LAPACK Working Note 60, UT CS-93-192
Parallel numerical linear algebra

James W. Demmel*
Computer Science Division and Mathematics Department
University of California at Berkeley
Berkeley, CA 94720 USA
E-mail: demmel@cs.berkeley.edu

Michael T. Heath†
Department of Computer Science
and
National Center for Supercomputing Applications
University of Illinois
Urbana, IL 61801 USA
E-mail: heath@ncsa.uiuc.edu

Henk A. van der Vorst‡
Mathematical Institute
Utrecht University
Utrecht, The Netherlands
E-mail: vorst@math.ruu.nl

August 18, 1993

Abstract

We survey general techniques and open problems in numerical linear algebra on parallel architectures. We first discuss basic principles of parallel processing, describing the costs of basic operations on parallel machines, including general principles for constructing efficient algorithms. We illustrate these principles using current architectures and software systems, and by showing how one would implement matrix multiplication. Then, we present direct and iterative algorithms for solving linear systems of equations, linear least squares problems, the symmetric eigenvalue problem, the nonsymmetric eigenvalue problem, and the singular value decomposition. We consider dense, band and sparse matrices.

Contents

1 Introduction **4**

2 Features of parallel systems **5**
- 2.1 General principles 5
- 2.2 Examples 7
- 2.3 Important tradeoffs 13

3 Matrix multiplication **14**
- 3.1 Matrix multiplication on a shared memory machine 15
- 3.2 Matrix multiplication on a distributed memory machine 17
- 3.3 Data layouts on distributed memory machines 21

*The author was supported by NSF grant ASC-9005933, DARPA contract DAAL03-91-C-0047 via a subcontract from the University of Tennessee administered by ARO, and DARPA grant DM28E04120 via a subcontract from Argonne National Laboratory. This work was partially performed during a visit to the Institute for Mathematics and its Applications at the University of Minnesota.

†The author was supported by DARPA contract DAAL03-91-C-0047 via a subcontract from the University of Tennessee administered by ARO.

‡This work was supported in part by NCF/Cray Research University Grant CRG 92.03.

4 Systems of linear equations **24**
4.1 Gaussian elimination on a shared memory machine 24
4.2 Gaussian elimination on a distributed memory machine 27
4.3 Clever but impractical parallel algorithms for solving $Ax = b$ . . 28
4.4 Solving banded systems 29

5 Least squares problems **33**
5.1 Shared memory algorithms 33
5.2 Distributed memory algorithms 34

6 Eigenproblems and the singular value decomposition **35**
6.1 General comments 35
6.2 Reduction to condensed forms 36
6.3 The symmetric tridiagonal eigenproblem 37
6.3.1 QR Iteration 37
6.3.2 Bisection and inverse iteration 38
6.3.3 Cuppen's divide-and-conquer algorithm 39
6.4 Jacobi's method for the symmetric eigenproblem and SVD 41
6.5 The nonsymmetric eigenproblem 43
6.5.1 Hessenberg QR iteration 43
6.5.2 Reduction to nonsymmetric tridiagonal form 44
6.5.3 Jacobi's method 45
6.5.4 Hessenberg divide-and-conquer 45
6.5.5 Spectral divide-and-conquer 46

7 Direct methods for sparse linear systems **48**
7.1 Cholesky factorization 48
7.2 Sparse matrices 49
7.3 Sparse factorization 51
7.4 Parallelism in sparse factorization 54
7.5 Parallel algorithms for sparse factorization 58

8 Iterative methods for linear systems **62**
8.1 Parallelism in the kernels of iterative methods 63
8.2 Parallelism and data locality in preconditioned CG 65
8.3 Parallelism and data locality in GMRES 70

9 Iterative methods for eigenproblems **72**
9.1 The Lanczos method 73
9.2 The Arnoldi method 75

1 Introduction

Accurate and efficient algorithms for many problems in numerical linear algebra have existed for years on conventional serial machines, and there are many portable software libraries that implement them efficiently (Dongarra, Bunch, Moler and Stewart, 1979; Dongarra and Grosse, 1987; Garbow, Boyle, Dongarra and Moler, 1977; Smith, Boyle, Dongarra, Garbow, Ikebe, Klema and Moler, 1976). One reason for this profusion of successful software is the simplicity of the cost model: the execution time of an algorithm is roughly proportional to the number of floating point operations it performs. This simple fact makes it relatively easy to design efficient and portable algorithms. In particular, one need not worry excessively about the location of the operands in memory, nor the order in which the operations are performed. That we can use this approximation is a consequence of the progress from drum memories to semiconductor cache memories, software to hardware floating point, assembly language to optimizing compilers, and so on. Programmers of current serial machines can ignore many details earlier programmers could ignore only at the risk of significantly slower programs.

With modern parallel computers we have come full circle and again need to worry about details of data transfer time between memory and processors, and which numerical operations are most efficient. Innovation is very rapid, with new hardware architectures and software models being proposed and implemented constantly. Currently one must immerse oneself in the multitudinous and often ephemeral details of these systems in order to write reasonably efficient programs. Perhaps not surprisingly, a number of techniques for dealing with data transfer in blocked fashion in the 1960s are being rediscovered and reused (Bell, Hatlestad, Hansteen and Araldsen, 1973).

Our first goal is to enunciate two simple principles for identifying the important strengths and weaknesses of parallel programming systems (both hardware and software): locality and regularity of operation. We do this in Section 2. Only by understanding how a particular parallel system embodies these principles can one design a good parallel algorithm for it; we illustrate this in Section 3 using matrix multiplication.[1]

Besides matrix multiplication, we discuss parallel numerical algorithms for linear equation solving, least squares problems, symmetric and nonsymmetric eigenvalue problems, and the singular value decomposition. We organize this material with dense and banded linear equation solving in Section 4, least squares problems in Section 5, eigenvalue and singular value problems in Section 6, direct methods for sparse linear systems in Section 7, iterative methods for linear systems in Section 8, and iterative methods for eigenproblems in Section 9. We restrict ourselves to general techniques, rather than techniques like

[1] This discussion will not entirely prepare the reader to write good programs on any particular machine, since many machine-specific details will remain.

multigrid and domain decomposition that are specialized for particular application areas.

We emphasize algorithms that are *scalable*, i.e. remain efficient as they are run on larger problems and larger machines. As problems and machines grow, it is desirable to avoid algorithm redesign. As we will see, we will sometimes pay a price for this scalability. For example, though many parallel algorithms are parallel versions of their serial counterparts with nearly identical roundoff and stability properties, others are rather less stable, and would not be the algorithm of choice on a serial machine.

Any survey of such a busy field is necessarily a snapshot reflecting some of the authors' biases. Other recent surveys include Dongarra, Duff, Sorensen and van der Vorst (1991) and Gallivan, Heath, Ng, Ortega, Peyton, Plemmons, Romine, Sameh and Voigt (1990), the latter of which includes a bibliography of over 2000 entries.

2 Features of parallel systems

2.1 General principles

A large number of different parallel computers (Gottlieb and Almasi, 1989), languages (see Zima and Chapman (1991) and the references therein), and software tools have recently been built or proposed. Though the details of these systems vary widely, there are two basic issues they must deal with, and these will guide us in understanding how to design and analyse parallel algorithms. These issues are *locality* and *regularity* of computation.

Locality refers to the proximity of the arithmetic and storage components of computers. Computers store data in memories, which are physically separated from the computational units that perform useful arithmetic or logical functions. The time it takes to move the data from the memory to the arithmetic unit can far exceed the time to perform arithmetic unless the memory is immediately proximate to the arithmetic unit; such memory is usually called the *register file* or *cache*. There are good electrical and economic reasons that not all memory can consist of registers or cache. Therefore all machines, even the simplest PCs, have *memory hierarchies* of fast, small, expensive memory like registers, then slower, larger and cheaper main memory, and finally down to disk or other peripheral storage. Parallel computers have even more levels, possibly including local memory as well as remote memory, which may serve as the local memory for other processors (see Figure 1). Useful arithmetic or logical work can occur only on data stored at the top of the memory hierarchy, and data must be moved from the lower, slower levels in the hierarchy to the top level to participate in computation. Therefore, much of algorithm design involves deciding where and when to store or fetch data in order to minimize this movement. The action of processor i storing or fetching data in memory j in Figure 1 is called

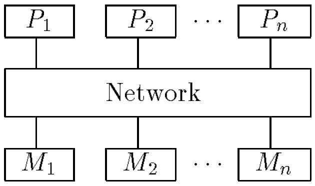

Figure 1: Diagram of a parallel computer (P = processor, M = memory).

communication. Depending on the machine, this may be done automatically by the hardware whenever the program refers to nonlocal data, or it may require explicit sending and/or receiving of messages on the part of the programmer. Communication among processors occurs over a *network*.

A special kind of communication worth distinguishing is *synchronization*, where two or more processors attempt to have their processing reach a commonly agreed upon stage. This requires an exchange of messages as well, perhaps quite short ones, and so qualifies as communication.

A very simple model for the time it takes to move n data items from one location to another is $\alpha + \beta \cdot n$, where $0 \leq \beta, \alpha$. One way to describe α is the *start up time* of the operation; another term for this is *latency*. The incremental time per data item moved is β; its reciprocal is called *bandwidth*. Typically $0 < \beta \ll \alpha$, i.e. it takes a relatively long time to start up an operation, after which data items arrive at a higher rate of speed. This cost model, which we will see again later, reflects the pipeline implementation of the hardware: the pipeline takes a while to fill up, after which data arrive at a high rate.

The constants α and β depend on the parts of the memory between which transfer occurs. Transfer between higher levels in the hierarchy may be orders of magnitude faster than those between lower levels (for example, cache–memory *versus* memory–disk transfer). Since individual memory levels are themselves built of smaller pieces, and may be shared among different parts of the machine, the values of α and β may strongly depend on the location of the data being moved.

Regularity of computation means that the operations parallel machines perform fastest tend to have simple, regular patterns, and efficiency demands that computations be decomposed into repeated applications of these patterns. These regular operations include not only arithmetic and logical operations but communication as well. Designing algorithms that use a very high fraction of these regular operations is, in addition to maintaining locality, one of the major challenges of parallel algorithm design. The simplest and most widely applicable cost models for these regular operations is again $\alpha + \beta \cdot n$, and for the same reason as before: pipelines are ubiquitous.

Amdahl's Law quantifies the importance of using the most efficient parallel operations of the machine. Suppose a computation has a fraction $0 < p < 1$ of its operations which can be effectively parallellized, while the remaining fraction $s = 1 - p$ cannot be. Then with n processors, the most we can decrease the run time is from $s + p = 1$ to $s + p/n$, for a speed up of $1/(s + p/n) \leq 1/s$; thus the serial fraction s limits the speed up, no matter how many parallel processors n we have. Amdahl's Law suggests that only large problems can be effectively parallellized, since for the problems we consider p grows and s shrinks as the problem size grows.

2.2 Examples

We illustrate the principles of regularity and locality with examples of current machines and software systems.

A sequence of machine instructions without a branch instruction is called a *basic block*. Many processors have pipelined execution units that are optimized to execute basic blocks; since there are no branches, the machine can have several instructions in partial stages of completion without worrying that a branch will require 'backing out' and restoring an earlier state. So in this case, regularity of computation means code without branches. An algorithmic implication of this is *loop unrolling*, where the body of a loop like

for $i = 1 : n$
$\quad a_i = a_i + b * c_i$

is replicated four times (say) yielding

for $i = 1 : n$ step 4
$\quad a_i = a_i + b * c_i$
$\quad a_{i+1} = a_{i+1} + b * c_{i+1}$
$\quad a_{i+2} = a_{i+2} + b * c_{i+2}$
$\quad a_{i+3} = a_{i+3} + b * c_{i+3}$

In this case the basic block is the loop body, since the end of the loop is a conditional branch back to the beginning. Unrolling makes the basic block four times longer.

One might expect compilers to perform simple optimizations like this automatically, but many do not, and seemingly small changes in loop bodies can make this difficult to automate (imagine adding the line 'if $i > 1$, $d_i = e_i$' to the loop body, which could instead be done in a separate loop from $i = 2$ to n without the 'if'). For a survey of such compiler optimization techniques see Zima and Chapman (1991). A hardware approach to this problem is *optimistic execution*, where the hardware guesses which way the branch will go and computes ahead under that assumption. The hardware retains enough information to undo what it did a few steps later if it finds out it decided incorrectly. But in the case of branches leading back to the beginning of loops, it will almost always make the right decision. This technique could make unrolling and similar low-level optimizations unnecessary in the future.

A similar example of regularity is *vector pipelining*, where a single instruction initiates a pipelined execution of a single operation on a sequence of data items; componentwise addition or multiplications of two arrays or 'vectors' is the most common example, and is available on machines from Convex, Cray, Fujitsu, Hitachi, NEC, and others. Programmers of such machines prefer the unrolled version of the above loop, and expect the compiler to convert it into, say, a single machine instruction to multiply the vector c by the scalar b, and then add it to vector a.

An even higher level of such regularity is so-called *SIMD parallellism*, which stands for Single Instruction Multiple Data, where each processor in Figure 1 performs the same operation in lockstep on data in its local memory. (SIMD stands in contrast to *MIMD* or Multiple Instruction Multiple Data, where each processor in Figure 1 works independently.) The CM-2 and MasPar depend on this type of operation for their speed. A sample loop easily handled by this paradigm is

for $i = 1 : n$
    if $c_i > 0$ then
        $a_i = b_i + \sqrt{c_i}$
    else
        $a_i = b_i - d_i$
    endif

A hidden requirement for these examples to be truly regular is that no exceptions arise during execution. An exception might be floating point overflow or address out of range. The latter error necessitates an interruption of execution; there is no reasonable way to proceed. On the other hand, there are reasonable ways to continue computing past floating point exceptions, such as *infinity arithmetic* as defined by the IEEE floating point arithmetic standard (ANSI/IEEE, 1985). This increases the regularity of computations by eliminating branches. IEEE arithmetic is implemented on almost all microprocessors, which are often building blocks for larger parallel machines. Whether or not we can make sense out of results that have overflowed or undergone other exceptions depends on the application; it is true often enough to be quite useful.

Now we give some examples of regularity in communication. The CM-2 (Thinking Machines Corporation, 1987) may be thought of in different ways; for us it is convenient to think of it as 2048 processors connected in an *11-dimensional hypercube*, with one processor and its memory at each of the 2048 corners of the cube, and a physical connection along each edge connecting each corner to its 11 nearest neighbours. All 11×2048 such connections may be used simultaneously, provided only one message is sent on each connection.

We illustrate such a regular communication by showing how to compute $f_i = \sum_{j=1}^{N} F(x_i, x_j)$, i.e. an N-body interaction where f_i is the force on body i,

$F(x_i, x_j)$ is the force on body i due to body j, and x_i and x_j are the positions of bodies i and j respectively (Brunet, Edelman and Mesirov, 1990). Consider implementing this on a d-dimensional hypercube, and suppose $N = d2^d$ for simplicity. We need to define the *Gray code* $G(d) \equiv (G_{d,0}, ..., G_{d,2^d-1})$, which is a permutation of the d-bit integers from 0 to $2^d - 1$, ordered so that adjacent codes $G_{d,k}$ and $G_{d,k+1}$ differ only in one bit. $G(d)$ may be defined recursively by taking the $d-1$-bit numbers $G(d-1)$, followed by the same numbers in reverse order and incremented by 2^{d-1}. For example, $G(2) = \{00, 01, 11, 10\}$ and $G(3) = \{000, 001, 011, 010, 110, 111, 101, 100\}$. Now imagining our hypercube as a unit hypercube in d-space with one corner at the origin and lying in the positive orthant, number each processor by the d-bit string whose bits are the coordinates of its position in d-space. Since the physically nearest neighbours of a processor lie one edge away, their coordinates or processor numbers differ in only one bit. Since $G_{d,k}$ and $G_{d,k+1}$ differ in only one bit, the Gray code sequence describes a path among the processors in a minimal number of steps visiting each one only once; such a path is called *Hamiltonian*. Now define the *shifted Gray code* $G^{(s)}(d) = \{G^{(s)}_{d,0}, ..., G^{(s)}_{d,2^d-1}\}$ where $G^{(s)}_{d,k}$ is obtained by left-circular shifting $G_{d,k}$ by s bits. Each $G^{(s)}(d)$ also defines a Hamiltonian path, and all may be traversed simultaneously without using any edges simultaneously. Let $g^{(s)}_{d,k}$ denote the bit position in which $G^{(s)}_{d,k}$ and $G^{(s)}_{d,k+1}$ differ.

Now we define the program each processor will execute in order to compute f_i for the bodies it owns. Number the bodies $x_{k,l}$, where $0 \le l \le 2^d - 1$ is the processor number and $0 \le k \le d-1$; so processor l owns $x_{0,l}$ through $x_{d-1,l}$. Then processor l executes the following code, where 'forall' means each iteration may be done in parallel (a sample execution for $d = 2$ is shown in Figure 2).

Algorithm 1 N-body force computation on a hypercube

for $k = 0 : d-1$, $tmp_k = x_{k,l}$
for $k = 0 : d-1$, $f_{k,l} = 0$ /* $f_{k,l}$ will accumulate force on $x_{k,l}$ */
for $m = 0 : 2^d - 1$
    forall $k = 0 : d-1$, swap tmp_k with processor in direction $g^{(k)}_{d,m}$
    for $k = 0 : d-1$
        for $k' = 0 : d-1$
            $f_{k,l} = f_{k,l} + F(x_{k,l}, tmp_{k'})$

In Section 3 we will show how to use Gray codes to implement matrix multiplication efficiently. Each processor of the CM-2 can also send data to any other processor, not just its immediate neighbours, with the network of physical connections forwarding a message along to the intended receiver like a network of post-offices. Depending on the communication pattern this may lead to congestion along certain connections and so be much slower than the special communication pattern discussed earlier.

Here are some other useful regular communication patterns. A *broadcast* sends data from a single source to all other processors. A *spread* may be de-

Initial data		After $m = 0$		After $m = 1$		After $m = 2$	
$x_{0,0}, x_{1,0}$	$x_{0,2}, x_{1,2}$	$x_{0,1}, x_{1,2}$	$x_{0,3}, x_{1,0}$	$x_{0,3}, x_{1,3}$	$x_{0,1}, x_{1,1}$	$x_{0,2}, x_{1,1}$	$x_{0,0}, x_{1,3}$
$x_{0,1}, x_{1,1}$	$x_{0,3}, x_{1,3}$	$x_{0,0}, x_{1,3}$	$x_{0,2}, x_{1,1}$	$x_{0,2}, x_{1,2}$	$x_{0,0}, x_{1,0}$	$x_{0,3}, x_{1,0}$	$x_{0,1}, x_{1,2}$

Figure 2: Force computation on two-dimensional hypercube.

scribed as partitioned broadcast, where the processors are partitioned and a separate broadcast done within each partition. For example, in a square array of processors we might want to broadcast a data item in the first column to all other processors in its row; thus we partition the processor array into rows and do a broadcast to all the others in the partition from the first column. This operation might be useful in Gaussian elimination, where we need to subtract multiples of one matrix column from the other matrix columns. Another operation is a *reduction*, where data distributed over the machine are reduced to a single datum by applying an associative operation like addition, multiplication, maximum, logical or, and so on; this operation is naturally supported by processors connected in a tree, with information being reduced as it passes from the leaves to the root of the tree.

A more general operation than reduction is the *scan* or *parallel prefix* operation. Let $x_0, ... x_n$ be data items, and $\cdot$ any associative operation. Then the scan of these n data items yields another n data items defined by $y_0 = x_0$, $y_1 = x_0 \cdot x_1$, ... , $y_i = x_0 \cdot x_1 \cdots x_i$; thus y_i is the reduction of x_0 through x_i. An attraction of this operation is its ease of implementation using a simple tree of processors. We illustrate in Figure 3 for $n = 15$, or f in hexadecimal notation; in the figure we abbreviate x_i by i and $x_i \cdots x_j$ by $i : j$. Each row indicates the values held by the processors; after the first row only the data that change are indicated. Each updated entry combines its current value with one a fixed distance to its left.

Parallel prefix may be used, for example, to solve linear recurrence relations $z_{i+1} = \sum_{j=0}^{n} a_{i,j} z_{i-j} + b_i$; this can be converted into simple parallel operations on vectors plus parallel prefix operations where the associative operators are n by n matrix multiplication and addition. For example, to evaluate $z_{i+1} = a_i z_i + b_i$, $i \geq 0$, $z_0 = 0$, we do the following operations:

Algorithm 2 Linear recurrence evaluation using Parallel Prefix

Compute $p_i = a_0 \cdots a_i$ using parallel prefix multiplication
Compute $\beta_i = b_i / p_i$ in parallel
Compute $s_i = \beta_0 + \cdots + \beta_{i-1}$ using parallel prefix addition
Compute $z_i = s_i \cdot p_{i-1}$ in parallel

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0:1		2:3		4:5		6:7		8:9		a:b		c:d		e:f
			0:3				4:7				8:b				c:f
							0:7								8:f
															0:f
											0:b				
					0:5				0:9				0:d		
		0:2		0:4		0:6		0:8		0:a		0:c		0:e	

Figure 3: Parallel prefix on 16 data items.

Similarly, we can use parallel prefix to evaluate certain rational recurrences $z_{i+1} = (a_i z_i + b_i)/(c_i z_i + d_i)$ by writing $z_i = u_i/v_i$ and reducing to a linear recurrence for u_i and v_i:

$$\begin{bmatrix} u_{i+1} \\ v_{i+1} \end{bmatrix} = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix} \cdot \begin{bmatrix} u_i \\ v_i \end{bmatrix}. \tag{1}$$

We may ask more generally about evaluating the scalar rational recurrence $z_{i+1} = f_i(z_i)$ in parallel. Let d be the maximum of the degrees of the numerators and denominators of the rational functions f_i. Then Kung (1974) has shown that z_i can be evaluated faster than linear time (i.e. z_i can be evaluated in $o(i)$ steps) if and only if $d \leq 1$; in this case the problem reduces to 2×2 matrix multiplication parallel prefix in (1). Interesting linear algebra problems that can be cast in this way include tridiagonal Gaussian elimination, solving bidiagonal linear systems of equations, Sturm sequence evaluation for the symmetric tridiagonal eigenproblem, and the bidiagonal dqds algorithm for singular values (Parlett and Fernando, 1992); we discuss some of these later. The numerical stability of these procedures remains open, although it is often good in practice (Swarztrauber, 1992).

We now turn to the principle of locality. Since this is an issue many algorithms do not take into account, a number of so-called *shared memory machines* have been designed in which the hardware attempts to make all memory locations look equidistant from every processor, so that old algorithms will continue to work well. Examples include machines from Convex, Cray, Fujitsu, Hitachi, NEC, and others (Gottlieb and Almasi, 1989). The memories of these machines are organized into some number, say b, of *memory banks*, so that memory address m resides in memory bank $m \bmod b$. A memory bank is designed so that it takes b time steps to read/write a data item after it is asked to do so; until then it is busy and cannot do anything else. Suppose one wished to read or write a sequence of $n+1$ memory locations i, $i+s$, $i+2s$, ... , $i+ns$; these will then refer to memory banks $i \bmod b$, $i+s \bmod b$, ... , $i+ns \bmod b$. If $s=1$, so that we refer to consecutive memory locations, or if s and b are relatively prime, b consecutive memory references will refer to b different memory banks, and so after a wait of b steps the memory will deliver a result once per time step; this is the fastest it can operate. If instead $gcd(s,b)=g>1$, then only b/g memory banks will be referenced, and speed of access will slow down by a factor of g. For example, suppose we store a matrix by columns, and the number of rows is s. Then reading a column of the matrix will be $gcd(s,b)$ times faster than reading a row of the matrix, since consecutive row elements have memory addresses differing by s; this clearly affects the design of matrix algorithms. Sometimes these machines also support *indirect addressing* or *gather/scatter*, where the addresses can be arbitrary rather than forming an arithmetic sequence, although it may be significantly slower.

Another hardware approach to making memory access appear regular are *virtual shared memory machines* like the Kendall Square Research machine and Stanford's Dash. Once the memory becomes large enough, it will necessarily be implemented as a large number of separate banks. These machines have a hierarchy of caches and directories of pointers to caches to enable the hardware to locate quickly and fetch or store a nonlocal piece of data requested by the user; the hope is that the cache will successfully anticipate enough of the user's needs to keep them local. To the extent that these machines fulfil their promise, they will make parallel programming much easier; as of this writing it is too early to judge their success.[2]

For machines on which the programmer must explicitly send or receive messages to move data, there are two issues to consider in designing efficient algorithms. The first issue is the relative cost of communication and computation. Recall that a simple model of communicating n data items is $\alpha + n\beta$; let γ be the average cost of a floating point operation. If $\alpha \gg \beta$, which is not uncommon, then sending n small messages will cost $n(\alpha+\beta)$, which can exceed by nearly a factor n the cost of a single message $\alpha + n\beta$. This forces us to design algorithms that do infrequent communications of large messages, which is not always convenient. If $\alpha \gg \gamma$ or $\beta \gg \gamma$, which are both common, then we will also be motivated to design algorithms that communicate as infrequently as possible. An algorithm which communicates infrequently is said to exhibit *coarse-grained parallellism*, and otherwise *fine-grained parallellism*. Again this

[2] There is a good reason to hope for the success of these machines: parallel machines will not be widely used if they are hard to program, and maintaining locality explicitly is harder for the programmer than having the hardware do it automatically.

is sometimes an inconvenient constraint, and makes it hard to write programs that run efficiently on more than one machine.

The second issue to consider when sending messages is the semantic power of the messages (Wen and Yelick, 1992). The most restrictive possibility is that the processor executing 'send' and the processor executing 'receive' must synchronize, and so block until the transaction is completed. So for example, if one processor sends long before the other receives, it must wait, even if it could have continued to do useful work. At the least restrictive the sending processor effectively interrupts the receiving processors and executes an arbitrary subroutine on the contents of the message, without any action by the receiving program; this minimizes time wasted waiting, but places a burden on the user program to do its own synchronization.

To illustrate these points, imagine an algorithm that recursively subdivides problems into smaller ones, but where the subproblems can be of widely varying complexity that cannot be predicted ahead of time. Even if we divide the initial set of problems evenly among our processors, the subproblems generated by each processor may be very different. A simple example is the use of Sturm sequences to compute the eigenvalues of a symmetric tridiagonal matrix. Here the problem is to find the eigenvalues in a given interval, and the subproblems correspond to subintervals. The time to solve a subproblem depends not only on the number but also on the distribution of eigenvalues in the subinterval, which is not known until the problem is solved. In the worst case, all processors but one finish quickly and remain idle while the other one does most of the work. Here it makes sense to do *dynamic load balancing*, which means redistributing to idle processors those subproblems needing further processing. This clearly requires communication, and may or may not be effective if communication is too expensive.

2.3 Important tradeoffs

We are accustomed to certain tradeoffs in algorithm design, such as time *versus* space: an algorithm that is constrained to use less space may have to go more slowly than one not so constrained. There are certain other tradeoffs that arise in parallel programming. They arise because of the constraints of regularity of computation and locality to which we should adhere. For example, load balancing to increase parallellism requires communication, which may be expensive. Limiting oneself to the regular operations the hardware performs efficiently may result in wasted effort or use of less sophisticated algorithms; we will illustrate this later in the case of the nonsymmetric eigenvalue problem.

Another interesting tradeoff is parallellism *versus* numerical stability. For some problems the most highly parallel algorithms known are less numerically stable than the conventional sequential algorithms. This is true for various kinds of linear systems and eigenvalue problems, which we will point out as they arise. Some of these tradeoffs can be mitigated by better floating point arithmetic (Demmel, 1992b). Others can be dealt with by using the following simple paradigm:

1. Solve the problem using a fast method, provided it is rarely unstable.
2. Quickly and reliably confirm or deny the accuracy of the computed solution. With high probability, the answer just (quickly) computed is accurate enough.
3. Otherwise, fall back on a slower but more reliable algorithm.

For example, the most reliable algorithm for the dense nonsymmetric eigenvalue problem is Hessenberg reduction and QR iteration, but this is hard to parallellize. Other routines are faster but occasionally unreliable. These routines can be combined according to the paradigm to yield a guaranteed stable algorithm which is fast with high probability (see Section 6.5).

3 Matrix multiplication

Matrix multiplication is a very regular computation that is basic to linear algebra and lends itself well to parallel implementation. Indeed, since it is the easiest nontrivial matrix operation to implement efficiently, an effective approach to designing other parallel matrix algorithms is to decompose them into a sequence of matrix multiplications; we discuss this in detail in later sections.

One might well ask why matrix multiplication is more basic than matrix–vector multiplication or adding a scalar times one vector to another vector. Matrix multiplication can obviously be decomposed into these simpler operations, and they also seem to offer a great deal of parallelism. The reason is that matrix multiplication offers much more opportunity to exploit locality than these simpler operations. An informal justification for this is as follows.

Table 1 gives the number of floating point operations (flops), the minimum number of memory references, and their ratio q for the three Basic Linear Algebra Subroutines, or BLAS: scalar-times-vector-plus-vector (or `saxpy` for short, for $\alpha x + y$), matrix–vector multiplication, and matrix–matrix multiplication (for simplicity only the highest order term in n is given for q). When the data are too large to fit in the top of the memory hierarchy, we wish to perform the most flops per memory reference to minimize data movement; q gives an upper bound on this ratio for any implementation. We see that only matrix multiplication offers us an opportunity to make this ratio large.

This table reflects a hierarchy of operations: Operations like `saxpy` operate on vectors and offer the worst q values; these are called Level 1 BLAS (Lawson, Hanson, Kincaid and Krogh, 1979) 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

Table 1: Memory references and operation counts for the BLAS.

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

(Dongarra, Du Croz, Hammarling and Richard Hanson, 1988), 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 (Dongarra, Du Croz, Duff and Hammarling, 1990), and include solving triangular systems of equations with many right hand sides. These operations have been standardized, and many high performance computers have highly optimized implementations of these that are useful for building more complicated algorithms (Anderson, Bai, Bischof, Demmel, Dongarra, Du Croz, Greenbaum, Hammarling, McKenney, Ostrouchov and Sorensen, 1992); this is the subject of several succeeding sections.

3.1 Matrix multiplication on a shared memory machine

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 3 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. 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 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 5 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 Hong and Kung (1981) an analysis of this problem leading to an upper bound near $\sqrt{M}$ is given, so we cannot expect to improve much on this algorithm for square matrices. 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 (but see the discussion in Section 3.3);

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 (Gallivan *et al.*, 1990).

Another possibility is Strassen's method (Aho, Hopcroft and Ullman, 1974), 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 (Bailey, Lee and Simon, 1991). A drawback is the need for significant workspace, and somewhat lower numerical stability, although it is adequate for many purposes (Demmel and Higham, 1992; Higham, 1990).

Given the complexity of optimizing the implementation of matrix multiplication, we cannot expect all other matrix algorithms to be equally optimized on all machines, at least not in a time users are willing to wait. Indeed, since architectures change rather quickly, we prefer to do as little machine-specific optimization as possible. Therefore, our shared memory algorithms in later sections assume only that highly optimized BLAS are available and build on top of them.

3.2 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.

A dominant issue is *data layout*, or how the matrices are partitioned across the machine. This will determine both the amount of parallellism and the cost of communication. 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 first algorithm is due to Cannon (1969) 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

$A^{(00)}$	$A^{(01)}$	$A^{(02)}$
$A^{(11)}$	$A^{(12)}$	$A^{(10)}$
$A^{(22)}$	$A^{(20)}$	$A^{(21)}$

$B^{(00)}$	$B^{(11)}$	$B^{(22)}$
$B^{(10)}$	$B^{(21)}$	$B^{(02)}$
$B^{(20)}$	$B^{(01)}$	$B^{(12)}$

A, B after skewing

$A^{(01)}$	$A^{(02)}$	$A^{(00)}$
$A^{(12)}$	$A^{(10)}$	$A^{(11)}$
$A^{(20)}$	$A^{(21)}$	$A^{(22)}$

$B^{(10)}$	$B^{(21)}$	$B^{(02)}$
$B^{(20)}$	$B^{(01)}$	$B^{(12)}$
$B^{(00)}$	$B^{(11)}$	$B^{(22)}$

A, B after shift $k = 1$

$A^{(02)}$	$A^{(00)}$	$A^{(01)}$
$A^{(10)}$	$A^{(11)}$	$A^{(12)}$
$A^{(21)}$	$A^{(22)}$	$A^{(20)}$

$B^{(20)}$	$B^{(01)}$	$B^{(12)}$
$B^{(00)}$	$B^{(11)}$	$B^{(22)}$
$B^{(10)}$	$B^{(21)}$	$B^{(02)}$

A, B after shift $k = 2$

Figure 4: Cannon's algorithm for $N = 3$.

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 6 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 4 illustrates the functioning of this 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) (Fox, Johnson, Lyzenga, Otto, Salmon and Walker, 1988).

This algorithm is easily adapted to a hypercube. The simplest way is to embed a grid (or two-dimensional torus) in a hypercube, i.e. map the processors

0000	0001	0011	0010
0100	0101	0111	0110
1100	1101	1111	1110
1000	1001	1011	1010

Figure 5: Embedding a 4×4 grid in a four-dimensional hypercube (numbers are processor numbers in hypercube).

in a grid to the processors in a hypercube, and the connections in a grid to a subset of the connections in a hypercube (Ho, 1990; Johnsson, 1987). Suppose the hypercube is d-dimensional, so the 2^d processors are labelled by d bit numbers. We embed a $2^n \times 2^m$ grid in this hypercube (where $m+n=d$) by mapping processor (i_1, i_2) in the grid to processor $G_{n,i_1}2^m + G_{m,i_2}$ in the hypercube; i.e. we just concatenate the n bits of G_{n,i_1} and m bits of G_{m,i_2}. Each row (column) of the grid thus occupies an m- (n-) dimensional subcube of the original hypercube, with nearest neighbours in the grid mapped to nearest neighbours in the hypercube (Ho, Johnsson and Edelman, 1991). We illustrate for a 4×4 grid in Figure 5. This approach easily extends to multi-dimensional arrays of size $2^{m_1} \times \cdots \times 2^{m_r}$, where $\sum_{i=1}^{r} m_i$ is at most the dimension of the hypercube.

This approach (which is useful for more than matrix multiplication) uses only a subset of the connections in a hypercube, which makes the initial skewing operations slower than they need be: if we can move only to nearest neighbours, each skewing operation takes $N-1$ communication steps, as many as in the computational loop. We may use all the wires of the hypercube to reduce the skewing to $\log_2 N$ operations. In the following algorithm, $\otimes$ denotes the bitwise exclusive-or operator. We assume the $2^n \times 2^n$ grid of data is embedded in the hypercube so that $A^{(i,j)}$ is stored in processor $i \cdot 2^n + j$ (Dekel, Nassimi and Sahni, 1981):

Algorithm 7 Dekel's matrix multiplication algorithm

for $k = 1 : n$
    Let $j_k = (k\text{th bit of } j) \cdot 2^k$
    Let $i_k = (k\text{th bit of } i) \cdot 2^k$
    forall $i = 0 : 2^n - 1$, forall $j = 0 : 2^n - 1$
        Swap $A^{(i,j \otimes i_k)}$ and $A^{(i,j)}$
        Swap $B^{(j_k \otimes i,j)}$ to $B^{(i,j)}$
for $k = 1 : 2^n$
    forall $i = 0 : 2^n - 1$, forall $j = 0 : 2^n - 1$
        $C^{(ij)} = C^{(ij)} + A^{(ij)} \cdot B^{(ij)}$
        Swap $A^{(i,j \otimes g_{d,k})}$ and $A^{(i,j)}$

Table 2: Cost of matrix multiplication on a hypercube.

Algorithm	Message Startups	Data Sending Steps	Floating Point Steps
Cannon (6)	$2(2^n-1)$	$2n^2(2^n-1)$	$2n^3 2^n$
Dekel (7)	$n+2^n-1$	$n^3+n^2(2^n-1)$	$2n^3 2^n$
Ho *et al.* (8)	$n+2^n-1$	$n^3+n(2^n-1)$	$2n^3 2^n$

Swap $B^{(i\otimes g_{d,k},j)}$ and $B^{(i,j)}$

Finally, we may speed this up further (Ho *et al.*, 1991; Johnsson and Ho, 1989) provided the $A^{(i,j)}$ blocks are large enough, by using the same algorithm as for force calculations in Section 2. If the blocks are n by n (so A and B are $n2^n \times n2^n$), then the algorithm becomes

Algorithm 8 Ho, Johnsson, and Edelman's matrix multiplication algorithm

for $k = 1 : n$
  Let $j_k = (k\text{th bit of } j) \cdot 2^k$
  Let $i_k = (k\text{th bit of } i) \cdot 2^k$
  forall $i = 0 : 2^n - 1$, forall $j = 0 : 2^n - 1$
    Swap $A^{(i,j\otimes i_k)}$ and $A^{(i,j)}$
    Swap $B^{(j_k\otimes i,j)}$ to $B^{(i,j)}$
for $k = 1 : 2^n$
  forall $i = 0 : 2^n - 1$, forall $j = 0 : 2^n - 1$
    $C^{(ij)} = C^{(ij)} + A^{(ij)} \cdot B^{(ij)}$
    forall $l = 0 : n - 1$
      Swap $A_l^{(i,j\otimes g_{d,k}^{(l)})}$ and $A_l^{(i,j)}$ ($A_l^{(ij)}$ is the lth row of $A^{(ij)}$)
      Swap $B_l^{(i\otimes g_{d,k}^{(l)},j)}$ and $B_l^{(i,j)}$ ($B_l^{(ij)}$ is the lth column of $B^{(ij)}$)

Algorithms 6, 7 and 8 all perform the same number of floating point operations in parallel. Table 2 compares the number of communication steps, assuming matrices are $n2^n \times n2^n$, swapping a datum along a single wire is one step, and the motions of A and B that can occur in parallel do occur in parallel. Note that for large enough n the number of floating point steps overwhelms the number of communication steps, so the efficiency gets better.

In this section we have shown how to 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. Our 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. If the architecture permitted us to overlap communication and computation, we could pipeline the algorithm to mask communication cost further.

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 (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.[3] 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).

3.3 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 algorithms of the previous section use 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 $r \times r$ matrix on each processor.

There is an emerging consensus about data layouts for distributed memory machines. This is being implemented in several programming languages (Fox, Hiranandani, Kennedy, Koelbel, Kremer, Tseng and Wu, 1990; High Peformance Fortran, 1991), that will be available to programmers in the near future. We describe these layouts here.

High Performance Fortran (HPF) (High Peformance Fortran, 1991) permits the user to define a virtual array of processors, align actual data structures

[3]The matrix multiplication subroutine in the CM-2 Scientific Subroutine Library took approximately 10 person-years of effort (Johnsson, 1990).

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* (or *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. 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 6 through 8 for a 16×16 matrix laid out on a 4×4 processor grid; each array entry is labelled by the number of the processor that stores it.

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 7 is identical to the layout in Figure 6 of P^TAP, where P is a permutation matrix. This shows that running the algorithms of the previous 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.

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

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 7: 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 8: Block scatter layout of a 16×16 matrix on a 4×4 processor grid with 2×2 blocks.

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 (Van de Velde, 1992).

4 Systems of linear equations

We discuss both dense and band matrices, on shared and distributed memory machines. We begin with dense matrices and shared memory, showing how the standard algorithm can be reformulated as a block algorithm, calling the Level 2 and 3 BLAS in its innermost loops. The distributed memory versions will be similar, with the main issue being laying out the data to maximize load balance and minimize communication. We also present some highly parallel, but numerically unstable, algorithms to illustrate the tradeoff between stability and parallellism. We conclude with some algorithms for band matrices.

4.1 Gaussian elimination on a shared memory machine

To solve $Ax = b$, we first use Gaussian elimination to factor the nonsingular matrix A as $PA = LU$, where L is lower triangular, U is upper triangular, and P is a permutation matrix. Then we solve the triangular systems $Ly = Pb$ and $Ux = y$ for the solution x. In this section we concentrate on factoring

$PA = LU$, which has the dominant number of floating point operations, $2n^3/3 + \mathcal{O}(n^2)$. Pivoting is required for numerical stability, and we use the standard partial pivoting scheme (Golub and Van Loan, 1989); this means L has unit diagonal and other entries bounded in magnitude by one. The simplest version of the algorithm involves adding multiples of one row of A to others to zero out subdiagonal entries, and overwriting A with L and U:

Algorithm 9 Row oriented Gaussian elimination (kij-LU decomposition)

for $k = 1 : n-1$
  { choose l so $|A_{lk}| = \max_{k \le i \le n} |A_{ik}|$, swap rows l and k of A }
  for $i = k+1 : n$
    $A_{ik} = A_{ik}/A_{kk}$
    for $j = k+1 : n$
      $A_{ij} = A_{ij} - A_{ik} \cdot A_{kj}$

There is obvious parallellism in the innermost loop, since each A_{ij} can be updated independently. If A is stored by column, as is the case in Fortran, then since the inner loop combines rows of A, it accesses memory entries (at least) n locations apart. As described in Section 2, this does not respect locality. Algorithm 9 is also called kij-LU decomposition, because of the nesting order of its loops. All the rest of 3! permutations of i, j and k lead to valid algorithms, some of which access columns of A in the innermost loop. Algorithm 10 is one of these, and is used in the LINPACK routine `sgefa` (Dongarra *et al.*, 1979):

Algorithm 10 Column oriented Gaussian elimination (kji-LU decomposition)

for $k = 1 : n-1$
  { choose l so $|A_{lk}| = \max_{k \le i \le n} |A_{ik}|$, swap A_{lk} and A_{kk} }
  for $i = k+1 : n$
    $A_{ik} = A_{ik}/A_{kk}$
  for $j = k+1 : n$
    { swap A_{lj} and A_{kj} }
    for $i = k+1 : n$
      $A_{ij} = A_{ij} - A_{ik} \cdot A_{kj}$

The inner loop of Algorithm 10 can be performed by a single call to the Level 1 BLAS operation `saxpy`. 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 (Anderson *et al.*, 1992). To make the use of BLAS clear, we use matrix/vector operations instead of loops:

Algorithm 11 Gaussian elimination using Level 2 BLAS

for $k = 1 : n-1$
  { choose l so $|A_{lk}| = \max_{k \le i \le n} |A_{ik}|$, swap rows l and k of A }

$$
\begin{aligned}
&A(k+1:n,k) = A(k+1:n,k)/A_{kk} \\
&A(k+1:n,k+1:n) = A(k+1:n,k+1:n) \\
&\qquad -A(k+1:n,k)\cdot A(k,k+1:n)
\end{aligned}
$$

The parallellism in the inner loop 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 (Anderson and Dongarra, 1990; Robert, 1990); 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 11 in the fast memory, and then use Level 3 BLAS to update the rest of the matrix:

Algorithm 12 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 11 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 11 and 12 are available as, respectively, subroutines `sgetf2` and `sgetrf` in LAPACK (Anderson *et al.*, 1992).

We illustrate these points with the slightly different example of Cholesky decomposition, which uses a very similar algorithm: Table 3 shows the speeds in megaflops of the various BLAS and algorithms on 1 and 8 processors of a Cray YMP.

Table 3: Speed of Cholesky on a Cray YMP

	1 PE	8 PEs
Maximum speed	330	2640
LINPACK (Cholesky with BLAS 1), $n = 500$	72	72
Matrix–vector multiplication	311	2285
Matrix–matrix multiplication	312	2285
Triangular solve (one right hand side)	272	584
Triangular solve (many right hand sides)	309	2398
LAPACK (Cholesky with BLAS 3), $n = 500$	290	1414
LAPACK (Cholesky with BLAS 3), $n = 1000$	301	2115

4.2 Gaussian elimination on a distributed memory machine

As described earlier, layout strongly 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 12, with 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 11 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, as described in Section 2. 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 12, 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 12 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 Section 2, 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 Robert (1990) and the references therein. See also Dongarra and Ostrouchov (1990) and Dongarra and van de Geijn (1991a). 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 3.3, 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_1AP_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 (see Eisenstat, Heath, Henkel and Romine, 1988; Heath and Romine, 1988; Li and Coleman, 1988; Romine and Ortega, 1988).

4.3 Clever but impractical parallel algorithms for solving $Ax = b$

The theoretical literature provides us with a number of apparently fast but ultimately unattractive algorithms for solving $Ax = b$. These may be unattractive because they need many more parallel processors than is reasonable, ignore locality, are numerically unstable, or any combination of these reasons. We begin with an algorithm for solving $n \times n$ triangular linear systems in $\mathcal{O}(\log^2 n)$ parallel steps. Suppose T is lower triangular with unit diagonal (the diagonal can be factored out in one parallel step). For each i from 1 to $n-1$, let T_i equal the identity matrix except for column i where it matches T. Then it is simple to verify $T = T_1T_2\cdots T_{n-1}$ and so $T^{-1} = T_{n-1}^{-1}\cdots T_2^{-1}T_1^{-1}$. One can also easily see that T_i^{-1} equals the identity except for the subdiagonal of column i, where it is the negative of T_i. Thus it takes no work to compute the T_i^{-1}, and the work involved is to compute the product $T_{n-1}^{-1}\cdots T_1^{-1}$ in $\log_2 n$ parallel steps using a tree. Each parallel step involves multiplying $n \times n$ matrices (which are initially quite sparse, but fill up), and so takes about $\log_2 n$ parallel substeps, for a total of $\log_2^2 n$. The error analysis of this algorithm (Sameh and Brent, 1977) yields an error bound proportional to $\kappa(T)^3\varepsilon$ where $\kappa(T) = \|T\|\cdot\|T^{-1}\|$ is the condition number and ε is machine precision; this is in contrast to the error bound $\kappa(T)\varepsilon$ for the usual algorithm. The error bound for the parallel algorithm may be pessimistic – the worst example we have found has an error growing like $\kappa(T)^{1.5}\varepsilon$ – but shows that there is a tradeoff between parallellism and stability. Also, to achieve the maximum speedup $\mathcal{O}(n^3)$ processors are required, which is unrealistic for large n.

We can use this algorithm to build an $\mathcal{O}(\log^2 n)$ algorithm for the general problem $Ax = b$ (Csanky, 1977), but this algorithm is so unstable as to be entirely useless in floating point (in IEEE double precision floating point, it achieves no precision in inverting $3I$, where I is an identity matrix of size 60 or larger). There are four steps:

1. Compute the powers of A (A^2, A^3, ... , A^{n-1}) by repeated squaring ($\log_2 n$ matrix multiplications of $\log_2 n$ steps each).
2. Compute the traces $s_i = \text{tr}(A^i)$ of the powers in $\log_2 n$ steps.
3. Solve the Newton identities for the coefficients a_i of the characteristic polynomial; this is a triangular system of linear equations whose matrix entries and right hand side are known integers and the s_i (we can do this in $\log_2^2 n$ steps as described above).
4. Compute the inverse using Cayley-Hamilton Theorem (in about $\log_2 n$ steps).

For a survey of other theoretical algorithms, see Bertsekas and Tsitsiklis (1989) and Karp and Ramachandran (1990).

4.4 Solving banded systems

These problems do not lend themselves as well to the techniques described above, especially for small bandwidth. The reason is that proportionately less and less parallel work is available in updating the trailing submatrix, and in the limiting case of tridiagonal matrices, the parallel algorithm derived as above and the standard serial algorithm are nearly identical. If the bandwidth is wide enough, however, the techniques of the previous sections still apply (Du Croz *et al.*, 1990; Fox *et al.*, 1988).

The problem of solving banded linear systems with a narrow band has been studied by many authors, see for instance the references in Gallivan, Heath, Ng *et al.* (1990) and Ortega (1988). 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 Babuska (1972) 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:

$$
\begin{pmatrix}
\star & \star & & & & & & \\
\star & \star & \star & & & & & \\
& \star & \star & \star & & & & \\
& & \star & \ddots & \ddots & & & \\
& & & \ddots & \ddots & \ddots & & \\
& & & & \ddots & \star & \star & \\
& & & & & \star & \star & \star \\
& & & & & & \star & \star
\end{pmatrix}
\tag{2}
$$

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

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 parallellism by working from both ends of the matrix simultaneously. For an analysis of this process for tridiagonal systems, see van der Vorst (1987a). Twisted factorization can be combined with any of the following techniques, often doubling the parallellism.

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 parallellization 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* (Stone, 1973). In Algorithm 2 (Section 2.2) it was presented as a special case of parallel prefix. It has been analysed and generalized for banded systems in Dubois and Rodrigue (1977). 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* (Lambiotte and Voigt, 1974; Heller, 1978). 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 (Ortega, 1988; de Groen, 1991; Schlichting and van der Vorst, 1987). 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 parallellism. 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 by Chen, Kuck and Sameh (1978), for tridiagonal systems it was proposed by Wang (1981).

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

$$\tilde{L} = \begin{pmatrix} 1 &&&&&&&&&&&&&&& \\ & 1 &&&&&&&&&&&&&& \\ && 1 &&&&&&&&&&&&& \\ &&& 1 &&&&&&&&&&&& \\ &&& \star & 1 &&&&&&&&&&& \\ &&& \star && 1 &&&&&&&&&& \\ &&& \star &&& 1 &&&&&&&&& \\ &&& \star &&&& 1 &&&&&&&& \\ &&&&&&& \star & 1 &&&&&&& \\ &&&&&&& \star && 1 &&&&&& \\ &&&&&&& \star &&& 1 &&&&& \\ &&&&&&& \star &&&& 1 &&&& \\ &&&&&&&&&&& \star & 1 &&& \\ &&&&&&&&&&& \star && 1 && \\ &&&&&&&&&&& \star &&& 1 & \\ &&&&&&&&&&& \star &&&& 1 \end{pmatrix} . \qquad (3)$$

There are two possibilities for the next step. In the original approach (Wang, 1981), 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 van der Vorst and Dekker (1989) that this leads to very efficient vectorized code for machines such as Cray, Fujitsu, etc.

For parallel computers, the parallellism in eliminating these subdiagonal blocks is relatively fine-grained compared with the more coarse-grained parallellism 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 Michielse and van der Vorst (1988) 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 (Michielse and van der Vorst, 1988).

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 (van der Vorst, 1989c). For a generalization of the divide-and-conquer approach for banded systems, see Meier (1985); the data transport aspects for distributed memory machines have been discussed in Michielse and van der Vorst (1988).

There are other variants of the divide-and-conquer approach that move the fill-in into other columns of the subblocks or are more stabile numerically. For example, in Mehrmann (1991) the matrix is split into a block diagonal matrix and a remainder via rank-1 updates.

5 Least squares problems

Most algorithms for finding the x minimizing $||Ax-b||_2$ require computing a QR decomposition of A, where Q is orthogonal and R is upper triangular. We will assume A is $m \times n$, $m \geq n$, so that Q is $m \times m$ and R is $m \times n$. For simplicity we consider only QR without pivoting, and mention work incorporating pivoting at the end.

The conventional approach is to premultiply A by a sequence of simple orthogonal matrices Q_i chosen to introduce zeros below the diagonal of A (Golub and Van Loan, 1989). Eventually A becomes upper triangular, and equal to R, and the product $Q_N \cdots Q_1 = Q$. One kind of Q_i often used is a *Givens rotation*, which changes only two rows of A, and introduces a single zero in one of them; it is the identity in all but two rows and columns, where it is $\begin{bmatrix} c & s \\ -s & c \end{bmatrix}$, with $c^2 + s^2 = 1$. A second kind of Q_i is a *Householder reflection*, which can change any number of rows of A, zeroing out all entries but one in the changed rows of one column of A; a Household reflection may be written $I - 2uu^T$, where u is a unit vector with nonzeros only in the rows to be changed.

5.1 Shared memory algorithms

The basic algorithm to compute a QR decomposition using Householder transformations is given in (Golub and Van Loan, 1989):

Algorithm 13 QR decomposition using Level 2 BLAS

for $k = 1 : n-1$
    Compute a unit vector u_k so that $(I - 2u_k u_k^T)A(k+1:m, k) = 0$
    Update $A = A - 2 * u_k(u_k^T A)$ $(= Q_k A$ where $Q_k = I - 2u_k u_k^T)$

Computing u_k takes $\mathcal{O}(n-k)$ flops and is essentially a Level 1 BLAS operation. Updating A is seen to consist of a matrix–vector multiplication $(w^T = u_k^T A)$ and a rank-1 update $(A - 2u_k w^T)$, both Level 2 BLAS operations. To convert to Level 3 BLAS requires the observation that one can write $Q_b \cdot Q_{b-1} \cdots Q_1 = I - UTU^T$ where $U = [u_1, ..., u_b]$ is $m \times b$, and T is $b \times b$ and triangular (Schreiber and Van Loan, 1989); for historical reasons this is called a *compact WY transformation*. Thus, by analogy with the LU decomposition with column blocking (Algorithm 12), we may first use Algorithm 13 on a block of n_b columns of A, form U and T of the compact WY transformation, and then update the rest of A by forming $A - UTU^TA$, which consists of three matrix–matrix multiplications. This increases the number of floating point operations by a small amount, and is as stable as the usual algorithm:

Algorithm 14 QR decomposition using Level 3 BLAS (same notation as Algorithm 12)

for $l = 1 : m$
  $k = (l-1) \cdot n_b + 1$
  Use Algorithm 13 to factorize $A^{(l)} = Q_l R_l$,
  Form matrices U_l and T_l from Q_l
  Multiply $X = U_l^T \cdot A(k:m, k+n_b:n)$
  Multiply $X = T_l X$
  Update $A(k:m, k+n_b:n) = A(k:m, k+n_b:n) - UX$

Algorithm 14 is available as subroutine `sgeqrf` from LAPACK (Anderson *et al.*, 1992). Pivoting complicates matters slightly. In conventional column pivoting at step k we need to pivot (permute columns) so the next column of A to be processed has the largest norm in rows k through m of all remaining columns. This cannot be directly combined with blocking as we have just described it, and so instead pivoting algorithms which only look among locally stored columns if possible have been developed (Bischof and Tang, 1991a,b).

Other shared memory algorithms based on Givens rotations have also been developed (Chu, 1988a; Gentleman and Kung, 1981; Sameh, 1985), although these do not seem superior on shared memory machines. It is also possible to use Level 2 and 3 BLAS in the modified Gram–Schmidt algorithm (Gallivan, Jalby, Meier and Sameh,1988).

5.2 Distributed memory algorithms

Just as we could map Algorithm 13 (Gaussian elimination with Level 3 BLAS) to a distributed memory machine with blocked and/or scattered layout by inserting appropriate communication, this can also be done for QR with Level 3 BLAS.

An interesting alternative that works with the same data layouts is based on Givens rotations (Chu, 1988a; Pothen, Jha and Vemulapati, 1987). We consider just the first block column in the block scattered layout, where each of a subset of the processors owns a set of p $r \times r$ subblocks of the block column evenly distributed over the column. Each processor reduces its own $p \cdot r \times r$ submatrix to upper triangular form, spreading the Givens rotations to the right for other processors to apply to their own data. This reduces the processor column to p $r \times r$ triangles, each owned by a different processor. Now there needs to be communication among the processors in the column. Organizing them in a tree, at each node in the tree two processors, each of which owns an $r \times r$ triangle, share their data to reduce to a single $r \times r$ triangle. The requisite rotations are again spread rightward. So in $\log_2 p$ of these steps, the first column has been reduced to a single $r \times r$ triangle, and the algorithm moves on to the next block column.

Other Givens based algorithms have been proposed, but seem to require more communication than this one (Pothen *et al.*, 1987).

6 Eigenproblems and the singular value decomposition

6.1 General comments

The standard serial algorithms for computing the eigendecomposition of a symmetric matrix A, a general matrix B, or the singular value decomposition (SVD) of a general matrix C have the same two-phase structure: apply orthogonal transformations to reduce the matrix to a condensed form, and then apply an iterative algorithm to the condensed form to compute its eigendecomposition or SVD. For the three problems of this section, the condensed forms are symmetric tridiagonal form, upper Hessenberg form and bidiagonal form, respectively. The motivation is that the iteration requires far fewer flops to apply to the condensed form than to the original dense matrix. We discuss reduction algorithms in Section 6.2.

The challenge for parallel computation is that the iteration algorithms for the condensed forms can be much harder to parallellize than the reductions, since they involve nonlinear, sometimes scalar recurrences and/or little opportunity to use the BLAS. For the nonsymmetric eigenproblem, this has led researchers to explore algorithms that are not parallel versions of serial ones. So far none is as stable as the serial one; this is discussed in Section 6.5.

For the symmetric eigenproblem and SVD, the reductions take $\mathcal{O}(n^3)$ flops, but subsequent iterations to find just the eigenvalues or singular values take only $\mathcal{O}(n^2)$ flops; therefore these iterations have not been bottlenecks on serial machines. But on some parallel machines, the reduction algorithms we discuss are so fast that the $\mathcal{O}(n^2)$ part becomes a bottleneck for surprisingly large values of n. Therefore, parallellizing the $\mathcal{O}(n^2)$ part is of interest; we discuss these problems in Section 6.3.

Other approaches to the symmetric eigenproblem and SVD apply to dense matrices instead of condensed matrices. The best known is Jacobi's method. While attractively parallellizable, the convergence rate is sufficiently slower than methods based on tridiagonal and bidiagonal forms that it is seldom competitive. On the other hand, the Jacobi method is sometimes faster and can be much more accurate than these other methods and so still deserves attention; see Section 6.4. Another method that applies to dense symmetric matrices is a variation of the spectral divide-and-conquer method for nonsymmetric matrices, and discussed in Section 6.5.

In summary, reasonably fast and stable parallel algorithms (if not always implementations) exist for the symmetric eigenvalue problem and SVD. However, no highly parallel and stable algorithms currently exist for the nonsymmetric problem; this remains an open problem.

6.2 Reduction to condensed forms

Since the different reductions to condensed forms are so similar, we discuss only reduction to tridiagonal form; for the others see Dongarra *et al.* (1989). At step k we compute a Householder transformation $Q_k = I - 2u_k u_k^T$ so that column k of $Q_k A$ is zero below the first subdiagonal; these zeros are unchanged by forming the similarity transformation $Q_k A Q_k^T$.

Algorithm 15 Reduction to tridiagonal form using Level 2 BLAS (same notation as Algorithm 12)

for $k = 1 : n-2$
  Compute a unit vector u_k so that $(I - 2u_k u_k^T)A(k+2:n, k) = 0$
  Update $A = (I - 2u_k u_k^T)A(I - 2u_k u_k^T)$ by computing
    $w_k = 2Au_k$
    $\gamma_k = w_k^T u_k$
    $v_k = w_k - \gamma_k u_k$
    $A = A - v_k u_k^T - u_k v_k^T$

The major work is updating $A = A - vu_k^T - u_k v^T$, which is a symmetric rank-2 update, a Level 2 BLAS operation. To incorporate Level 3 BLAS, we emulate Algorithm 14 by reducing a single column-block of A to tridiagonal form, aggregating the Householder transformations into a few matrices, and then updating via matrix multiply:

Algorithm 16 Reduction to tridiagonal form using Level 3 BLAS (same notation as Algorithm 12)

for $l = 1 : m$
  $k = (l-1) \cdot n_b + 1$
  Use Algorithm 15 to tridiagonalize the first n_b columns of
    $A(k:n, k:n)$ as follows:
    Do not update all of A at each step, just $A^{(l)}$
    Compute $w_k = 2Au_k$ as $2(A - \sum_{q=1}^{k-1}(v_q u_q^T + u_q v_q^T))u_k$
    Retain $U^{(l)} = [u_1, ..., u_k]$ and $V^{(l)} = [v_1, ..., v_k]$
  Update $A(k:n, k:n) = A(k:n, k:n) - U^{(l)}V^{(l)T} - V^{(l)}U^{(l)T}$

Algorithms 15 and 16 are available from LAPACK (Anderson *et al.*, 1992) as subroutines `ssytd2` and `ssytrf`, respectively. Hessenberg reduction is `sgehrd`, and bidiagonal reduction is `sgebrd`. The mapping to a distributed memory machine follows as with previous algorithms like QR and Gaussian elimination (Dongarra and van de Geijn, 1991).

For parallel reduction of a band symmetric matrix to tridiagonal form, see Bischof and Sun (1992) and Lang (1992).

The initial reduction of a generalized eigenproblem $A - \lambda B$ involves finding orthogonal matrices Q and Z such that QAZ is upper Hessenberg and QBZ is triangular. So far no profitable way has been found to introduce higher level BLAS into this reduction, in contrast to the other reductions previously mentioned. We return to this problem in Section 6.5.

6.3 The symmetric tridiagonal eigenproblem

The basic algorithms to consider are QR iteration, (accelerated) bisection and inverse iteration, and divide-and-conquer. Since the bidiagonal SVD is equivalent to finding the nonnegative eigenvalues of a tridiagonal matrix with zero diagonal (Demmel and Kahan, 1990; Golub and Van Loan, 1989), our comments apply to that problem as well.

6.3.1 QR Iteration

The classical algorithm is QR iteration, which produces a sequence of orthogonally similar tridiagonal matrices $T = T_0, T_1, T_2, \ldots$ converging to diagonal form. The mapping from T_i to T_{i+1} is usually summarized as (1) computing a *shift* σ_i, an approximate eigenvalue; (2) factoring $T_i - \sigma_i I = QR$; and (3) forming $T_{i+1} = RQ + \sigma_i I$. Once full advantage is taken of the tridiagonal form, this becomes a nonlinear recurrence that processes the entries of T_i from one end to the other, and amounts to updating T repeatedly by forming PTP^T, with P a Givens rotation. If the eigenvectors are desired, the Ps are accumulated by forming PV, where V is initially the identity matrix. As it stands this recurrence is not parallellizable, but by squaring the matrix entries it can be changed into a recurrence of the form (1) in Section 2.2 (see Kuck and Sameh, 1977). The numerical stability of this method is not known, but available analyses are pessimistic (Kuck and Sameh, 1977). Furthermore, QR iterations must be done sequentially, with usually just one eigenvalue converging at a time. If one only wants eigenvalues, this method does not appear to be competitive with the alternatives below. When computing eigenvectors, however, it is easy to parallellize: Each processor redundantly runs the entire algorithm updating PTP^T, but only computes n/p of the columns of PV, where p is the number of processors and n is the dimension of T. At the end each processor has n/p components of each eigenvector. Since computing the eigenvectors takes $\mathcal{O}(n^3)$ flops but updating T just $\mathcal{O}(n^2)$, we succeed in parallellizing the majority of the computational work.

6.3.2 Bisection and inverse iteration

One of the two most promising methods is (accelerated) bisection for the eigenvalues, followed by inverse iteration for the eigenvectors (Ipsen and Jessup, 1990; Lo, Phillipe and Sameh, 1987). If T has diagonal entries $a_1, ..., a_n$ and offdiagonals $b_1, ..., b_{n-1}$, then we can count the number of eigenvalues of T less than σ (Golub and Van Loan, 1989).

Algorithm 17 Counting eigenvalues using Sturm sequences (1)

> $count = 0$, $d = 1$, $b_0 = 0$
> for $i = 1 : n$
> $d = a_i - \sigma - b_{i-1}^2/d$
>   if $d < 0$, $count = count + 1$

This nonlinear recurrence may be transformed into a two-term linear recurrence in $p_i = d_1 d_2 \cdots d_i$:

Algorithm 18 Counting eigenvalues using Sturm sequences (2)

> $count = 0$, $p_0 = 1$, $p_{-1} = 0$, $b_0 = 0$
> for $i = 1 : n$
> $p_i = (a_i - \sigma)p_{i-1} - b_{i-1}^2 p_{i-2}$
>   if $p_i p_{i-1} < 0$, $count = count + 1$

In practice, these algorithms need to protected against over/underflow; Algorithm 17 is much easier to protect (Kahan, 1968). Using either of these algorithms, we can count the number of eigenvalues in an interval. The traditional approach is to bisect each interval, say $[\sigma_1, \sigma_2]$, by running Algorithm 17 or 18 at $\eta = (\sigma_1 + \sigma_2)/2$. By continually subdividing intervals containing eigenvalues, we can compute eigenvalue bounds as tight as we like (and roundoff permits). Convergence of the intervals can be accelerated by using a zero-finder such as `zeroin` (Brent, 1973; Lo *et al.*, 1987), Newton's method, Rayleigh quotient iteration (Beattie and Fox, 1989), Laguerre's method, or other methods (Li, Zhang and Sun, 1991). to choose η as an approximate zero of d_n or p_n, i.e. an approximate eigenvalue of T.

There is parallellism both within Algorithm 18 and by running Algorithm 17 or 18 simultaneously for many values of σ. The first kind of parallellism uses parallel prefix as described in (1) in Section 2.2, and so care needs to be taken to avoid over/underflow. The numerical stability of the serial implementations of Algorithms 17 (Kahan, 1968) and 18 (Wilkinson, 1965) is very good, but that of the parallel prefix algorithm is unknown, although numerical experiments are promising (Swarztrauber, 1992). This requires good support for parallel prefix operations, and is not as easy to parallellize as simply having each processor refine different sets of intervals containing different eigenvalues (Demmel, 1992a).

Within a single processor one can also run Algorithm 17 or 18 for many different σ by pipelining or vectorizing (Simon, 1989). These many σ could come from disjoint intervals or from dividing a single interval into more than two small ones (*multi-section*); the latter approach appears to be efficient only when a few eigenvalues are desired, so that there are not many disjoint intervals over which to parallellize (Simon, 1989). Achieving good speedup requires load balancing, and this is not always possible to do by statically assigning work to processors. For example, having the ith processor out of p find eigenvalues $(i-1)n/p$ through in/p results in redundant work at the beginning, as each processor refines the initial large interval containing all the eigenvalues. Even if each processor is given a disjoint interval containing an equal number of eigenvalues to find, the speedup may be poor if the eigenvalues in one processor are uniformly distributed in their interval and all the others are tightly clustered in theirs; this is because there will only be one interval to refine in each clustered interval, and many in the uniform one. This means we need to rebalance the load dynamically, with busy processors giving intervals to idle processors. The best way to do this depends on the communication properties of the machine. Since the load imbalance is severe and speedup poor only for problems that run quickly in an absolute sense anyway, pursuing uniformly good speedup may not always be important. The eigenvalues will also need to be sorted at the end if we use dynamic load balancing.

Given the eigenvalues, we can compute the eigenvectors by using inverse iteration in parallel on each processor. At the end each processor will hold the eigenvectors for the eigenvalues it stores; this is in contrast to the parallel QR iteration, which ends up with the transpose of the eigenvector matrix stored. If we simply do inverse iteration without communication, the speedup will be nearly perfect. However, we cannot guarantee orthogonality of eigenvectors of clustered eigenvalues (Ipsen and Jessup, 1990), which currently seems to require reorthogonalization of eigenvectors within clusters (other methods are under investigation (Parlett, 1992)). We can certainly reorthogonalize against eigenvectors of nearby eigenvalues stored on the same processor without communication, or even against those of neighbouring processors with little communication; this leads to a tradeoff between orthogonality, on the one hand, and communication and load balance, on the other.

Other ways to count the eigenvalues in intervals have been proposed as well (Krishnakumar and Morf, 1986; Swarztrauber, 1992), although these are more complicated than either Algorithm 17 or 18. There have also been generalizations to the band definite generalized symmetric eigenvalue problem (Ma, Patrick and Szyld, 1989).

6.3.3 Cuppen's divide-and-conquer algorithm

The third algorithm is a divide-and-conquer algorithm by Cuppen (Cuppen, 1981), and later analysed and modified by many others (Barlow, 1991; Dongarra and Sorensen, 1987; Gu and Eisenstat, 1992; Ipsen and Jessup, 1990; Jessup and Sorensen, 1989; Sorensen and Tang, 1991). If T is $2n \times 2n$, we decompose it into a sum

$$T = \begin{bmatrix} T_1 & 0 \\ 0 & T_2 \end{bmatrix} + \rho x x^T$$

of a block diagonal matrix with tridiagonal blocks T_1 and T_2, and a rank-1 matrix $\rho x x^T$ which is nonzero only in the four entries at the intersection of rows and columns n and $n+1$. Suppose we now compute the eigendecompositions $T_1 = Q_1 \Lambda_1 Q_1^T$ and $T_2 = Q_2 \Lambda_2 Q_2^T$, which can be done in parallel and recursively. This yields the partial eigendecomposition

$$\begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix} \cdot \left(\begin{bmatrix} \Lambda_1 & 0 \\ 0 & \Lambda_2 \end{bmatrix} + \rho z z^T \right) \cdot \begin{bmatrix} Q_1^T & 0 \\ 0 & Q_2^T \end{bmatrix}$$

where $z = \text{diag}\,(Q_1^T, Q_2^T)x$. So to compute the eigendecomposition of T, we need to compute the eigendecomposition of the matrix $\text{diag}\,(\Lambda_1, \Lambda_2) + \rho z z^T \equiv D + \rho z z^T$, a diagonal matrix plus a rank-1 matrix. We can easily write down the characteristic polynomial of $D + \rho z z^T$, of which the relevant factor is $f(\lambda)$ in the following so-called *secular equation*

$$f(\lambda) \equiv 1 + \rho \sum_{i=1}^{2n} \frac{z_i^2}{d_i - \lambda} = 0.$$

The roots of $f(\lambda) = 0$ are the desired eigenvalues. Assume the diagonal entries d_i of D are sorted in increasing order. After deflating out easy to find eigenvalues (corresponding to tiny z_i or nearly identical d_i) we get a function with guaranteed inclusion intervals $[d_i, d_{i+1}]$ for each zero, and which is also monotonic on each interval. This lets us solve quickly using a Newton-like method (although care must be taken to guarantee convergence (Li, 1992)). The corresponding eigenvector for a root λ_j is then simply given by $(D - \lambda_j I)^{-1} z$. This yields the eigendecomposition $D + \rho z z^T = Q \Lambda Q^T$, from which we compute the full eigendecomposition $T = (\text{diag}\,(Q_1, Q_2)Q)\Lambda(\text{diag}\,(Q_1, Q_2)Q)^T$.

This algorithm, while attractive, proved hard to implement stably. The trouble was that to guarantee the computed eigenvectors were orthogonal, $d_i - \lambda_j$ had to be computed with reasonable relative accuracy, which is not guaranteed even if λ_j is known to high precision; cancellation in $d_i - \lambda_j$ can leave a tiny difference with high relative error. Work by several authors (Barlow, 1991; Sorensen and Tang, 1991) led to the conclusion that λ_i had to be computed to double the input precision in order to determine $d_i - \lambda_i$ accurately. When the input is already in double precision (or whatever is the largest precision supported by the machine), then quadruple is needed, which may be simulated using double, provided double is accurate enough (Dekker, 1971; Priest, 1991). Recently, however, Gu and Eisenstat (1992) have found a new algorithm that makes this unnecessary.

There are two types of parallellism available in this algorithm and both must be exploited to speed up the whole algorithm (Dongarra and Sorensen, 1987; Ipsen and Jessup, 1990). Independent tridiagonal submatrices (such as T_1 and T_2) can obviously be solved in parallel. Initially there are a great many such small submatrices to solve in parallel, but after each secular equation solution, there are half as many submatrices of twice the size. To keep working in parallel, we must find the roots of the different secular equations in parallel; there are equally many roots to find at each level. Also, there is parallellism in the matrix multiplication diag $(Q_1, Q_2) \cdot Q$ needed to update the eigenvectors.

While there is a great deal of parallellism available, there are still barriers to full speedup. First, the speed of the serial algorithm depends strongly on there being a great deal of deflation, or roots of the secular equation that can be computed with little work. If several processors are cooperating to solve a single secular equation, they must either communicate to decide which of their assigned roots were deflated and to rebalance the work load of finding nontrivial roots, or else not communicate and risk a load imbalance. This is the same tradeoff as for the bisection algorithm, except that rebalancing involves more data movement (since eigenvectors must be moved). If it turns out, as with bisection, that load imbalance is severe and speedup poor only when the absolute run time is fast anyway, then dynamic load balancing may not be worth it. The second barrier to full speedup is simply the complexity of the algorithm, and the need to do many different kinds of operations in parallel, including sorting, matrix multiplication, and solving the secular equation. The current level of parallel software support on many machines can make this difficult to implement well.

6.4 Jacobi's method for the symmetric eigenproblem and SVD

Jacobi's method has been used for the nonsymmetric eigenproblem, the symmetric eigenproblem, the SVD, and generalizations of these problems to pairs of matrices (Golub and Van Loan, 1989). It works by applying a series of Jacobi rotations (a special kind of Givens rotation) to the left and/or right of the matrix in order to drive it to a desired canonical form, such as the diagonal form for the symmetric eigenproblem. These Jacobi rotations, which affect only two rows and/or columns of the matrix, are chosen to solve the eigenproblem associated with those two rows and/or columns (this is what makes Jacobi rotations special). By repeatedly solving all 2×2 subproblems of the original, one eventually solves the entire problem. The Jacobi method works reliably on the symmetric eigenvalue problem and SVD, and less so on the nonsymmetric problem. We will consider only the symmetric problem and SVD in this section, and the nonsymmetric Jacobi later.

Until recently Jacobi methods were of little interest on serial machines because they are usually several times slower than QR or divide-and-conquer schemes, and seemed to have the same accuracy. Recently, however, it has been shown that Jacobi's method can be much more accurate than QR in certain cases (Deichmoller, 1991; Demmel and Veselić, 1992; Slapničar, 1992), which makes it of some value on serial machines.

It has also been of renewed interest on parallel machines because of its inherent parallellism: Jacobi rotations can be applied in parallel to disjoint pairs of rows and/or columns of the matrix, so a matrix with n rows and/or columns can have $\lfloor n/2 \rfloor$ Jacobi rotations applied simultaneously (Brent and Luk, 1985). The question remains of the order in which to apply the simultaneous rotations to achieve quick convergence. A number of good parallel orderings have been developed and shown to have the same convergence properties as the usual serial implementations (Luk and Park, 1989; Shroff and Schreiber, 1989); we illustrate one here in the following diagram (P1–P4 denotes Processor 1–Processor 4). Assume we have distributed $n = 8$ columns on $p = 4$ processors, two per processor. We may leave one column fixed, and 'rotate' the others so that after $n-1$ steps all possible pairs of columns have simultaneously occupied a single processor, so they could have a Jacobi rotation applied to them:

	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
P1:	1,8	1,7	1,6	1,5	1,4	1,3	1,2
P2:	2,7	8,6	7,5	6,4	5,3	4,2	3,8
P3:	3,6	2,5	8,4	7,3	6,2	5,8	4,7
P4:	4,5	3,4	2,3	8,2	7,8	6,7	5,6

This is clearly easiest to apply when we are applying Jacobi rotations only to columns of the matrix, rather than to both rows and columns. Such a one-sided Jacobi is natural when computing the SVD (Hari and Veselić, 1987), but requires some preprocessing for the symmetric eigenproblem (Demmel and Veselić, 1992; Slapničar, 1992); for example, in the symmetric positive definite case one can perform Cholesky on A to obtain $A = LL^T$, apply one-sided Jacobi on L or L^T to get its (partial) SVD, and then square the singular values to get the eigenvalues of A. It turns out it accelerates convergence to do the Cholesky decomposition with pivoting, and then apply Jacobi to the columns of L rather than the columns of L^T (Demmel and Veselić, 1992). It is possible to use the symmetric-indefinite decomposition of an indefinite symmetric matrix in the same way (Slapničar, 1992).

Jacobi done in this style is a fine-grain algorithm, operating on pairs of columns, and so cannot exploit higher level BLAS. One can instead use block Jacobi algorithms (Bischof, 1989; Shroff and Schreiber, 1989), which work on blocks, and apply the resulting orthogonal matrices to the rest of the matrix using more efficient matrix–matrix multiplication.

6.5 The nonsymmetric eigenproblem

Five kinds of parallel methods for the nonsymmetric eigenproblem have been investigated:

1. Hessenberg QR iteration (Bai and Demmel, 1989; Davis, Funderlic and Geist, 1987; Dubrulle, 1991; Geist and Davis, 1990; Stewart, 1987; van de Geijn, 1987; 1989; Watkins, 1992; Watkins and Elsner, 1991);

2. Reduction to nonsymmetric tridiagonal form (Dongarra, Geist and Romine, 1990; Geist, 1990; 1991; Geist, Lu and Wachspress, 1989);

3. Jacobi's method (Eberlein, 1962; 1987; Paardekooper, 1989; Sameh, 1971; Shroff, 1991; Stewart, 1985; Veselić, 1979);

4. Hessenberg divide-and-conquer (Chu, 1988b; Chu, Li and Sauer, 1988; Dongarra and Sidani, 1991; Li and Zeng, 1992; Li, Zeng and Cong, 1992; Zeng, 1991);

5. Spectral divide-and-conquer (Bai and Demmel, 1992; Lin and Zmijewski, 1991; Malyshev, 1991).

In contrast to the symmetric problem or SVD, no guaranteed stable and highly parallel algorithm for the nonsymmetric problem exists. As described in Section 6.2, reduction to Hessenberg form can be done efficiently, but so far it has been much harder to deal with a Hessenberg matrix (Dubrulle, 1991; Jessup, 1991). [4]

6.5.1 Hessenberg QR iteration

Parallelizing Hessenberg QR is attractive because it would yield an algorithm that is as stable as the quite acceptable serial one. Unfortunately, doing so involves some of the same difficulties as tridiagonal QR: one is faced with either fine-grain synchronization or larger block operations that execute more quickly but also do much more work without accelerating convergence much. The serial method computes one or two shifts from the bottom right corner of the matrix, and then processes the matrix from the upper left by a series of row and column operations (this processing is called *bulge chasing*). One way to introduce parallellism is to spread the matrix across the processors, but communication costs may exceed the modest computational costs of the row and column operations (Davis *et al.*, 1987; Geist and Davis, 1990; Stewart, 1987; van de Geijn and Hudson, 1989; van de Geijn, 1987). Another way to introduce parallellism is to compute $k > 2$ shifts from the bottom corner of the matrix (the eigenvalues of the bottom right $k \times k$ matrix, say), which permits us to work on k rows and

[4] As noted in Section 6.2, we cannot even efficiently reduce to condensed form for the generalized eigenproblem $A - \lambda B$.

columns of the matrix at a time using Level 2 BLAS (Bai and Demmel, 1989). Asymptotic convergence remains quadratic (Watkins and Elsner, 1991). The drawbacks to this scheme are twofold. First, any attempt to use Level 3 BLAS introduces rather small (hence inefficient) matrix–matrix operations, and raises the operation count considerably. Second, the convergence properties degrade significantly, resulting in more overall work as well (Dubrulle, 1991). As a result, speedups have been extremely modest. This routine is available in LAPACK as `shseqr` (Anderson *et al.*, 1992).

Yet another way to introduce parallellism into Hessenberg QR is to pipeline several bulge chasing steps (van de Geijn, 1987; Watkins, 1992; Watkins and Elsner, 1991). If we have several shifts available, then as soon as one bulge chase is launched from the upper left corner, another one may be launched, and so on. Since each bulge chase operates on only two or three adjacent rows and columns, we can potentially have $n/2$ or $n/3$ bulge chasing steps going on simultaneously on disjoint rows (and columns). The problem is that in the serial algorithm, we have to wait until an entire bulge chase has been completed before computing the next shift; in the parallel case we cannot wait. Therefore, we must use 'out-of-date' shifts to have enough available to start multiple bulge chases. This destroys the usual local quadratic convergence, but it remains superlinear (van de Geijn, 1987). It has been suggested that choosing the eigenvalues of the bottom right $k \times k$ submatrix may have superior convergence to just choosing a sequence from the bottom 1×1 or 2×2 submatrices (Watkins, 1992). Parallelism is still fine-grain, however.

6.5.2 Reduction to nonsymmetric tridiagonal form

This approach begins by reducing B to nonsymmetric tridiagonal form with a (necessarily) nonorthogonal similarity, and then finding the eigenvalues of the resulting nonsymmetric tridiagonal matrix using the tridiagonal LR algorithm (Dongarra *et al.*, 1990; Geist, 1990; 1991; Geist *et al.*, 1989). This method is attractive because finding eigenvalues of a tridiagonal matrix (even nonsymmetric) is much faster than for a Hessenberg matrix (Wilkinson, 1965). The drawback is that reduction to tridiagonal form may require very ill conditioned similarity transformations, and may even break down (Parlett, 1992). Breakdown can be avoided by restarting the process with different initializing vectors, or by accepting a 'bulge' in the tridiagonal form. This happens with relatively low probability, but keeps the algorithm from being fully reliable. The current algorithms pivot at each step to maintain and monitor stability, and so can be converted to use Level 2 and Level 3 BLAS in a manner analogous to Gaussian elimination with pivoting. This algorithm illustrates how one can trade off numerical stability for speed. Other nonsymmetric eigenvalue algorithms we discuss later make this tradeoff as well.

6.5.3 Jacobi's method

As with the symmetric eigenproblem, nonsymmetric Jacobi methods solve a sequence of 2×2 eigenvalue subproblems by applying 2×2 similarity transformations to the matrix. There are two basic kinds of transformations used. Methods that use only orthogonal transformations maintain numerical stability and converge to Schur canonical form, but converge only linearly at best (Eberlein, 1987; Stewart, 1985). If nonorthogonal transformations are used, one can try to drive the matrix to diagonal form, but if it is close to having a nontrivial Jordan block, the required similarity transformation will be very ill conditioned and so stability is lost. Alternatively, one can try to drive the matrix to be *normal* ($AA^T = A^T A$), at which point an orthogonal Jacobi method can be used to drive it to diagonal form; this still does not get around the problem of (nearly) nontrivial Jordan blocks (Eberlein, 1962; Paardekooper, 1989; Sameh, 1971; Shroff, 1991; Veselić, 1979). On the other hand, if the matrix has distinct eigenvalues, asymptotic quadratic convergence is achieved (Shroff, 1991). Using n^2 processors arranged in a mesh, these algorithms can be implemented in time $\mathcal{O}(n \log n)$ per sweep. Again, we trade off control over numerical stability for speed (of convergence).

6.5.4 Hessenberg divide-and-conquer

The divide-and-conquer algorithms we consider here involve setting a middle subdiagonal entry of the original upper Hessenberg matrix H to zero, resulting in a block upper Hessenberg matrix S. The eigenproblems for the two Hessenberg matrices on the diagonal of S can be solved in parallel and recursively. To complete the algorithm, one must merge the eigenvalues and eigenvectors of the two halves of S to get the eigendecomposition of H. Two ways have been proposed to do this: homotopy continuation and Newton's method. Parallelism lies in having many Hessenberg submatrices whose eigendecompositions are needed, in being able to solve for n eigenvalues simultaneously, and in the linear algebra operations needed to find an individual eigenvalue. The first two kinds of parallellism are analogous to those in Cuppen's method (Section 6.3.3). The main drawback of these methods is loss of guaranteed stability and/or convergence. Newton's method may fail to converge, and both Newton and homotopy may appear to converge to several copies of the same root without any easy way to tell if a root has been missed, or if the root really is multiple. The subproblems produced by divide-and-conquer may be much more ill conditioned than the original problem. These drawbacks are discussed in Jessup (1991).

Homotopy methods replace the original Hessenberg matrix H by the one-parameter linear family $H(t) = tS + (1-t)H$, $0 \leq t \leq 1$. As t increases from 0 to 1, the eigenvalues (and eigenvectors) trace out curves connecting the eigenvalues of S to the desired ones of H. The numerical method follows these curves by standard curve-following schemes, predicting the position of a nearby point on the curve using the derivative of the eigenvalue with respect to t, and then correcting its predicted value using Newton's method.

Two schemes have been investigated. The first (Li *et al.*, 1992) follows eigenvalue/eigenvector pairs. The homotopy function is $h(z, \lambda, t) = [(H(t)z - \lambda z)^T, \|z\|_2^2 - 1]^T$, i.e. the homotopy path is defined by choosing $z(t)$ and $\lambda(t)$ so that $h(z(t), \lambda(t), t) = 0$ along the path. The simplicity of the homotopy means that over 90% of the paths followed are simple straight lines that require little computation, resulting in a speed up of a factor of up to 2 over the serial QR algorithm. The drawbacks are lack of stability and convergence not being guaranteed. For example, when homotopy paths get very close together, one is forced to take smaller steps (and so converge more slowly) during the curve following. Communication is necessary to decide if paths get close. And as mentioned previously, if two paths converge to the same solution, it is hard to tell if the solution really is a multiple root or if some other root is missing. A different homotopy scheme uses only the determinant to follow eigenvalues (Li and Zeng, 1992; Zeng, 1991); here the homotopy function is simply $\det(H(t) - \lambda I)$. Evaluating the determinant of a Hessenberg matrix costs only a triangular solve and an inner product, and therefore is efficient. It shares similar advantages and disadvantages as the previous homotopy algorithm.

Alternatively, one can use Newton's method to compute the eigendecomposition of H from S (Dongarra and Sidani, 1991). The function to which one applies Newton's method is $f(z, \lambda) = [(Hz - \lambda z)^T, e^T z - 1]^T$, where e is a fixed unit vector. The starting values for Newton's method are obtained from the solutions to S.

6.5.5 Spectral divide-and-conquer

A completely different way to divide-and-conquer a matrix is using a projection on part of the spectrum. It applies to a dense matrix B. Suppose Q_1 is an $n \times m$ orthogonal matrix spanning a right invariant subspace of B, and Q_2 is an $n \times (n-m)$ matrix constructed so that $Q = [Q_1, Q_2]$ is square and orthogonal. Then Q deflates B as follows:

$$Q^T B Q = \begin{bmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{bmatrix} .$$

Note that this is equivalent to having Q_2 span a left invariant subspace of B. The eigenvalues of B_{11} are those corresponding to the invariant subspace spanned by Q_1. Provided we can construct Q_1 effectively, we can use this to divide-and-conquer the matrix.

Of course Hessenberg QR iteration fits into this framework, with Q_2 being $n \times 1$ or $n \times 2$, and computed by (implicit) inverse iteration applied to $B - \sigma I$, where σ is a shift. Just splitting so that B_{22} is 1×1 or 2×2 does not permit much parallellism, however; it would be better to split the matrix nearer the middle. Also, it would be nice to be able to split off just that part of the spectrum of interest to the user, rather than computing all eigenvalues as these methods must all do.

There are several approaches to computing Q. They may be motivated by analogy to Hessenberg QR, where Q is the orthogonal part of the QR factorization $QR = B - \sigma I$. If σ is an exact eigenvalue, so that $B - \sigma I$ is singular, then the last column of Q is (generically) a left eigenvector for 0. One can then verify that the last row of $Q^T(B - \sigma I)Q$ is zero, so that we have deflated the eigenvalue at σ. Now consider a more general function $f(B)$; in principle any (piecewise) analytic function will do. Then the eigenvalues of $f(B)$ are just f evaluated at the eigenvalues of B, and $f(B)$ and B have (modulo Jordan blocks) the same eigenvectors. Suppose that the rank of $f(B)$ is $m < n$, so that $f(B)$ has (at least) $n - m$ zero eigenvalues. Factorize $QR = f(B)$. Then the last $n - m$ columns of Q (generally) span the left null space of $f(B)$, i.e. a left invariant subspace of $f(B)$ for the zero eigenvalue. But this is also a left invariant subspace of B so we get

$$Q^T f(B) Q = \begin{bmatrix} \hat{B}_{11} & \hat{B}_{12} \\ 0 & 0 \end{bmatrix} \text{ and } Q^T B Q = \begin{bmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{bmatrix}.$$

The problem thus becomes finding functions $f(B)$ that are easy to evaluate and have large null spaces, or which map selected eigenvalues of B to zero. One such function f is the *sign function* (Bai and Demmel, 1992; Howland, 1983; Kenney and Laub, 1991; Lin and Zmijewski, 1991; Robert, 1980; Stickel, 1991) which maps points with positive real part to $+1$ and those with negative real part to -1; adding 1 to this function then maps eigenvalues in the right half plane to 2 and in the left plane to 0, as desired.

The only operations we can easily perform on (dense) matrices are multiplication and inversion, so in practice f must be a rational function. A globally, asymptotically quadratically convergent iteration to compute the sign function of B is $B_{i+1} = (B_i + B_i^{-1})/2$ (Howland, 1983; Robert, 1980; Stickel, 1991); this is simply Newton's method applied to $B^2 = I$, and can also be seen to be equivalent to repeated squaring (the power method) of the Cayley transform of B. It converges more slowly as eigenvalues approach the imaginary axis, and is in fact nonconvergent if there are imaginary eigenvalues, as may be expected since the sign function is discontinuous there. Other higher order convergent schemes exist, but they can be more expensive to implement as well (Kenney and Laub, 1991; Pandey, Kenney and Laub, 1990). Another scheme that divides the spectrum between the eigenvalues inside and outside the unit circle is given in Malyshev (1991).

If the eigenvalues are known to be real (as when the matrix is symmetric), we need only construct a function f that maps different parts of the real axis to 0 and 1 instead of the entire left and right half planes. This simplifies both the computation of $f(B)$ and the extraction of its null space. See Auslander and Tsao (1992), Bischof and Sun (1992) and Lederman, Tsao and Turnbull (1992) for details.

Of course, we wish to split not just along the imaginary axis or unit circle but other boundaries as well. By shifting the matrix and multiplying by a complex number $e^{i\theta}$ one can split along an arbitrary line in the complex plane, but at the cost of introducing complex arithmetic. By working on a shifted and squared real matrix, one can divide along lines at an angle of $\pi/4$ and retain real arithmetic (Bai and Demmel, 1992; Howland, 1983; Stickel, 1991).

This method is promising because it allows us to work on just that part of the spectrum of interest to the user. It is stable because it applies only orthogonal transformations to B. On the other hand, if it is difficult to find a good place to split the spectrum, convergence can be slow, and the final approximate invariant subspace inaccurate. At this point, iterative refinement could be used to improve the factorization (Demmel, 1987). These methods apply to the generalized nonsymmetric eigenproblem as well (Bai and Demmel, 1992; Malyshev, 1991).

7 Direct methods for sparse linear systems

7.1 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 parallellism 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 parallellism 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 parallellism 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 Heath, Ng and Peyton (1991). 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 19 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 19 Cholesky factorization

for $j = 1,\ n$
    for $k = 1,\ j-1$
        for $i = j,\ n$ $\quad\{\mathrm{cmod}(j,k)\}$
            $a_{ij} = a_{ij} - a_{ik} \cdot a_{jk}$
    $a_{jj} = \sqrt{a_{jj}}$
    for $k = j+1,\ n$ $\quad\{\mathrm{cdiv}(j)\}$
        $a_{kj} = a_{kj}/a_{jj}$

The outer loop in Algorithm 19 is over successive columns of A. The 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 19 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 parallellism that we will explore in detail later.

7.2 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 19, 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., Duff, Erisman and Reid (1986) and George and Liu (1981) 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 neighbours 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

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

and

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

In other words, $\mathrm{Struct}(M_{i*})$ is the sparsity structure of row i of the strict lower triangle of M, while $\mathrm{Struct}(M_{*j})$ 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:

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

Thus, $\mathrm{parent}(j)$ 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 $i = \mathrm{parent}(j)$. 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 analysing and organizing sparse Cholesky factorization are surveyed in Liu (1990). We will illustrate these concepts pictorially in several examples below.

7.3 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:

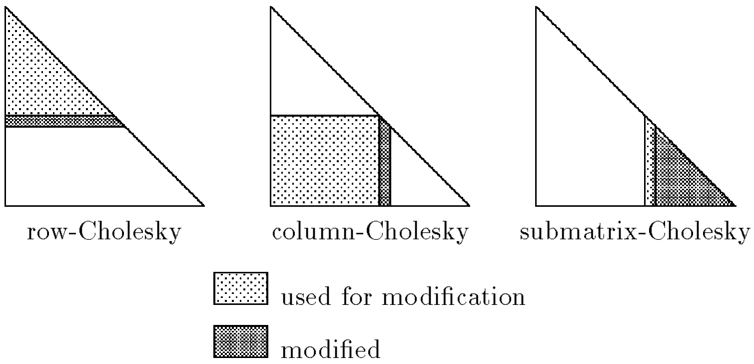

Figure 9: Three forms of Cholesky factorization.

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 9, 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 efficiently during the factorization, and the difficulty in vectorizing or parallellizing 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 20 Sparse column-Cholesky factorization

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

cmod(j, k)
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 (1988) 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 21 Sparse submatrix-Cholesky factorization

for $k = 1,\ n$
    cdiv(k)
    for $j \in \text{Struct}(L_{*k})$
        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 (1988) 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., Duff *et al.* (1986)), 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 (Irons, 1970)). 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 (Ashcraft, Grimes, Lewis, Peyton and Simon, 1987) and on RISC workstations (Rothberg and Gupta, 1989).

7.4 Parallelism in sparse factorization

We now examine in greater detail the opportunities for parallellism 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 (1986) 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 parallellism, 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 parallellism, 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 parallellism for sparse factorization as well. This parallellism 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 parallellism, 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 parallellism 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 parallellism. These measures are only very rough, however, since the medium level parallellism 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 parallellism 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 parallellism, and indeed this is possible (see, e.g., Jess and Kees (1982), Lewis, Peyton and Pothen (1989), Liu (1989)), but such an objective may conflict to some degree with preservation of sparsity. Roughly speaking, sparsity and parallellism are largely compatible, since the large-grain parallellism 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 10 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 factor-

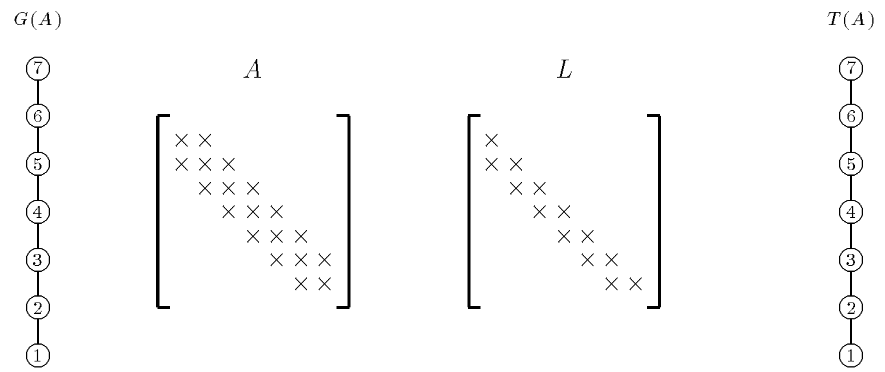

Figure 10: One-dimensional grid and corresponding tridiagonal matrix (left), with Cholesky factor and elimination tree (right).

ization is minimal. However, we see that the elimination tree is simply a chain, and therefore there is no large-grain parallellism available. Each column of L depends on the immediately preceding one, and thus they must be computed sequentially. This behaviour is typical of orderings that minimize the bandwidth of a sparse matrix: they tend to inhibit rather than enhance large-grain parallellism in the factorization. (As previously discussed in Section 4.4, there is in fact little parallellism 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 11 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 (George and Liu, 1989). 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 neighbours 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 11 that L suffers no fill in the new ordering, and the elimination tree now shows some large-grain parallellism. In particular, columns 1 and 2 can be computed simultaneously, then columns 3 and 4, and so on. This twofold parallellism 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 11, we have deliberately broken ties in the most favourable way (with respect to parallellism); the least favourable tie breaking would have reproduced the orig-

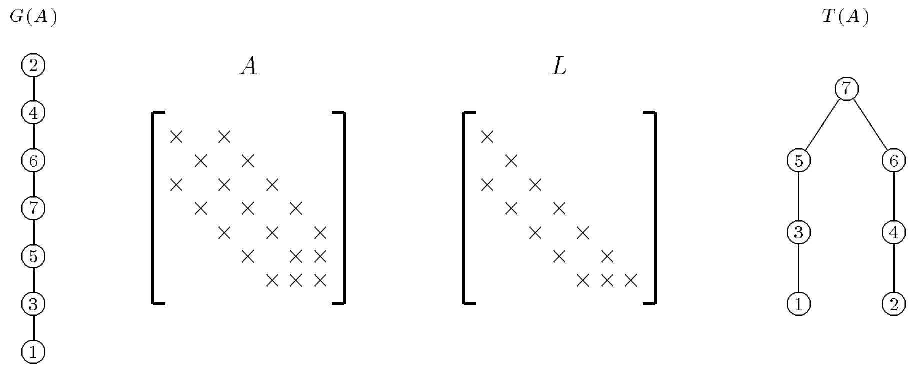

Figure 11: Graph and matrix reordered by minimum degree (left), with corresponding Cholesky factor and elimination tree (right).

inal ordering of Figure 10, resulting in no parallellism. 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 parallellism, but which is taller and less well balanced than our example in Figure 11. Again, this is typical of minimum degree orderings. In view of this property, Liu (1989) 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 12 shows the same mesh again, this time ordered by nested dissection, a divide-and-conquer strategy (George, 1973). 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 12 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 parallellism, and its height has been reduced further. This behaviour 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 parallellism more systematically and produce better balanced elimination trees.

Finally, Figure 13 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 parallellism in tridiagonal and related systems, so we illustrate it

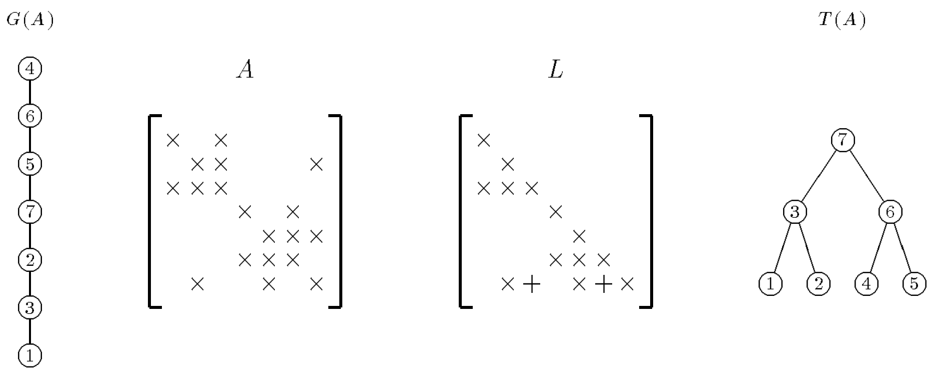

Figure 12: Graph and matrix reordered by nested dissection (left), with corresponding Cholesky factor and elimination tree (right).

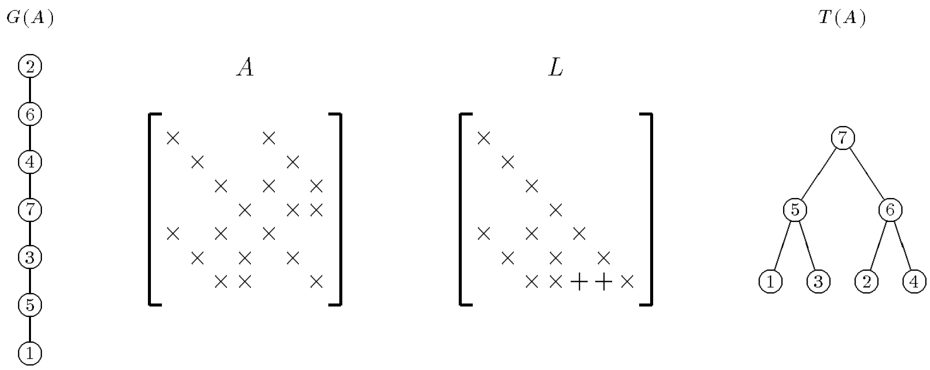

Figure 13: Graph and matrix reordered by odd–even reduction (left), with corresponding Cholesky factor and elimination tree (right).

for the sake of comparison with more general purpose methods. In odd–even reduction (see, e.g., Duff *et al.* (1986)), odd node numbers come before even node numbers, and then this same renumbering is applied recursively within each 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.

7.5 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 George, Heath, Liu and Ng (1986).

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 parallellism 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 neighbouring processors. A good example of this technique is the 'subtree-to-subcube' mapping often used with hypercube multicomputers (George, Heath, Liu and Ng, 1989). 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 (George, Heath, Liu and Ng, 1988). Algorithm 22 runs on each processor, with each responsible for its own subset, *mycols*, of columns.

Algorithm 22 Distributed fan-out sparse Cholesky factorization

```
for j ∈ 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 ≠ ∅
    receive any column of L, say L_{*k}
    for j ∈ mycols ∩ 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 22, 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 22 potentially exploits both the large-grain parallellism characterized by concurrent cdivs and the medium-grain parallellism 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., George, Liu and Ng (1989), Mu and Rice (1992), Zmijewski (1989)), 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 (Ashcraft, Eisenstat and Liu, 1990):

Algorithm 23 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 23 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 23 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 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., Benner, Montry and Weigand (1987), Duff (1986)) and distributed-memory (e.g., Gilbert and Schreiber (1992), Lucas, Blank and Tieman (1987)) 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 Ashcraft, Eisenstat, Liu and Sherman (1990).

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.

8 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 Freund, Golub and Nachtigal (1992). 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 8.1 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 Dongarra *et al.* (1991). In Section 8.2, coarse-grained parallel and data-locality issues of CG will be discussed, while in Section 8.3 the same will be done for GMRES.

8.1 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 parallellized; 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 parallellizable: each processor updates its 'own' segment. The matrix–vector products are often easily parallellized 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 neighbouring blocks. If the number of connections to these neighbouring 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 de Sturler (1991) and Pommerell (1992).

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 Golub and Van Loan (1989), Meijerink and van der Vorst (1977) and Meijerink and van der Vorst (1981). Solving $Kw = r$ leads to solving successively $Lz = r$ and $Uw = z$. These triangular solves lead to recurrence relations that are not easily parallellized. We will now discuss a number of approaches to obtain parallellism in the preconditioning part.

1. *Reordering the computations.* Depending on the structure of the matrix a *frontal approach* may lead to successful parallellism. 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 parallellized 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 Ashcraft and Grimes (1988), Dongarra *et al.* (1991), van der Vorst (1989a) and (1989b)). For coarse-grained parallellism this approach is insufficient. By a similar approach more parallellism can be obtained in three-dimensional situations: the so-called *hyperplane approach* (Schlichting and van der Vorst, 1989; van der Vorst, 1989a; 1989b). 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 parallellism.

2. *Reordering the unknowns.* One may also use a *colouring scheme* for reordering the unknowns, so that unknowns with the same colour are not explicitly coupled. This means that the triangular solves can be parallellized for each colour. Of course, communication is required for couplings between groups of different colours. Simple colouring schemes, like red-black ordering for the five-point discretized Poisson operator, seem to have a negative effect on the convergence behaviour. Duff and Meurant (1989) 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 parallellism can be obtained, however, with so-called incomplete twisted factorizations (Dongarra *et al.*, 1991; van der Vorst, 1987b; van der Vorst, 1989a). Multi-colour schemes with a large number of colours (e.g., 20 to 100) may lead to little or no degradation in convergence behaviour (Doi, 1991), but also to less parallellism. Moreover, the ratio of computation to communication may be more unfavourable.

3. *Forced parallellism.* 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 (Seager, 1986). Again, this may not always reduce the overall solution time, since the effects of increased parallellism are more than undone by an increased number of iteration steps. In order to reduce this

effect, it is suggested in Radicati di Brozolo and Robert (1989) to construct incomplete decompositions on slightly overlapping domains. This requires communication similar to that of matrix–vector products. In Radicati di Brozolo and Robert (1989) results are reported on a six-processor shared memory system (IBM3090), and speedups close to 6 have been observed.

The problems with parallellism in the preconditioner have led to searches for other preconditioners. Often simple diagonal scaling is an adequate preconditioner and this is trivially parallellizable. For results on a Connection Machine, see Berryman, Saltz, Gropp and Mirchandaney (1989). 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 parallellize (Saad, 1985b). 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.* (1976) discuss approximate inversion of A, which from the implementation point of view is equivalent to polynomial preconditioning. In van der Vorst (1982) 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 parallellism (vectorization).

8.2 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 (Freund *et al.*, 1991; Golub and Van Loan, 1989; Hestenes and Stiefel, 1954). 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 parallellize. We have discussed some efforts to obtain more parallellism in the preconditioner in Section 8.1. 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 24 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 $i = 0, 1, 2, \ldots.$
  $p_i = w_i + \beta_{i-1} p_{i-1}$;
  $q_i = Ap_i$;
  $\alpha_i = \rho_i / (p_i, q_i)$
  $x_{i+1} = x_i + \alpha_i p_i$;
  $r_{i+1} = r_i - \alpha_i q_i$;
  if x_{i+1} accurate enough then quit;
  Solve for w_{i+1} in $Kw_{i+1} = r_{i+1}$;
  $\rho_{i+1} = (r_{i+1}, w_{i+1})$;
  $\beta_i = \rho_{i+1} / \rho_i$;
end;

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

Now we consider parallellizing 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 $Kw_{i+1} = r_{i+1}$, 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 (Chronopoulos and Gear, 1989; Meurant, 1984a,b; van der Vorst, 1986) 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 24 there are two such synchronization points, namely the computation of both inner products. Meurant (1984a) (see also Saad (1985b)) 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 (1989).

Algorithm 25 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 $Kw_0 = r_0$;
$s_0 = Aw_0$;

$\rho_0 = (r_0, w_0); \mu_0 = (s_0, w_0);$
$\alpha_0 = \rho_0/\mu_0;$
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 (1989) 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 (1989) propose to improve further the data locality and parallellism 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 Chronopoulos (1991).

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 (Saad, 1988) because of these problems. However, a similar approach, suggested by Chronopoulos and Kim (1990) 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 8.3.

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 26 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 $i = 0, 1, 2, \ldots.$
    $w_i = L^{-T} s$; (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 parallellized 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 synchroniation points in CG while retaining numerical stability appears in D'Azevedo and Romine (1992) and Eijkhout (1992).

8.3 Parallelism and data locality in GMRES

GMRES, proposed by Saad and Schultz (1985), is a CG-like method for solving general nonsingular linear systems $Ax = b$. GMRES minimizes the residual over the Krylov subspace span$[r_0, Ar_0, A^2 r_0, ..., A^i r_0]$, 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 (Golub and Van Loan, 1989). 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 Saad and Schultz (1985)):

Algorithm 27 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;
        $h_{i+1,i} = \|w\|_2$;
        $v_{i+1} = w/h_{i+1,i}$;
    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 Walker (1988). For some applications the additional computation required by Householder orthogonalization is compensated by improved numerical properties: the better orthogonality saves iteration steps. In van der Vorst and Vuik (1991) 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 parallellizable, 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 $v_1, v_2, \ldots, v_j$. In Algorithm 27, 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 (Saad, 1988)). 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 parallellism 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) (Chronopoulos and Kim, 1990). 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 Bai, Hu and Reichel (1991), and Chebychev polynomials in de Sturler (1991).

After generating a suitable starting set, we still have to orthogonalize it. In de Sturler (1991) 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 (1991) 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.* (1991) 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.

9 Iterative methods for eigenproblems

The oldest iterative scheme for determining a few dominant eigenvalues and corresponding eigenvectors of a matrix A is the *power method* (Parlett, 1980):

Algorithm 28 Power method

select x_0 with $||x_0||_2 = 1$
$k = 0$
repeat
    $k = k + 1$
    $y_k = Ax_{k-1}$
    $\lambda = ||y_k||_2$
    $x_k = y_k/\lambda$
until x_k converges

If the eigenvalue of A of maximum modulus is well separated from the others, then x_k converges to the corresponding eigenvector and λ converges to the modulus of the eigenvalue. The power method has been superseded by more efficient techniques. However, the method is still used in the form of *inverse iteration* for the rapid improvement of eigenvalue and eigenvector approximations obtained by other schemes. In inverse iteration, the line '$y_k = Ax_{k-1}$' is replaced by 'Solve for y_k in $Ay_k = x_{k-1}$'. Most of the computational effort will be required by this operation, whose (iterative) solution we discussed in Section 8.

All operations in the power method are easily parallellizable, except possibly for the convergence test. There is only one synchronization point: x_k can be computed only after the reduction operation for λ has completed. This synchronization could be avoided by changing the operation $y_k = Ax_{k-1}$ to $y_k = Ay_{k-1}$ (assuming $y_0 = x_0$). This means λ would change every iteration by a factor which converges to the maximum modulus of the eigenvalues of A, and so risks overflow or underflow after enough steps.

The power method constructs basis vectors x_k for the Krylov subspace determined by x_0 and A. It is faster and more accurate to keep all these vectors and then determine stationary points of the Rayleigh quotient over the Krylov subspace. In order to minimize work and improve numerical stability, we compute an orthonormal basis for the Krylov subspace. This can be done by either short recurrences or long recurrences. The short (three-term) recurrence is known as the *Lanczos method*. When A is symmetric this leads to an algorithm with can efficiently compute many, if not all, eigenvalues and eigenvectors (Parlett, 1980).

In fact, the CG method (and Bi-CG) can be viewed as a solution process on top of Lanczos. The long recursion process is known as *Arnoldi's method* (Arnoldi, 1951), which we have seen already as the underlying orthogonalization procedure for GMRES. Not surprisingly, a short discussion on parallellizing the Lanczos and Arnoldi methods would have much in common with our earlier discussions of CG and GMRES.

9.1 The Lanczos method

The Lanczos algorithm is described by the following scheme (Parlett (1980)):

Algorithm 29 Lanczos method

select $r_0 \neq 0$; $q_0 = 0$
$k = 0$
repeat
    $k = k + 1$
    $\beta_{k-1} = \|r_{k-1}\|_2$
    $q_k = r_{k-1}/\beta_{k-1}$
    $u_k = Aq_k$

$s_k = u_k - \beta_{k-1} q_{k-1}$
$\alpha_k = (q_k, s_k)$
$r_k = s_k - \alpha_k q_k$
until the eigenvalues of T_k converge (see below)

The q_ks can be saved in secondary storage (they are required for backtransformation of the so-called Ritz vectors below).

The α_m and β_m, for $m = 1, 2, ..., k$, form a tridiagonal matrix T_k:

$$T_k = \begin{pmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \cdot & \cdot & \\ & & \cdot & \cdot & \beta_{k-1} \\ & & & \beta_{k-1} & \alpha_k \end{pmatrix}.$$

The eigenvalues and eigenvectors of T_k are called the *Ritz values* and *Ritz vectors*, respectively, of A with respect to the Krylov subspace of dimension k. The Ritz values converge to eigenvalues of A as k increases, and after backtransformation with the q_ms, the corresponding Ritz vectors approximate eigenvectors of A. For improving these approximations one might consider inverse iteration.

The parallel properties of the Lanczos scheme are similar to those of CG. On distributed memory machines there are two synchronization points caused by the reduction operations for β_{k-1} and α_k. The first synchronization point can, at the cost of n additional flops, be removed by delaying the normalization of the n-vector r_{k-1}. This would lead to the following sequence of statements:

$$u_k = Ar_{k-1}; \quad q_k = r_{k-1}/\beta_{k-1}; \quad s_k = u_k/\beta_{k-1} - \beta_{k-1} q_{k-1} \ .$$

In this approach, the reduction operation for β_{k-1} can be overlapped with computing Ar_{k-1}. Much in the same spirit as the approach suggested for CG by Chronopoulos and Gear (see algorithm 25), the synchronization point caused by the reduction operation for α_k can be removed by computing As_k right after, and in overlap with, this operation. In that case we reconstruct Ar_{k-1} from recurrence relations for r_{k-1}. This increases the operation count by another n flops, and, even more serious, leads to a numerically less stable algorithm. The Ritz values and Ritz vectors can be computed in parallel by techniques discussed in Section 6. For small k there will not be much to do in parallel, but we also need not compute the eigensystem of T_k for each k, nor check convergence for each k. An elegant scheme for tracking convergence of the Ritz values is discussed in Parlett and Reid (1981). If the Ritz value $\theta_j^{(k)}$, i.e. the jth eigenvalue of T_k, is acceptable as an approximation to some eigenvalue of A, then an approximation $e_j^{(k)}$ to the corresponding eigenvector of A is given by

$$e_j^{(k)} = Q_k y_j^{(k)}, \tag{4}$$

where $y_j^{(k)}$ is the jth eigenvector of T_k, and $Q_k = [q_1, q_2, ..., q_k]$. This is easy to parallellize.

As with CG, one may attempt to improve parallellism in Lanczos by combining s steps in the orthogonalization step. However, the eigensystem of T_k is very sensitive to loss of orthogonality in the q_m vectors. For the standard Lanczos method this loss of orthogonality goes hand in hand with the convergence of Ritz values and leads to multiple eigenvalues of T_k (see Paige (1976) and Parlett (1980)), and so can be accounted for, for instance, by selective reorthogonalization, for which the converged Ritz vectors are required (Parlett, 1980). It is as yet unknown how rounding errors will affect the s step approach, and whether that may lead to inaccurate eigenvalue approximations.

9.2 The Arnoldi method

The Arnoldi algorithm is just the orthogonalization scheme we used before in GMRES:

Algorithm 30 Arnoldi's method

```
w is an initial vector with ||w||_2 ≠ 0;
v_1 = w/||w||_2;
k = 0;
repeat
    k = k + 1;
    w = A v_k;
    for i = 1, ..., k              orthogonalization of w
        h_{i,k} = (w, v_i);        against vs, by modified
        w = w - h_{i,k} v_i;       Gram–Schmidt process
    end i;
    h_{k+1,k} = ||w||_2;
    v_{k+1} = w/h_{k+1,k};
until the eigenvalues of H̄_k converge (see below)
```

The elements $h_{i,j}$ computed after k steps build an upper Hessenberg matrix H_k of dimension $(k+1) \times k$. The eigenvalues and eigenvectors of the upper $k \times k$ part $\bar{H}_k$ of H_k are the Ritz values and Ritz vectors of A with respect to the k dimensional Krylov subspace generated by A and w. The Ritz values $\theta_j^{(k)}$ of $\bar{H}_k$ converge to eigenvalues of A, and the corresponding Ritz vectors $y_j^{(k)}$ can then be backtransformed to approximate eigenvectors of e_j of A via

$$e_j = V_k y_j^{(k)},$$

where $V_k = [v_1, v_2, ..., v_k]$. The parallel solution of the eigenproblem for a Hessenberg matrix is far from easy, for a discussion see Section 6. Normally it is assumed that the order n of A is much larger than the number of Arnoldi steps k. In this case it may be acceptable to solve the eigenproblem for H_k on a single processor. In order to limit k, it has been proposed to carry out the Arnoldi process with a fixed small value for k (a few times larger than the desired number of dominant eigenvalues m), and to repeat the process, very much in the same manner as GMRES(k). At each repetition of the process, it is started with a w that is taken as a combination of the Ritz vectors corresponding to the m dominant Ritz values of the previous cycle. Hopefully, the subspace of m dominant Ritz vectors converges to an invariant subspace of dimension m. This process is known as *subspace iteration*, a generalization of the power method, for details see Saad (1980) and Saad (1985a). For a description, as well as a discussion of its performance on the Connection Machine, see Petiton (1992).

A different approach for computing an invariant subspace of order m, based on Arnoldi's process, is discussed in Sorensen (1992). Here one starts with m steps of Arnoldi to create an initial approximation of the invariant subspace of dimension m corresponding to m desired eigenvalues, say the m largest eigenvalues in modulus. Then this subspace is repeatedly expanded by p new vectors, using the Arnoldi process, so that the $m+p$ vectors form a basis for a $m+p$ dimensional Krylov subspace. This information is compressed to the first m vectors of the subset, by a QR algorithm that drives the residual in the projected operator to a small value using p shifts (usually the p unwanted Ritz values of the projected operator). If this expansion and compression process is repeated i times, then the computed m dimensional subspace will be a subset of the $m+i\cdot p$ dimensional Krylov subspace one would get without compression. The hope is that by compressing well, the intersection of the desired invariant subspace with the final m dimensional subspace is close to the intersection with the larger $m+i\cdot p$ dimensional subspace. The benefit of this method is in limiting storage and time spent on the projected Hessenberg eigenproblems to depend on $m+p$ rather than $m+i\cdot p$. An advantage over the previous approach (*subspace iteration*) is in the implicit construction of a suitable starting vector for the $m+p$ dimensional Krylov subspace. For a basis of this subspace only p matrix vector products have to be evaluated for each iteration cycle. In these approaches the eigendecomposition of the projected Hessenberg matrices is still the hardest step to parallellize.

We do not know of successful attempts to combine s successive Krylov subspace vectors v, Av, A^2v, ..., $A^{s-1}v$ (as was proposed in combination with GMRES). In the case of subspace iteration numerical instability may not be as severe as for the Lanczos process.

References

A. Aho, J. Hopcroft and J. Ullman (1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley (New York).

E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen (1992), *LAPACK Users' Guide, Release 1.0*, SIAM (Philadelphia).

E. Anderson and J. Dongarra (1990), 'Evaluating block algorithm variants in LAPACK', Computer Science Dept. Technical Report CS-90-103, University of Tennessee, Knoxville, TN (LAPACK Working Note 19).

ANSI/IEEE, New York, (1985), *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 edition (New York).

W. E. Arnoldi (1951), 'The principle of minimized iteration in the solution of the matrix eigenproblem', *Quart. Appl. Math.* **9**, 17–29.

C. Ashcraft, S.C. Eisenstat and J.W.-H. Liu (1990), 'A fan-in algorithm for distributed sparse numerical factorization', *SIAM J. Sci. Stat. Comput.* **11**, 593–599.

C. Ashcraft, S.C. Eisenstat, J.W.-H. Liu and A.H. Sherman (1990), 'A comparison of three column-based distributed sparse factorization schemes', Technical Report YALEU/DCS/RR-810, Dept of Computer Science, Yale University, New Haven, CT.

C. Ashcraft and R. Grimes (1988), 'On vectorizing incomplete factorizations and SSOR preconditioners', *SIAM J. Sci. Stat. Comput.* **9**, 122–151.

C. Ashcraft, R. Grimes, J. Lewis, B. Peyton and H. Simon (1987), 'Progress in sparse matrix methods for large linear systems on vector supercomputers', *Int. J. Supercomput. Appl.* **1**(4), 10–30.

L. Auslander and A. Tsao (1992), 'On parallelizable eigensolvers', *Adv. Appl. Math.* **13**, 253–261.

I. Babuska (1972), 'Numerical stability in problems of linear algebra', *SIAM J. Numer. Anal.* **9**, 53–77.

Z. Bai and J. Demmel (1989), 'On a block implementation of Hessenberg multishift QR iteration', *Int. J. High Speed Comput.* **1**(1), 97–112 (also LAPACK Working Note 8).

Z. Bai and J. Demmel (1992), 'Design of a parallel nonsymmetric eigenroutine toolbox', Computer Science Dept preprint, University of California, Berkeley, CA.

Z. Bai, D. Hu and L. Reichel (1991), 'A Newton basis GMRES implementation', Technical Report 91-03, University of Kentucky.

D. H. Bailey, K. Lee and H. D. Simon (1991), 'Using Strassen's algorithm to accelerate the solution of linear systems', *J. Supercomput.* **4**, 97–371.

J. Barlow (1991), 'Error analysis of update methods for the symmetric eigenvalue problem', to appear in *SIAM J. Math. Anal. Appl.* (Tech Report CS-91-21, Computer Science Department, Penn State University.)

C. Beattie and D. Fox (1989), 'Localization criteria and containment for Rayleigh quotient iteration', *SIAM J. Math. Anal. Appl.* **10** (1), 80–93.

K. Bell, B. Hatlestad, O. Hansteen and P. Araldsen (1973), *NORSAM—A programming system for the finite element method, User's Manual, Part I, General description.* Institute for Structural Analysis, NTH, N-7034 Trondheim, Norway.

R. Benner, G. Montry and G. Weigand (1987), 'Concurrent multifrontal methods: shared memory, cache, and frontwidth issues', *Int. J. Supercomput. Appl.* **1** (3), 26–44.

H. Berryman, J. Saltz, W. Gropp and R. Mirchandaney (1989), 'Krylov methods preconditioned with incompletely factored matrices on the CM-2', Technical Report 89-54, NASA Langley Research Center, ICASE, Hampton, VA.

D. Bertsekas and J. Tsitsiklis (1989), *Parallel and Distributed Comptutation: Numerical Methods*, Prentice Hall (New York).

C. Bischof (1989), 'Computing the singular value decomposition on a distributed system of vector processors', *Parallel Comput.* **11** 171–186.

C. Bischof and X. Sun (1992), 'A divide and conquer method for tridiagonalizing symmetric matrices with repeated eigenvalues', MCS Report P286-0192, Argonne National Lab.

C. Bischof and P. Tang (1991a), 'Generalized incremental condition estimation', Computer Science Dept, Technical Report CS-91-132, University of Tennessee, Knoxville, TN (LAPACK Working Note 32).

C. Bischof and P. Tang (1991b), 'Robust incremental condition estimation', Computer Science Dept. Technical Report CS-91-133, University of Tennessee, Knoxville, TN (LAPACK Working Note 33).

R. P. Brent (1973), *Algorithms for Minimization Without Derivatives*, Prentice-Hall (New York).

R. Brent and F. Luk (1985), 'The solution of singular value and symmetric eigenvalue problems on multiprocessor arrays', *SIAM J. Sci. Stat. Comput.* **6**, 69–84.

J. Brunet, A. Edelman and J. Mesirov (1990), 'An optimal hypercube direct n-body solver on the connection machine', in: *Proceedings of Supercomputing '90*, IEEE Computer Society Press (New York), 748–752.

D. Calvetti, J. Petersen and L. Reichel (1991), 'A parallel implementation of the GMRES method', Technical Report ICM-9110-6, Institute for Computational Mathematics, Kent, OH.

L. Cannon (1969), 'A cellular computer to implement the Kalman filter algorithm', PhD thesis, Montana State University, Bozeman, MN.

S. C. Chen, D. J. Kuck and A. H. Sameh (1978), 'Practical parallel band triangular system solvers', *ACM Trans. Math. Software* **4**, 270–277.

A. T. Chronopoulos (1991), 'Towards efficient parallel implementation of the CG method applied to a class of block tridiagonal linear systems', in: *Supercomputing '91*, IEEE Computer Society Press (Los Alamitos, CA), 578–587.

A. T. Chronopoulos and C. W. Gear (1989), 's-step iterative methods for symmetric linear systems', *J. Comput. Appl. Math.* **25**, 153–168.

A. T. Chronopoulos and S. K. Kim (1990), 's-Step Orthomin and GMRES implemented on parallel computers', Technical Report 90/43R, UMSI, Minneapolis.

E. Chu (1988a), 'Orthogonal decomposition of dense and sparse matrices on multiprocessors', PhD thesis, University of Waterloo.

M. Chu (1988b), 'A note on the homotopy method for linear algebraic eigenvalue problems', *Lin. Alg. Appl.* **105**, 225–236.

M. Chu, T.-Y. Li and T. Sauer (1988), 'Homotopy method for general λ-matrix problems', *SIAM J. Math. Anal. Appl.* **9** (4), 528–536.

L. Csanky (1977), 'Fast parallel matrix inversion algorithms', *SIAM J. Comput.* **5** 618–623.

J.J.M. Cuppen (1981), 'A divide and conquer method for the symmetric tridiagonal eigenproblem', *Numer. Math.* **36**, 177–195.

G. Davis, R. Funderlic and G. Geist (1987), 'A hypercube implementation of the implicit double shift QR algorithm', in: *Hypercube Multiprocessors 1987*, SIAM (Philadelphia), 619–626.

E. F. D'Azevedo and C. H. Romine (1992), 'Reducing communication costs in the conjugate gradient algorithm on distributed memory multiprocessors', Technical Report ORNL/TM-12192, Oak Ridge National Laboratory.

P. P. N. de Groen (1991), 'Base p-cyclic reduction for tridiagonal systems of equations', *Appl. Numer. Math.* **8** 117–126.

E. de Sturler (1991), 'A parallel restructured version of GMRES(m)', Technical Report 91-85, Delft University of Technology, Delft.

A. Deichmoller (1991), 'Über die Berechnung verallgemeinerter singulärer Werte mittles Jacobi-ähnlicher Verfahren', PhD thesis, Fernuniversität—Hagen, Hagen, Germany.

E. Dekel, D. Nassimi and S. Sahni (1981), 'Parallel matrix and graph algorithms', *SIAM J. Comput.* **10** (4), 657–675.

T. Dekker (1971), 'A floating point technique for extending the available precision', *Numer. Math.* **18** 224–242.

J. Demmel (1987), 'Three methods for refining estimates of invariant subspaces', *Computing* **38**, 43–57.

J. Demmel (1992a), 'Specifications for robust parallel prefix operations', Technical report, Thinking Machines Corp., Cambridge, MA.

J. Demmel (1992b), 'Trading off parallelism and numerical stability', Computer Science Division Technical Report UCB//CSD-92-702, University of California, Berkeley, CA.

J. Demmel and N. J. Higham (1992), 'Stability of block algorithms with fast Level 3 BLAS', *ACM Trans. Math. Soft.* **18** (3), 274–291.

J. Demmel and W. Kahan (1990), 'Accurate singular values of bidiagonal matrices', *SIAM J. Sci. Stat. Comput.* **11** (5), 873–912.

J. Demmel and K. Veselić (1992), 'Jacobi's method is more accurate than QR', *SIAM J. Mat. Anal. Appl.* **13** (4), 1204–1246.

S. Doi (1991), 'On parallelism and convergence of incomplete LU factorizations', *Appl. Numer. Math.* **7**, 417–436.

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

J. Dongarra, J. Du Croz, I. Duff and S. Hammarling (1990), 'A set of Level 3 Basic Linear Algebra Subprograms', *ACM Trans. Math. Soft.* **16** (1), 1–17.

J. Dongarra, J. Du Croz, S. Hammarling and Richard J. Hanson (1988), 'An extended set of fortran basic linear algebra subroutines', *ACM Trans. Math. Soft.* **14** (1), 1–17.

J. Dongarra, I. Duff, D. Sorensen and H. van der Vorst (1991), *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM (Philadelphia).

J. Dongarra, G. A. Geist, and C. Romine (1990b), 'Computing the eigenvalues and eigenvectors of a general matrix by reduction to tridiagonal form', Technical Report ORNL/TM-11669, Oak Ridge National Laboratory (to appear in *ACM TOMS*).

J. Dongarra and E. Grosse (1987), 'Distribution of mathematical software via electronic mail', *Commun. ACM* **30**(5), 403–407.

J. Dongarra, S. Hammarling and D. Sorensen (1989), 'Block reduction of matrices to condensed forms for eigenvalue computations', *J. Comput. Appl. Math.* **27**, 215–227 (LAPACK Working Note 2).

J. Dongarra and S. Ostrouchov (1990), 'LAPACK block factorization algorithms on the Intel iPSC/860', Computer Science Dept Technical Report CS-90-115, University of Tennessee, Knoxville, TN (LAPACK Working Note 24).

J. Dongarra and M. Sidani (1991), 'A parallel algorithm for the non-symmetric eigenvalue problem', Computer Science Dept Technical Report CS-91-137, University of Tennessee, Knoxville, TN.

J. Dongarra and D. Sorensen (1987), 'A fully parallel algorithm for the symmetric eigenproblem', *SIAM J. Sci. Stat. Comput.* **8**(2), 139–154.

J. Dongarra and R. van de Geijn (1991), 'Reduction to condensed form for the eigenvalue problem on distributed memory computers', Computer Science Dept Technical Report CS-91-130, University of Tennessee, Knoxville, TN (LAPACK Working Note 30), to appear in *Parallel Comput.*.

J. Dongarra and R. van de Geijn (1991), 'Two dimensional basic linear algebra communication subprograms', Computer Science Dept Technical Report CS-91-138, University of Tennessee, Knoxville, TN (LAPACK Working Note 37).

J. Du Croz, P. J. D. Mayes and G. Radicati di Brozolo (1990), 'Factorizations of band matrices using Level 3 BLAS', Computer Science Dept Technical Report CS-90-109, University of Tennessee, Knoxville, TN (LAPACK Working Note 21).

P. Dubois and G. Rodrigue (1977), 'An analysis of the recursive doubling algorithm', *High Speed Computer and Algorithm Organization*, (D. J. Kuck and A. H. Sameh, eds), Academic Press (New York).

A. Dubrulle (1991), 'The multishift QR algorithm: is it worth the trouble?', Palo Alto Scientific Center Report G320-3558x, IBM Corp., 1530 Page Mill Road, Palo Alto, CA 94304.

I.S. Duff (1986), 'Parallel implementation of multifrontal schemes', *Parallel Comput.* **3**, 193–204.

I.S. Duff, A.M. Erisman and J.K. Reid (1986), *Direct Methods for Sparse Matrices*, Oxford University Press (Oxford, UK).

I.S. Duff and G.A. Meurant (1989), 'The effect of ordering on preconditioned conjugate gradient', *BIT* **29**, 635–657.

P. Eberlein (1962), 'A Jacobi method for the automatic computation of eigenvalues and eigenvectors of an arbitrary matrix', *J. SIAM* **10**, 74–88.

P. Eberlein (1987), 'On the Schur decomposition of a matrix for parallel computation', *IEEE Trans. Comput.* **36**, 167–174.

V. Eijkhout (1992), 'Qualitative properties of the conjugate gradient and Lanczos methods in a matrix framework', Computer Science Dept. Technical Report CS-92-170, University of Tennessee, Knoxville, TN (LAPACK Working Note 51).

S. Eisenstat, M. Heath, C. Henkel and C. Romine (1988), 'Modified cyclic algorithms for solving triangular systems on distributed memory multiprocessors', *SIAM J. Sci. Stat. Comput.* **9**, 589–600.

G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng and M.-Y. Wu (1990), 'Fortran D language specification', Computer Science Department Report CRPC-TR90079, Rice University, Houston, TX.

G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker (1988), *Solving Problems on Concurrent Processors, vol. I*, Prentice Hall (New York).

R. W. Freund, G. H. Golub and N. M. Nachtigal (1992), 'Iterative solution of linear systems', *Acta Numerica* **1**, 57–100.

K. Gallivan, M. Heath, E. Ng, J. Ortega, B. Peyton, R. Plemmons, C. Romine, A. Sameh and R. Voigt (1990), *Parallel Algorithms for Matrix Computations*, SIAM (Philadelphia).

K. Gallivan, W. Jalby, U. Meier and A. Sameh (1988), 'Impact of hierarchical memory systems on linear algebra algorithm design', *Int. J. Supercomput. Appl.* **2**, 12–48.

K. A. Gallivan, M. T. Heath, E. Ng, *et al.* (1990), *Parallel Algorithms for Matrix Computations*, SIAM (Philadelphia).

K. A. Gallivan, R. J. Plemmons and A. H. Sameh (1990), 'Parallel algorithms for dense linear algebra computations', *SIAM Rev.* **32**, 54–135.

B. S. Garbow, J. M. Boyle, J. J. Dongarra and C. B. Moler (1977), Matrix eigensystem routines – EISPACK guide extension, *Lecture Notes in Computer Science, vol. 51*, Springer-Verlag (Berlin).

G. A. Geist (1990), 'Parallel tridiagonalization of a general matrix using distributed memory multiprocessors', in *Proc. Fourth SIAM Conf. on Parallel Processing for Scientific Computing*, SIAM (Philadelphia), 29–35.

G. A. Geist (1991), 'Reduction of a general matrix to tridiagonal form', *SIAM J. Math. Anal. Appl.* **12**(2), 362–373.

G. A. Geist and G. J. Davis (1990), 'Finding eigenvalues and eigenvectors of unsymmetric matrices using a distributed memory multiprocessor', *Parallel Comput.* **13**(2), 199–209.

G. A. Geist, A. Lu and E. Wachspress (1989), 'Stabilized reduction of an arbitrary matrix to tridiagonal form', Technical Report ORNL/TM-11089, Oak Ridge National Laboratory.

M. Gentleman and H. T. Kung (1981), 'Matrix triangularization by systolic arrays', in *Proc. SPIE 298, Real Time Signal Processing*, San Diego, CA, 19–26.

J.A. George (1973), 'Nested dissection of a regular finite element mesh', *SIAM J. Numer. Anal.* **10**, 345–363.

J.A. George, M.T. Heath, J.W.-H. Liu and E. Ng (1986), 'Solution of sparse positive definite systems on a shared memory multiprocessor', *Int. J. Parallel Progr.* **15**, 309–325.

J.A. George, M.T. Heath, J.W.-H. Liu and E. Ng (1988), 'Sparse Cholesky factorization on a local-memory multiprocessor', *SIAM J. Sci. Stat. Comput.* **9**, 327–340.

J.A. George, M.T. Heath, J.W.-H. Liu and E. Ng (1989), 'Solution of sparse positive definite systems on a hypercube', *J. Comput. Appl. Math.* **27**, 129–156.

J.A. George and J.W.-H. Liu (1981), *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall (Englewood Cliffs, NJ).

J.A. George and J.W.-H. Liu (1989), 'The evolution of the minimum degree ordering algorithm', *SIAM Rev.* **31**, 1–19.

J.A. George, J.W.-H. Liu and E. Ng (1989), 'Communication results for parallel sparse Cholesky factorization on a hypercube', *Parallel Comput.* **10**, 287–298.

J. Gilbert and R. Schreiber (1992), 'Highly parallel sparse cholesky factorization', *SIAM J. Sci. Stat. Comput.* **13**, 1151–1172.

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

A. Gottlieb and G. Almasi (1989), *Highly Parallel Computing*, Benjamin Cummings (Redwood City, CA).

M. Gu and S. Eisenstat (1992), 'A stable and efficient algorithm for the rank-1 modification of the symmetric eigenproblem', Computer Science Dept Report YALEU/DCS/RR-916, Yale University.

V. Hari and K. Veselić (1987), 'On Jacobi methods for singular value decompositions', *SIAM J. Sci. Stat. Comput.* **8**, 741–754.

M.T. Heath, E. Ng and B. Peyton (1991), 'Parallel algorithms for sparse linear systems', *SIAM Rev.* **33**, 420–460.

M.T. Heath and C. Romine (1988), 'Parallel solution of triangular systems on distributed memory multiprocessors', *SIAM J. Sci. Stat. Comput.* **9**, 558–588.

D. Heller (1978), 'Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems', *SIAM J. Numer. Anal.* **13**, 484–496.

M. R. Hestenes and E. Stiefel (1954), 'Methods of conjugate gradients for solving linear systems', *J. Res. Natl Bur. Stand.* **49**, 409–436.

High Peformance Fortran (1991), documentation available via anonymous ftp from titan.cs.rice.edu in directory public/HPFF.

N. J. Higham (1990), 'Exploiting fast matrix multiplication within the Level 3 BLAS', *ACM Trans. Math. Soft.* **16**, 352–368.

C. T. Ho (1990), Optimal communication primitives and graph embeddings on hypercubes, PhD thesis, Yale University.

C. T. Ho, S. L. Johnsson and A. Edelman (1991), 'Matrix multiplication on hypercubes using full bandwidth and constant storage', in *The Sixth Distributed Memory Computing Conf. Proc.*, IEEE Computer Society Press (New York), 447–451.

X. Hong and H. T. Kung (1981), 'I/O complexity: the red blue pebble game', in *Proc. 13th Symposium on the Theory of Computing*, ACM (New York), 326–334.

J. Howland (1983), 'The sign matrix and the separation of matrix eigenvalues', *Lin. Alg. Appl.* **49**, 221–232.

I. Ipsen and E. Jessup (1990), 'Solving the symmetric tridiagonal eigenvalue problem on the hypercube', *SIAM J. Sci. Stat. Comput.* **11**(2), 203–230.

B.M. Irons (1970), 'A frontal solution program for finite element analysis', *Int. J. Numer. Math. Engrg* **2**, 5–32.

J.A.G. Jess and H.G.M. Kees (1982), 'A data structure for parallel L/U decomposition', *IEEE Trans. Comput.* **C-31**, 231–239.

E. Jessup (1991), 'A case against a divide and conquer approach to the nonsymmetric eigenproblem', Technical Report ORNL/TM-11903, Oak Ridge National Laboratory.

E. Jessup and I. Ipsen (1992), 'Improving the accuracy of inverse iteration', *SIAM J. Sci. Stat. Comput.* **13**(2), 550–572.

E. Jessup and D. Sorensen (1989), 'A divide and conquer algorithm for computing the singular value decomposition of a matrix', in *Proc. Third SIAM Conf. on Parallel Processing for Scientific Computing* SIAM (Philadelphia), 61–66. SIAM.

S. L. Johnsson (1987), 'Communication efficient basic linear algebra computations on hypercube architecture', *J. Parallel Distr. Comput.* **4**, 133–172.

S. L. Johnsson (1990), private communication.

S. L. Johnsson and C. T. Ho (1989), 'Matrix multiplication on Boolean cubes using generic communication primitives', in *Parallel Processing and Medium Scale Multiprocessors*, (A. Wouk, ed), SIAM (Philadelphia), 108–156.

W. Kahan (1968), 'Accurate eigenvalues of a symmetric tridiagonal matrix', Computer Science Dept Technical Report CS41, Stanford University, Stanford, CA, July 1966 (revised June 1968).

R. Karp and V. Ramachandran (1990), 'Parallel algorithms for shared memory machines', in *Handbook of Theoretical Computer Science, vol. A: Algorithms and Complexity*, (J. van Leeuwen, ed), Elsevier and MIT Press (New York), 869–941.

C. Kenney and A. Laub (1991), 'Rational iteration methods for the matrix sign function', *SIAM J. Math. Anal. Appl.* **21**, 487–494.

A. S. Krishnakumar and Morf (1986), 'Eigenvalues of a symmetric tridiagonal matrix: a divide and conquer approach', *Numer. Math.* **48**, 349–368.

D. Kuck and A. Sameh (1977), 'A parallel QR algorithm for symmetric tridiagonal matrices', *IEEE Trans. Comput.* **C-26**(2).

H. T. Kung (1974), 'New algorithms and lower bounds for the parallel evaluation of certain rational expressions', Technical report, Carnegie Mellon University.

J. J. Lambiotte and R. G. Voigt (1974), 'The solution of tridiagonal linear systems on the CDC-STAR-100 computer', Technical report, ICASE-NASA Langley Research Center, Hampton, VA.

B. Lang (1992), 'Reducing symmetric band matrices to tridiagonal form – a comparison of a new parallel algorithm with two serial algorithms on the iPSC/860', Institut für angewandte mathematik report, Universität Karlsruhe.

C. Lawson, R. Hanson, D. Kincaid and F. Krogh (1979), 'Basic linear algebra subprograms for fortran usage', *ACM Trans. Math. Soft.* **5**, 308–323.

S. Lederman, A. Tsao and T. Turnbull (1992), 'A parallelizable eigensolver for real diagonalizable matrices with real eigenvalues', Report TR-01-042, Supercomputing Research Center, Bowie, MD.

J.G. Lewis, B.W. Peyton and A. Pothen (1989), 'A fast algorithm for reordering sparse matrices for parallel factorization', *SIAM J. Sci. Stat. Comput.* **10**, 1156–1173.

G. Li and T. Coleman (1988), 'A parallel triangular solver on a distributed memory multiprocessor', *SIAM J. Sci. Stat. Comput.* **9**, 485–502.

R. Li (1992), 'Solving the secular equation stably', UC Berkeley Math. Dept Report, in preparation.

T.-Y. Li and Z. Zeng (1992), 'Homotopy-determinant algorithm for solving nonsymmetric eigenvalue problems', *Math. Comput.* to appear.

T.-Y. Li, Z. Zeng and L. Cong (1992), 'Solving eigenvalue problems of non-symmetric matrices with real homotopies', *SIAM J. Numer. Anal.* **29**(1), 229–248.

T.-Y. Li, H. Zhang and X.-H. Sun (1991), 'Parallel homotopy algorithm for symmetric tridiagonal eigenvalue problem', *SIAM J. Sci. Stat. Comput.* **12**(3), 469–487.

C-C. Lin and E. Zmijewski (1991), 'A parallel algorithm for computing the eigenvalues of an unsymmetric matrix on an SIMD mesh of processors', Department of Computer Science TRCS 91-15, University of California, Santa Barbara, CA.

J.W.-H. Liu (1986), 'Computational models and task scheduling for parallel sparse Cholesky factorization', *Parallel Comput.* **3**, 327–342.

J.W.-H. Liu (1989), 'Reordering sparse matrices for parallel elimination', *Parallel Comput.* **11**, 73–91.

J.W.-H. Liu (1990), 'The role of elimination trees in sparse factorization', *SIAM J. Matrix Anal. Appl.* **11**, 134–172.

S.-S. Lo, B. Phillipe and A. Sameh (1987), 'A multiprocessor algorithm for the symmetric eigenproblem', *SIAM J. Sci. Stat. Comput.* **8**(2), 155–165.

R. Lucas, W. Blank and J. Tieman (1987), 'A parallel solution method for large sparse systems of equations', *IEEE Trans. Comput. Aided Des.* **CAD-6**, 981–991.

F. Luk and H. Park (1989), 'On parallel Jacobi orderings', *SIAM J. Sci. Stat. Comput.* **10**(1), 18–26.

S. C. Ma, M. Patrick and D. Szyld (1989), 'A parallel, hybrid algorithm for the generalized eigenproblem', in *Parallel Processing for Scientific Computing*, (Garry Rodrigue, ed), SIAM (Philadelphia), ch 16, 82–86.

N. K. Madsen, G. H. Rodrigue and J. I. Karush (1976), 'Matrix multiplication by diagonals on a vector/parallel processor', *Inform. Process. Lett.* **5**, 41–45.

A. N. Malyshev (1991), 'Parallel aspects of some spectral problems in linear algebra', Dept of Numerical Mathematics Report NM-R9113, Centre for Mathematics and Computer Science, Amsterdam.

V. Mehrmann (1991), 'Divide and conquer methods for block tridiagonal systems', Technical Report Bericht Nr. 68, Inst. fuer Geometrie und Prakt. Math., Aachen.

U. Meier (1985), 'A parallel partition method for solving banded systems of linear equations', *Parallel Comput.* **2**, 33–43.

J. A. Meijerink and H. A. van der Vorst (1977), 'An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix', *Math. Comput.* **31**, 148–162.

J. A. Meijerink and H. A. van der Vorst (1981), 'Guidelines for the usage of incomplete decompositions in solving sets of linear equations as they occur in practical problems', *J. Comput. Phys.* **44**, 134–155.

G. Meurant (1984a), 'The block preconditioned conjugate gradient method on vector computers', *BIT* **24**, 623–633.

G. Meurant (1984b), 'Numerical experiments for the preconditioned conjugate gradient method on the CRAY X-MP/2', Technical Report LBL-18023, University of California, Berkeley, CA.

P. H. Michielse and H. A. van der Vorst (1988), 'Data transport in Wang's partition method', *Parallel Comput.* **7**, 87–95.

M. Mu and J.R. Rice (1992), 'A grid based subtree-subcube assignment strategy for solving partial differential equations on hypercubes', *SIAM J. Sci. Stat. Comput.* **13**, 826–839.

J.M. Ortega (1988), *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Press (New York and London).

M.H.C. Paardekooper (1989), 'A quadratically convergent parallel Jacobi process for diagonally dominant matrices with distinct eigenvalues', *J. Comput. Appl. Math.* **27**, 3–16.

C. C. Paige (1976), 'Error analysis of the Lanczos algorithm for tridiagonalizing a symmetric matrix', *J. Inst. Math. Appl.* **18**, 341–349.

P. Pandey, C. Kenney and A. Laub (1990), 'A parallel algorithm for the matrix sign function', *Int. J. High Speed Comput.* **2**(2), 181–191.

B. N. Parlett (1980), *The Symmetric Eigenvalue Problem*, Prentice-Hall (Englewood Cliffs, NJ).

B. Parlett (1992), private communication.

B. Parlett (1992b), 'Reduction to tridiagonal form and minimal realizations', *SIAM J. Math. Anal. Appl.* **13**(2), 567–593.

B. Parlett and V. Fernando (1992), 'Accurate singular values and differential QD algorithms', Math. Department PAM-554, University of California, Berkeley, CA.

B. N. Parlett and J. K. Reid (1981), 'Tracking the progress of the Lanczos algorithm for large symmetric eigenproblems', *IMA J. Numer. Anal.* **1**, 135–155.

S. G. Petiton (1992), 'Parallel subspace method for non-Hermitian eigenproblems on the Connection Machine (CM2)', *Appl. Numer. Math.* **10**.

C. Pommerell (1992), Solution of large unsymmetric systems of linear equations, PhD thesis, Swiss Federal Institute of Technology, Zürich.

A. Pothen, S. Jha and U. Vemulapati (1987), 'Orthogonal factorization on a distributed memory multiprocessor', in *Hypercube Multiprocessors 1987*, Knoxville, TN, SIAM (Philadelphia), 587–598.

D. Priest (1991), 'Algorithms for arbitrary precision floating point arithmetic', in *Proc. 10th Symp. Computer Arithmetic, Grenoble, France*, (P. Kornerup and D. Matula, eds), IEEE Computer Society Press (New York), 132–145.

G. Radicati di Brozolo and Y. Robert (1989), 'Parallel conjugate gradient-like algorithms for solving sparse nonsymmetric linear systems on a vector multiprocessor', *Parallel Comput.* **11**, 223–239.

Y. Robert (1990), *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm*, Wiley (New York).

J. Roberts (1980), 'Linear model reduction and solution of the algebraic Riccati equation', *Int. J. Control* **32**, 677–687.

C. Romine and J. Ortega (1988), 'Parallel solution of triangular systems of equations', *Parallel Comput.* **6**, 109–114.

E. Rothberg and A. Gupta (1989), 'Fast sparse matrix factorization on modern workstations', Technical Report STAN-CS-89-1286, Stanford University, Stanford, California.

Y. Saad (1980), 'Variations on Arnoldi's method for computing eigenelements of large unsymmetric matrices', *Lin. Alg. Appl.* **34**, 269–295.

Y. Saad (1985a), 'Partial eigensolutions of large nonsymmetric matrices', Technical Report, Dept. of Comp. Science, New Haven, CN.

Y. Saad (1985b), 'Practical use of polynomial preconditionings for the conjugate gradient method', *SIAM J. Sci. Stat. Comput.* **6**, 865–881.

Y. Saad (1988), 'Krylov subspace methods on supercomputers', Technical report, RIACS, Moffett Field, CA.

Y. Saad and M. H. Schultz (1985), 'Conjugate Gradient-like algorithms for solving nonsymmetric linear systems', *Math. Comput.* **44**, 417–424.

A. Sameh (1971), '1On Jacobi and Jacobi-like algorithms for a parallel computer', *Math. Comput.* **25**, 579–590.

A. Sameh (1985), 'On some parallel algorithms on a ring of processors', *Comput. Phys. Commun.* **37**, 159–166.

A. Sameh and R. Brent (1977), 'Solving triangular systems on a parallel computer', *SIAM J. Numer. Anal.* **14**, 1101–1113.

J. J. F. M. Schlichting and H. A. van der Vorst (1987), 'Solving bidiagonal systems of linear equations on the CDC CYBER 205', Technical Report NM-R8725, CWI, Amsterdam, the Netherlands.

J. J. F. M. Schlichting and H. A. van der Vorst (1989), 'Solving 3D block bidiagonal linear systems on vector computers', *J. Comput. Appl. Math.* **27**, 323–330.

R. Schreiber and C. Van Loan (1989), 'A storage efficient WY representation for products of Householder transformations', *SIAM J. Sci. Stat. Comput.* **10**, 53–57.

M. K. Seager (1986), 'Parallelizing conjugate gradient for the CRAY X-MP', *Parallel Comput.* **3**, 35–47.

G. Shroff (1991), 'A parallel algorithm for the eigenvalues and eigenvectors of a general complex matrix', *Numer. Math.* **58**, 779–805.

G. Shroff and R. Schreiber (1989), 'On the convergence of the cyclic Jacobi method for parallel block orderings', *SIAM J. Matt. Anal. Appl.* **10**, 326–346.

H. Simon (1989), 'Bisection is not optimal on vector processors', *SIAM J. Sci. Stat. Comput.* **10**(1), 205–209.

I. Slapničar (1992), Accurate symmetric eigenreduction by a Jacobi method, PhD thesis, Fernuniversität – Hagen, Hagen, Germany.

B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema and C. B. Moler (1976), *Matrix Eigensystem Routines – EISPACK Guide: Springer Lecture Notes in Computer Science 6*, Springer-Verlag (Berlin).

D. Sorensen (1992), 'Implicit application of polynomial filters in a k-step Arnoldi method', *SIAM J. Math. Anal. Appl.* **13**(1), 357–385.

D. Sorensen and P. Tang (1991), 'On the orthogonality of eigenvectors computed by divide-and-conquer techniques', *SIAM J. Numer. Anal.* **28**(6), 1752–1775.

G. W. Stewart (1985), 'A Jacobi-like algorithm for computing the Schur decomposition of a non-Hermitian matrix', *SIAM J. Sci. Stat. Comput.* **6**, 853–864.

G. W. Stewart (1987), 'A parallel implementation of the QR algorithm', *Parallel Comput.* **5**, 187–196.

E. Stickel (1991), 'Separating eigenvalues using the matrix sign function', *Lin. Alg. Appl.* **148**, 75–88.

H. S. Stone (1973), 'An efficient parallel algorithm for the solution of a tridiagonal linear system of equations', *J. Assoc. Comput. Mach.* **20**, 27–38.

P. Swarztrauber (1992), 'A parallel algorithm for computing the eigenvalues of a symmetric tridiagonal matrix', *Math. Comput.* to appear.

Thinking Machines Corporation (1987), *Connection Machine Model CM-2 Technical Summary*, IBM.

R. van de Geijn (1987), Implementing the QR algorithm on an array of processors, PhD thesis, University of Maryland, College Park, Computer Science Department Report TR-1897.

R. van de Geijn and D. Hudson (1989), 'Efficient parallel implementation of the nonsymmetric QR algorithm', in *Hypercube Concurrent Computers and Applications*, (J. Gustafson, ed), ACM (New York).

E. Van de Velde (1992), *Introduction to Concurrent Scientific Computing*, Caltech (Pasadena).

H. A. van der Vorst (1982), 'A vectorizable variant of some ICCG methods', *SIAM J. Sci. Stat. Comput.* **3**, 86–92.

H. A. van der Vorst (1986), 'The performance of Fortran implementations for preconditioned conjugate gradients on vector computers', *Parallel Comput.* **3**, 49–58.

H. A. van der Vorst (1987a), 'Analysis of a parallel solution method for tridiagonal linear systems', *Parallel Comput.* **5**, 303–311.

H. A. van der Vorst (1987b), 'Large tridiagonal and block tridiagonal linear systems on vector and parallel computers', *Parallel Comput.* **5**, 45–54.

H. A. van der Vorst (1989a), 'High performance preconditioning', *SIAM J. Sci. Statist. Comput.* **10**, 1174–1185.

H. A. van der Vorst (1989b), 'ICCG and related methods for 3D problems on vector computers', *Comput. Phys. Commun.* **53**, 223–235.

H. A. van der Vorst (1989c), 'Practical aspects of parallel scientific computing', *Future Gener. Comput. Sys.* **4**, 285–291.

H. A. van der Vorst and K. Dekker (1989), 'Vectorization of linear recurrence relations', *SIAM J. Sci. Stat. Comput.* **10**, 27–35.

H. A. van der Vorst and C. Vuik (1991), 'GMRESR: A family of nested GMRES methods', Technical Report 91-80, Delft University of Technology, Faculty of Technical Mathematics.

K. Veselić (1979), 'A quadratically convergent Jacobi-like method for real matrices with complex conjugate eigenvalues', *Numer. Math.* **33**, 425–435.

H. F. Walker (1988), 'Implementation of the GMRES method using Householder transformations', *SIAM J. Sci. Stat. Comput.* **9**, 152–163.

H. H. Wang (1981), 'A parallel method for tridiagonal equations', *ACM Trans. Math. Soft.* **7**, 170–183.

D. Watkins (1992), 'Shifting strategies for the parallel QR algorithm', Dept of Pure and Applied Mathematics report, Washington State Univ., Pullman, WA.

D. Watkins and L. Elsner (1991), 'Convergence of algorithms of decomposition type for the eigenvalue problem', *Lin. Alg. Appl.* **143**, 19–47.

C.-P. Wen and K. Yelick (1992), 'A survey of message passing systems', Computer science division, University of California, Berkeley, CA.

J. H. Wilkinson (1965), *The Algebraic Eigenvalue Problem*, Oxford University Press (Oxford).

Z. Zeng (1991), Homotopy-determinant algorithm for solving matrix eigenvalue problems and its parallelizations, PhD thesis, Michigan State University.

H. Zima and B. Chapman (1991), *Supercompilers for Parallel and Vectors Computers*, ACM (New York).

E. Zmijewski (1989), 'Limiting communication in parallel sparse Cholesky factorization', Technical Report TRCS89-18, Department of Computer Science, University of California, Santa Barbara, CA.