^H A + \beta C, C - n \times n$	C, Z
$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C, C \leftarrow \alpha A^T B + \alpha B^T A + \beta C, C - n \times n$	S, D, C, Z
$B \leftarrow \alpha op(A) B, B \leftarrow \alpha B op(A), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z
$B \leftarrow \alpha op(A^{-1}) B, B \leftarrow \alpha B op(A^{-1}), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z

Notes

Meaning of prefixes

S - REAL	C - COMPLEX	
D - DOUBLE PRECISION	Z - COMPLEX*16	(this may not be supported by all machines)

For the Level 2 BLAS a set of extended-precision routines with the prefixes ES, ED, EC, EZ may also be available.

Level 1 BLAS

In addition to the listed routines there are two further extended-precision dot product routines DQDOTI and DQDOTA.

Level 2 and Level 3 BLAS

Matrix types

GE - GEneral	GB - General Band	
SY - SYmmetric	SB - Symmetric Band	SP - Symmetric Packed
HE - HErmitian	HB - Hermitian Band	HP - Hermitian Packed
TR - TRiangular	TB - Triangular Band	TP - Triangular Packed

Options

The dummy options arguments are declared as CHARACTER*1 and may be passed as character strings.

TRAN_	= '**N**o transpose', '**T**ranspose', '**C**onjugate transpose' (X, X^T, X^H)
UPLO	= '**U**pper triangular', '**L**ower triangular'
DIAG	= '**N**on-unit triangular', '**U**nit triangular'
SIDE	= '**L**eft', '**R**ight' (A or op(A) on the left, or A or op(A) on the right)

For real matrices, TRANS_ = 'T' and TRANS_ = 'C' have the same meaning.
For Hermitian matrices, TRANS_ = 'T' is not allowed.
For complex symmetric matrices, TRANS_ = 'H' is not allowed.

13 Exercises

Exercise 1 *Confirm the values of q computed for the above three algorithms.*

On the other hand, this brief analysis ignores a number of practical issues:

(1) high level language constructs do not yet support block layout of matrices as described here,

(2) if the fast memory consists of vector registers and has vector operations supporting `saxpy` but not inner products, a column blocked code may be superior;

(3) a real code will have to deal with nonsquare matrices, for which the optimal block sizes may not be square.

This is why computer manufacturers are often in the best position to write the matrix-multiplication (and other BLAS) for their machines, because they often have the best understanding of these machine specific details.[2]

Another quite different algorithm is Strassen's method [17], which multiplies matrices recursively by dividing them into 2×2 block matrices, and multiplying the subblocks using 7 matrix multiplications (recursively) and 15 matrix additions of half the size. This leads to an asymptotic complexity of $n^{\log_2 7} \approx n^{2.81}$ instead of n^3. The value of this algorithm is not just this asymptotic complexity but its reduction of the problem to smaller subproblems which eventually fit in fast memory; once the subproblems fit in fast memory standard matrix multiplication may be used. This approach has led to speedups on relatively large matrices on some machines [18]. A drawback is the need for significant workspace, and somewhat lower numerical stability, although it is adequate for many purposes [19].

Exercise 2 *A comparison of blocked implementations for the basic linear algebra subroutine and the matrix-multiplication implemention, previously provided.*

Consider the basic linear algebra subroutine which solves $TX = B$ for X, where T is a given n-by-n triangular matrix, B is a given n-by-m matrix, and X is an n-by-m matrix of unknowns. Give blocked implementations for this subroutine analogous to the ones for matrix-multiplication above, and compare their ratios q of flops to memory references. How do your answers depend on m?

Exercise 3 *Implement Cannon's algorithm on a serial machine in your favorite programming language and confirm that it works.*

Exercise 4 *Confirm this timing analysis for Cannon's method.*

Exercise 5 *A consideration of the cost involved in an algorithm and a slight variation of the algorithm.*

Show how to reduce the number of messages sent in the first two loops of the algorithm by sometimes shifting right (down) instead of left (up). How does this reduce the overall cost?

Exercise 6 *The algorithm derivation to return two matrices to their original state.*

When Cannon's algorithm completes, A and B are not in their original storage locations. Write an algorithm to return A and B to their original positions. What is the running time of your algorithm?

[2] The matrix-multiplication subroutine in the CM-2 Scientific Subroutine Library took approximately 10 person-years of effort to write [16].

Exercise 7 *Illustrate the block scattered layout of a* 50×60 *matrix on a* 4×8 *processor grid with* 5×5 *blocks.*

By being a little more flexible about the algorithms we implement, we can mitigate the apparent tradeoff between load balance and applicability of BLAS. For example, the layout of A in Figure 3 is identical to the layout in Figure 2 of $P^T AP$, where P is a permutation matrix. This shows that running Cannon's algorithm from the last section to multiply A times B in scatter layout is the same as multiplying PAP^T and PBP^T to get $PABP^T$, which is the desired product. Indeed, as long as (1) A and B are both distributed over a square array of processors; (2) the permutations of the columns of A and rows of B are identical; and (3) for all i the number of columns of A stored by processor column i is the same as the number of rows of B stored by processor row i, the algorithms of the previous section will correctly multiply A and B. The distribution of the product will be determined by the distribution of the rows of A and columns of B. We will see a similar phenomenon for other distributed memory algorithms later.

Exercise 8 *Verication of Cannon's algorithm when the matrices are stored in a scattered manner.*

Using your implementation of Cannon's algorithm from an earlier exercise, confirm by running it that it multiplies matrices correctly even when they are stored in a scattered layout.

Exercise 9 *An illustration of an analog of algorithm for Cholesky, and the derivation of other analogs for some algorithms.*

The *Cholesky factorization* of a symmetric positive definite matrix A is $A = LL^T$, where L is a lower triangular matrix. The algorithm is very similar to Gaussian elimination, but the special properties of A mean only half as much storage and half as many flops are needed as for standard Gaussian elimination. Here is the analog of Algorithm 6.1 for Cholesky:

Algorithm 13.1 *Row oriented Cholesky (access lower triangle of A only)*

for $k = 1 : n$
  for $i = k : n$
    for $j = 1 : k - 1$
      $A(i,k) = A(i,k) - A(i,j) \cdot A(k,j)$
  Exit if $A(k,k) \leq 0$
  $A(k,k) = \sqrt{A(k,k)}$
  for $i = k : n$
    $A(i,k) = A(i,k)/A(k,k)$

Note that Cholesky does not require pivoting. Derive analogs of Algorithms 6.2 through 6.4 for Cholesky. Ideally your algorithm should only need to read and write the lower (or upper) triangular part of A.

Exercise 10 *An illustration of the use of a netlib routine, the modification, and the consideration of the errors.*

Retrieve the LAPACK routine `sgeevx` from netlib (see section 11 below on how to do this). Write a modified version, which we will call `sgesvn`, which does *not* do pivoting. Run both routines on randomly generated test matrices, and compare the error bounds they compute. Modify the test matrices to make the (1,1) entry very small. How do the error bounds compare now? Note that LAPACK produces other error bounds besides the one described here.

References

[1] J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart, *LINPACK User's Guide* (SIAM, Philadelphia, PA, 1979).

[2] E. Anderson *et al.*, *LAPACK Users' Guide, Release 1.0* (SIAM, Philadelphia, 1992), 235 pages.

[3] J. Demmel, M. Heath, and H. van der Vorst, in *Acta Numerica, volume 2*, edited by A. Iserles (Cambridge University Press, ADDRESS, 1993).

[4] G. Golub and C. Van Loan, *Matrix Computations*, 2nd ed. (Johns Hopkins University Press, Baltimore, MD, 1989).

[5] D. Watkins, *Fundamentals of Matrix Computations* (Wiley, ADDRESS, 1991).

[6] J. Demmel, Berkeley Lecture Notes in Numerical Linear Algebra, Mathematics Department, University of California, 1993, 215 pages.

[7] J. Dongarra and E. Grosse, Communications of the ACM **30**, 403 (1987).

[8] J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst, *Solving linear systems on vector and shared memory computers* (SIAM, Philadelphia, PA, 1991), 256 pages.

[9] K. Gallivan *et al.*, *Parallel algorithms for matrix computations* (SIAM, Philadelphia, PA, 1990).

[10] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, ACM Trans. Math. Soft. **5**, 308 (1979).

[11] J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, ACM Trans. Math. Soft. **14**, 1 (1988).

[12] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, ACM Trans. Math. Soft. **16**, 1 (1990).

[13] B. T. Smith *et al.*, *Matrix Eigensystem Routines – EISPACK Guide*, Vol. 6 of *Lecture Notes in Computer Science* (Springer-Verlag, Berlin, 1976).

[14] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler, *Matrix Eigensystem Routines – EISPACK Guide Extension*, Vol. 51 of *Lecture Notes in Computer Science* (Springer-Verlag, Berlin, 1977).

[15] X. Hong and H. T. Kung, in *Proceedings of the 13th Symposium on the Theory of Computing* (ACM, ADDRESS, 1981), pp. 326–334.

[16] S. L. Johnsson, private communication, 1990.

[17] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms* (Computer Science Press, Potomac, MD, 1978).

[18] D. H. Bailey, K. Lee, and H. D. Simon, J. Supercomputing **4**, 97 (1991).

[19] J. Demmel and N. J. Higham, ACM Trans. Math. Soft. **18**, 274 (1992).

[20] L. Cannon, Ph.D. thesis, Montana State University, Bozeman, MN, 1969.

[21] G. Fox *et al.*, *Solving problems on concurrent processors, v. I* (Prentice Hall, ADDRESS, 1988).

[22] C. T. Ho, Ph.D. thesis, Yale University, 1990.

[23] S. L. Johnsson, J. of Parallel and Distributed Computing **4**, 133 (1987).

[24] E. Dekel, D. Nassimi, and S. Sahni, SIAM J. Comput. **10**, 657 (1981).

[25] C. T. Ho, S. L. Johnsson, and A. Edelman, in *The Sixth Distributed Memory Computing Conference Proceedings* (IEEE Computer Society Press, ADDRESS, 1991), pp. 447–451.

[26] G. Fox *et al.*, Computer Science Department Report CRPC-TR90079, Rice University, Houston, TX (unpublished).

[27] High Performance Fortran, documentation available via anonymous ftp from titan.cs.rice.edu in directory public/HPFF, 1991.

[28] E. Van de Velde, caltech, Pasadena, CA (unpublished).

[29] E. Anderson and J. Dongarra, Computer Science Dept. Technical Report CS-90-103, University of Tennessee, Knoxville (unpublished), (LAPACK Working Note #19).

[30] Y. Robert, *The impact of vector and parallel architectures on the Gaussian elimination algorithm* (Wiley, ADDRESS, 1990).

[31] J. Dongarra and S. Ostrouchov, Computer Science Dept. Technical Report CS-90-115, University of Tennessee, Knoxville (unpublished), (LAPACK Working Note #24).

[32] J. Dongarra and R. van de Geijn, Computer Science Dept. Technical Report CS-91-138, University of Tennessee, Knoxville (unpublished), (LAPACK Working Note #37).

[33] S. Eisenstat, M. Heath, C. Henkel, and C. Romine, SIAM J. Sci. Stat. Comput. **9**, 589 (1988).

[34] M. Heath and C. Romine, SIAM J. Sci. Stat. Comput. **9**, 558 (1988).

[35] G. Li and T. Coleman, SIAM J. Sci. Stat. Comput. **9**, 485 (1988).

[36] C. Romine and J. Ortega, Parallel Computing **6**, 109 (1988).

[37] J. Du Croz, P. J. D. Mayes, and G. Radicati di Brozolo, Computer Science Dept. Technical Report CS-90-109, University of Tennessee, Knoxville (unpublished), (LAPACK Working Note #21.

[38] J. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems* (Plenum Press, ADDRESS, 1988).

[39] I. Babuska, SIAM J. Numer. Anal. **9**, 53 (1972).

[40] H. A. van der Vorst, Parallel Computing **5**, 303 (1987).

[41] H. Stone, J. Assoc. Comput. Mach. **20**, 27 (1973).

[42] P. Dubois and G. Rodrigue, in *High speed computer and algorithm organization*, edited by D. J. Kuck and A. H. Sameh (Academic Press, New York, 1977).

[43] J. J. Lambiotte and R. G. Voigt, Technical report, ICASE-NASA Langley Research Center, Hampton, VA (unpublished).

[44] D. Heller, SIAM J. Numer. Anal. **13**, 484 (1978).

[45] P. P. N. de Groen, Appl. Num. Math. **8**, 117 (1991).

[46] J. J. F. M. Schlichting and H. A. van der Vorst, Technical Report No. NM-R8725, CWI, Amsterdam, the Netherlands (unpublished).

[47] S. C. Chen, D. J. Kuck, and A. H. Sameh, ACM Trans. Math. Software **4**, 270 (1978).

[48] H. H. Wang, ACM Trans. Math. Software **7**, 170 (1981).

[49] H. A. van der Vorst and K. Dekker, SIAM J. Sci. Stat. Comput. **10**, 27 (1989).

[50] P. H. Michielse and H. A. van der Vorst, Parallel Computing **7**, 87 (1988).

[51] H. A. van der Vorst, Future Generation Computer Systems **4**, 285 (1989).

[52] U. Meier, Parallel Comput. **2**, 33 (1985).

[53] V. Mehrmann, Technical Report No. Bericht Nr. 68, Inst. fuer Geometrie und Prakt. Math., Aachen (unpublished).

[54] M. Heath, E. Ng, and B. Peyton, SIAM Review **33**, 420 (1991).

[55] I. Duff, A. Erisman, and J. Reid, *Direct Methods for Sparse Matrices* (Oxford University Press, Oxford, England, 1986).

[56] A. George and J. Liu, *Computer Solution of Large Sparse Positive Definite Systems* (Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981).

[57] J. Liu, SIAM J. Matrix Anal. Appl. **11**, 134 (1990).

[58] B. Irons, Internat. J. Numer. Meth. Engrg. **2**, 5 (1970).

[59] C. Ashcraft *et al.*, Internat. J. Supercomp. Appl **1**, 10 (1987).

[60] E. Rothberg and A. Gupta, Technical Report No. STAN-CS-89-1286, Stanford University, Stanford, California (unpublished).

[61] J. Liu, Parallel Computing **3**, 327 (1986).

[62] J. Jess and H. Kees, IEEE Trans. Comput. **C-31**, 231 (1982).

[63] J. Lewis, B. Peyton, and A. Pothen, SIAM J. Sci. Stat. Comput. **10**, 1156 (1989).

[64] J. Liu, Parallel Computing **11**, 73 (1989).

[65] A. George and J. Liu, SIAM Review **31**, 1 (1989).

[66] A. George, SIAM J. Num. Anal. **10**, 345 (1973).

[67] A. George, M. Heath, J. Liu, and E. Ng, Internat. J. Parallel Programming **15**, 309 (1986).

[68] A. George, M. Heath, J. Liu, and E. Ng, J. Comp. Appl. Math. **27**, 129 (1989).

[69] A. George, M. Heath, J. Liu, and E. Ng, SIAM J. Sci. Stat. Comput. **9**, 327 (1988).

[70] A. George, J. Liu, and E. Ng, Parallel Computing **10**, 287 (1989).

[71] M. Mu and J. Rice, SIAM J. Sci. Stat. Comput. **13**, 826 (1992).

[72] E. Zmijewski, Technical Report No. TRCS89-18, Department of Computer Science, University of California, Santa Barbara, CA (unpublished).

[73] C. Ashcraft, S. Eisenstat, and J. Liu, SIAM J. Sci. Stat. Comput. **11**, 593 (1990).

[74] R. Benner, G. Montry, and G. Weigand, Internat. J. Supercomp. Appl **1**, 26 (1987).

[75] I. Duff, Parallel Computing **3**, 193 (1986).

[76] J. Gilbert and R. Schreiber, SIAM J. Sci. Stat. Comput. **13**, 1151 (1992).

[77] R. Lucas, W. Blank, and J. Tieman, IEEE Trans. Computer Aided Design **CAD-6**, 981 (1987).

[78] C. Ashcraft, S. Eisenstat, J. Liu, and A. Sherman, Technical Report No. YALEU/DCS/RR-810, Dept. of Computer Science, Yale University, New Haven, CT (unpublished).

[79] R. Freund, G. Golub, and N. Nachtigal, in *Acta Numerica 1992*, edited by A. Iserles (Cambridge University Press, ADDRESS, 1992), pp. 57–100.

[80] E. de Sturler, Technical Report No. 91-85, Delft University of Technology, Delft (unpublished).

[81] C. Pommerell, Ph.D. thesis, Swiss Federal Institute of Technology, Zürich, 1992.

[82] J. A. Meijerink and H. A. van der Vorst, Math.Comp. **31**, 148 (1977).

[83] J. A. Meijerink and H. A. van der Vorst, J. Comp. Phys. **44**, 134 (1981).

[84] C. Ashcraft and R. Grimes, SIAM J. Sci. Stat. Comput. **9**, 122 (1988).

[85] H. A. van der Vorst, SIAM J. Sci. Statist. Comput. **10**, 1174 (1989).

[86] H. A. van der Vorst, in *Proc. of the fifth Int.Symp. on Numer. Methods in Eng.*, edited by R. Gruber, J. Periaux, and R. P. Shaw (PUBLISHER, ADDRESS, 1989), vol 1.

[87] J. J. F. M. Schlichting and H. A. van der Vorst, Journal of Comp. and Appl. Math. **27**, 323 (1989).

[88] I. S. Duff and G. A. Meurant, BIT **29**, 635 (1989).

[89] H. A. van der Vorst, J. Comp. and Appl. Math. **18**, 249 (1987).

[90] S. Doi, Appl. Num. Math. **7**, 417 (1991).

[91] M. K. Seager, Parallel Computing **3**, 35 (1986).

[92] G. Radicati di Brozolo and Y. Robert, Technical Report No. 681-M, IMAG/TIM3, Grenoble (unpublished).

[93] H. Berryman, J. Saltz, W. Gropp, and R. Mirchandaney, Technical Report No. 89-54, NASA Langley Research Center, ICASE, Hampton, VA (unpublished).

[94] Y. Saad, SIAM J. Sci. Stat. Comput. **6**, 865 (1985).

[95] N. K. Madsen, G. H. Rodrigue, and J. I. Karush, Inform. Process.Lett. **5**, 41 (1976).

[96] H. A. van der Vorst, SIAM J. Sci. Stat. Comput. **3**, 86 (1982).

[97] M. R. Hestenes and E. Stiefel, J. Res. Natl. Bur. Stand. **49**, 409 (1954).

[98] A. T. Chronopoulos and C. W. Gear, J. on Comp. and Appl. Math. **25**, 153 (1989).

[99] G. Meurant, Technical Report No. LBL-18023, University of California, Berkeley, CA (unpublished).

[100] G. Meurant, BIT **24**, 623 (1984).

[101] H. A. van der Vorst, Parallel Computing **3**, 49 (1986).

[102] A. T. Chronopoulos, in *Supercomputing '91* (IEEE Computer Society Press, Los Alamitos, CA, 1991), pp. 578–587.

[103] Y. Saad, Technical report, RIACS, Moffett Field, CA (unpublished).

[104] A. T. Chronopoulos and S. K. Kim, Technical Report No. 90/43R, UMSI, Minneapolis (unpublished).

[105] E. D'Azevedo and C. Romine, Technical Report ORNL/TM-12192, Oak Ridge National Laboratory (unpublished).

[106] V. Eijkhout, Compute Science Dept. Technical Report CS-92-170, University of Tennessee at Knoxville (unpublished).

[107] Y. Saad and M. H. Schultz, SIAM J. Sci. Statist. Comput. **7**, 856 (1986).

[108] Y. Saad and M. H. Schultz, Math. of Comp. **44**, 417 (1985).

[109] H. F. Walker, SIAM J. Sci. Stat. Comp. **9**, 152 (1988).

[110] H. A. van der Vorst and C. Vuik, Technical Report No. 91-80, Delft University of Technology, Faculty of Tech. Math. (unpublished).

[111] Z. Bai, D. Hu, and L. Reichel, Technical Report No. 91-03, University of Kentucky (unpublished).

[112] D. Calvetti, J. Petersen, and L. Reichel, Technical Report No. ICM-9110-6, Institute for Computational Mathematics, Kent, OH (unpublished).

[113] W. W. Hager, SIAM J. Sci. Stat. Comput. **5**, 311 (1984).

[114] N. J. Higham, SIAM Review **29**, 575 (1987).

[115] N. J. Higham, SIAM J. Sci. Stat. Comput. **11**, 804 (1990).