



MARTIN MARIETTA ENERGY SYSTEMS LIBRARIES



3 4456 0334510 0

ORNL/TM-11786

OAK RIDGE  
NATIONAL  
LABORATORY

MARTIN MARIETTA

# Modeling Speedup in Parallel Sparse Matrix Factorization

L. S. Ostrouchov  
M. T. Heath  
C. H. Romine

OAK RIDGE NATIONAL LABORATORY  
CENTRAL RESEARCH LIBRARY  
CIRCULATION SECTION  
4000 PAVEN 173  
**LIBRARY LOAN COPY**  
DO NOT TRANSFER TO ANOTHER PERSON  
If you wish someone else to see this  
report, send in name with report and  
the library will arrange a loan.

MANAGED BY  
MARTIN MARIETTA ENERGY SYSTEMS, INC.  
FOR THE UNITED STATES  
DEPARTMENT OF ENERGY

This report has been reproduced exactly from the best available copy.

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831, prices available from (615) 576-8401, FTIS 626-8401.

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161.

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ORNL/TM-11786

Engineering Physics and Mathematics Division

Mathematical Sciences Section

**MODELING SPEEDUP IN PARALLEL  
SPARSE MATRIX FACTORIZATION**

L.S. Ostrouchov †  
M.T. Heath †  
C.H. Romine †

† Computer Science Department  
University of Tennessee  
Knoxville, TN 37996-1301

‡ Mathematical Sciences Section  
Oak Ridge National Laboratory  
Oak Ridge, TN 37831-8083

Prepared for  
Office of Energy Research  
KC 07 01 01 0

Date Published: December, 1990

Prepared by the  
Oak Ridge National Laboratory  
Oak Ridge, Tennessee 37831  
managed by  
Martin Marietta Energy Systems, Inc.  
for the  
U.S. DEPARTMENT OF ENERGY  
under Contract No. DE-AC05-84OR21400



3 4456 0334510 0



## Contents

1	INTRODUCTION . . . . .	1
2	SPARSE CHOLESKY FACTORIZATION . . . . .	4
2.1	Column Task Graph . . . . .	5
2.2	An Example . . . . .	7
3	DETERMINATION OF MAXIMUM SPEEDUP . . . . .	10
3.1	Strategy 1 . . . . .	11
3.2	Strategy 2 . . . . .	12
3.3	Strategy 3 . . . . .	13
3.4	Strategy 4 . . . . .	14
3.5	Summary of Results for Example Problem . . . . .	16
4	COMPARISON WITH OBSERVED SPEEDUPS . . . . .	17
5	CONCLUSIONS . . . . .	21
6	References . . . . .	22



## **Acknowledgements**

We wish to thank Al Geist, Esmond Ng, and Barry Peyton for many helpful discussions and their contributions to this research.



# MODELING SPEEDUP IN PARALLEL SPARSE MATRIX FACTORIZATION

L.S. Ostrouchov  
M.T. Heath  
C.H. Romine

## Abstract

This paper is an attempt to explain the observed performance of sparse matrix factorization algorithms on parallel computers. In particular, we examine whether the disappointing performance of these algorithms is due to insufficient parallelism in the problem or to the architectural characteristics of existing parallel computers. Through a series of theoretical models of increasing realism, we first determine upper and lower bounds on the speedup that can be expected in practice for this problem, and end with a parameterized model that is capable of reproducing the full range of behavior within these bounds, including the speedups actually observed in practice. This model suggests that the current limits on speedup in sparse factorization are due to poor communication performance of the present generation of parallel computer architectures rather than to a lack of parallelism in the problem.



## 1. INTRODUCTION

In this paper we attempt to gain, through a detailed study of a particular problem, a better understanding of the factors affecting the performance of parallel architectures. The problem we have chosen is Cholesky factorization of symmetric positive definite sparse matrices. This factorization is the most computationally intensive step in solving many large linear systems that arise in all areas of science and engineering, including the analysis of structures and networks, and thus has received a substantial amount of attention from developers of parallel algorithms (see [19] for a survey). However, sparse Cholesky factorization has often shown disappointing performance results on parallel computers. An additional motivation for selecting this problem is that we can make use of the theoretical concepts and techniques, based largely on graph theory, that have been developed for analyzing sparse elimination algorithms. Parallel sparse Cholesky factorization is sufficiently complex to be typical of scientific computations in general, as well as being interesting in its own right, yet is amenable to theoretical analysis.

The potential performance of a parallel computer in solving a given computational problem depends on the nature of the problem being solved, the parallel algorithm employed, and the architectural details of the particular parallel computer. For realistic problems of interest, these factors interact in an extremely complex manner that is difficult to analyze in detail. Two relatively simple and commonly used measures of effectiveness are *speedup* and *efficiency*. The speedup resulting from the use of  $p$  processors is defined by the ratio of execution times  $S_p = T_1/T_p$ , where the subscript indicates the number of processors used, and the best sequential algorithm is used for the single-processor case. The efficiency in using  $p$  processors is defined by the ratio  $E_p = S_p/p$ , which can be interpreted as the average utilization of the  $p$  processors. Attaining perfect efficiency would require that  $S_p = p$ , so that  $E_p = 1$ . Such ideal performance is not generally achievable in practice, however, due to communication costs, synchronization overhead, load imbalances, contention for resources, and various other inhibiting factors. In attempting to understand and improve performance, it is important to identify those limitations that are due to the architecture (which we will refer to as *hardware* limits) and those that are due to the algorithm and/or the problem (which we will refer to as *software* limits).

Perhaps the simplest and best known model of parallel computer performance is due to Amdahl [1,2]. In this model, the hardware is characterized by a single parameter, namely the number of processors  $p$ , and the software is also characterized by a single parameter, the fraction  $f$  of the computation that is inherently serial. "Amdahl's Law" then gives the following upper bound on speedup for a given problem of given size:

$$S_p \leq 1/(f + (1 - f)/p).$$

Although Amdahl's analysis can yield useful insights, it suffers from a number of shortcomings: (1) the serial fraction  $f$  is difficult to determine *a priori*; indeed, in practice it is often inferred *a posteriori* from observed performance rather than used to predict performance, (2) there is an implicit assumption that at any given time all work is either completely serial or completely ( $p$ -fold) parallel, which is seldom true, and (3) the serial fraction  $f$  is not a very robust measure in that it tends to be a function not

only of the algorithm used, but also the problem size and the number of processors used. Thus, for example, a constant value for  $f$  usually entails a fixed problem size, which does not reflect the way that supercomputers tend to be used in practice (see [18,28] for a discussion of this last point).

A performance model with greater flexibility and predictive power, yet still appealing in its simplicity, has been given by Eager, Zahorjan, and Lazowska [8]. Their approach is based on the observation that speedup and efficiency, rather than rising or falling together, often show an inverse relationship. Thus, for a fixed size problem, execution time may be reduced by using more processors, but the consequent improvement in speedup is usually accompanied by a decline in the average utilization of the processors (i.e., efficiency). Intuitively speaking, as the number of processors grows it becomes increasingly difficult to keep them all busy doing useful work; the number of “wasted” machine cycles increases more rapidly than the execution time decreases, so the overall efficiency declines. The analysis in [8] of this tradeoff between speedup and efficiency uses the concepts of hardware bound, software bound, and average parallelism. The *hardware bound* on speedup is simply the number of processors used (i.e., the speedup cannot exceed the number of processors). The *software bound* is based on representing the parallel computation by a directed acyclic graph (DAG), whose nodes correspond to (serial) computational subtasks, and whose arcs reflect precedence constraints between subtasks. The length of a path in this graph is defined to be the sum of the computations at the nodes along the path. The software bound on speedup is then given by the ratio of the total amount of computation to the length of a longest serial path in the subtask graph (i.e., regardless of how many processors are used, execution time must still be at least as long as a longest serial path in the subtask graph). Finally, *average parallelism* is defined in four ways, which are then shown to be equivalent [8]:

1. the average number of processors that are busy during the execution of a program, given an unlimited number of processors;
2. the speedup, given an unlimited number of processors;
3. the ratio of the total amount of computation to the length of a longest path in the subtask graph; and
4. the intersection point of the hardware bound and the software bound on speedup.

The hardware and software bounds and the actual speedup for a simple example of a software system (taken from [8]) are shown in Figure 1. Based on the second definition above, we will use in this paper the more self-explanatory term *maximum speedup* rather than average parallelism. Our principal tool in determining this quantity, however, will be the third definition, using the subtask graph of the parallel computation.

In exploring the limits of parallelism in Cholesky factorization, we will naturally focus on parallel architectures having a relatively large number of processors. In the current state of technology, shared-memory architectures tend to be limited to a relatively small number of processors, typically up to about thirty. To go beyond this level,

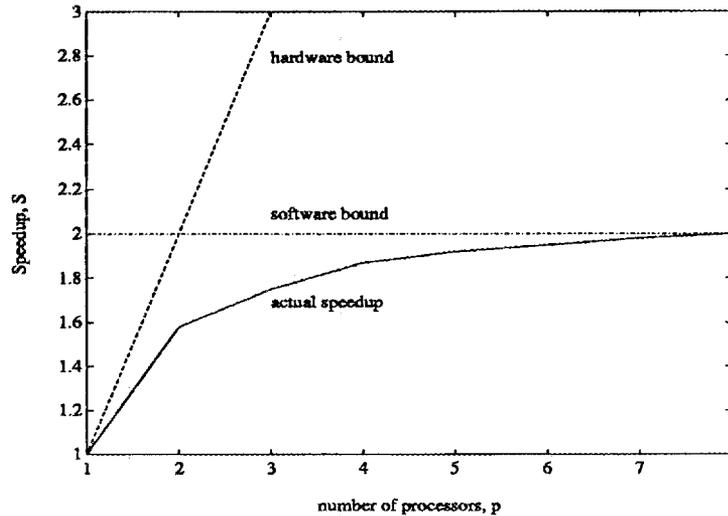


Figure 1: Hardware and software bounds on actual speedup.

to say a hundred or a thousand processors, the most popular and widely available solution at present is distributed-memory architectures, typified by hypercubes. This is the class of parallel architectures on which we will focus in selecting appropriate parallel algorithms and in conducting numerical experiments. Still higher levels of parallelism, with many thousands of processors, currently requires the use of very fine-grained algorithms and SIMD architectures, typified by the Connection Machine, and is beyond the scope of the present investigation.

Parallel algorithms for sparse Cholesky factorization on hypercubes and other distributed-memory, message-passing architectures have been the object of considerable research. Although the performance of these algorithms has shown steady improvement, it is still fair to say that they have yet to achieve satisfactory performance levels in practice, with efficiencies seldom exceeding 50%. One of our main objectives in this study is to gain a better understanding of this disappointing performance: is it due to an inherent lack of parallelism in the problem, or is it due to the architectural characteristics of the current generation of distributed-memory parallel computers? We will attempt to answer this question by developing theoretical models of the sparse factorization process that will enable us to estimate the maximum speedup achievable for a given problem. Further, we will compare these theoretical estimates with results obtained on a real machine.

The remainder of the paper is organized as follows. Section 2 contains a detailed discussion of parallel sparse Cholesky factorization, an explanation of the subtask graph we use to model this application, and a small example problem illustrating these concepts. Section 3 contains a discussion of the calculation of maximum speedup for our application, including three strategies that ignore communication costs and one strategy that incorporates communication costs. Section 3 concludes with a comparison of the four strategies on a small example problem. Section 4 compares our theoretical

results with actual speedups observed on an Intel iPSC/2 hypercube multicomputer for a series of finite-difference grid problems. Finally, section 5 contains our conclusions.

## 2. SPARSE CHOLESKY FACTORIZATION

Consider an  $n \times n$  symmetric positive definite matrix  $A$ . Its Cholesky factor is a lower triangular matrix  $L$  such that  $A = LL^T$ . The computational significance of the Cholesky factor is that a system of linear equations  $Ax = b$  can be solved by successive forward and back substitutions in the triangular systems  $Ly = b$  and  $L^T x = y$ . If  $A$  is a sparse matrix, meaning that most of its entries are zero, then during the course of the factorization process some entries that are initially zero in the lower triangle of  $A$  may become nonzero entries in  $L$ . These entries of  $L$  are known as *fill* or *fill-in*. Usually, however, many zero entries in the lower triangle of  $A$  remain zero in  $L$ . For efficient use of computer memory and processing time, it is desirable to keep the amount of fill small, and to store and operate on only the nonzero entries of  $A$  and  $L$ .

A given linear system yields the same solution regardless of the particular order in which the equations and unknowns are numbered. This freedom in choosing the ordering can be exploited to enhance the preservation of sparsity in the Cholesky factorization process. Let  $P$  be any permutation matrix. Since  $PAP^T$  is also a symmetric positive definite matrix,  $P$  can often be chosen so that the Cholesky factor  $\bar{L}$  of  $PAP^T$  has less fill than  $L$ . The permuted system is equally useful for solving the original linear system, with the triangular solution steps simply becoming  $\bar{L}y = Pb$  and  $\bar{L}^T z = y$ , and finally  $x = P^T z$ . Unfortunately, finding a permutation  $P$  that minimizes fill is a very difficult combinatorial problem (an NP-complete problem) [29]. Thus, a great deal of research effort has been devoted to developing good heuristics for limiting fill in sparse Cholesky factorization, including the nested dissection algorithm [10,14] and the minimum degree algorithm [17,22]. The choice of ordering also has a substantial effect on the potential parallelism with which the Cholesky factorization can be computed, as we will discuss in section 2.1.

Graph theory provides a number of extremely helpful tools in modeling the structural aspects of sparse elimination algorithms. The *graph* of an  $n \times n$  symmetric matrix  $A$ , denoted by  $G(A)$ , is an undirected graph having  $n$  vertices (or nodes), with an edge between two vertices  $i$  and  $j$  if the corresponding entry  $a_{ij}$  is nonzero in the matrix. The *filled graph* of  $A$ , denoted by  $F(A)$ , is the graph of  $A$  with all fill edges added: there is an edge between two vertices  $i$  and  $j$  of  $F(A)$ , with  $i > j$ , if  $l_{ij} \neq 0$  in the Cholesky factor matrix  $L$  (equivalently,  $F(A)$  is the graph of  $L + L^T$ ). Finally, the *elimination tree* associated with the Cholesky factor  $L$  of  $A$ , denoted by  $T(A)$ , is a graph having  $n$  vertices, with an edge between two vertices  $i$  and  $j$ , for  $i > j$ , if  $i = \text{parent}(j)$ , where  $\text{parent}(j)$  is the row index of the first off-diagonal nonzero, if any, in column  $j$  of  $L$ . Throughout this paper, we will assume that the matrix  $A$  is irreducible, so that column  $n$  is the only column having no off-diagonal nonzero, and hence  $T(A)$  is indeed a tree with node  $n$  as its unique root. See [24] for a survey of the role of elimination trees in sparse factorization. For a much more detailed general discussion of sparse Cholesky factorization, we refer the reader to [16].

## 2.1. Column Task Graph

In exploiting parallelism to solve any problem, the computational work must be broken into a number of subtasks that can be assigned to separate processors. The most appropriate number and size of these tasks depend on the target parallel architecture and the extent of the parallelism at various levels in the problem. The term often used to denote the size of computational tasks in a parallel implementation is *granularity*. In sparse factorization, as in most problems, a number of levels of computational granularity can potentially be exploited. Liu [23] characterizes three models of parallel Cholesky factorization that exhibit fine, medium, and large granularity, respectively:

1. *fine-grain parallelism*, in which each subtask is a single multiply-add pair,
2. *medium-grain parallelism*, in which each task is an operation on an entire column. Examples of such operations include adding a scalar multiple of one column to another or multiplying a column by a scalar. These operations correspond, respectively, to *saxpy* and *sscal* from the BLAS (Basic Linear Algebra Subroutines) [21].
3. *large-grain parallelism*, in which each task is the complete computation of a column of the Cholesky factor or perhaps an entire set of columns in a subtree of the elimination tree.

A fine-grained model for studying the parallel solution of linear systems was introduced by Wing and Huang [27]. It associates each task with a single multiplicative operation in the factorization. A precedence relation is maintained in the following way. If one task computes a value needed by another task, then the first task must precede the second one. The directed edges of the task graph follow this precedence relation. The resulting task graph is a directed acyclic graph (DAG). The number of nodes in the graph is equal to the number of multiplicative operations required to perform the Cholesky factorization. For large problems, this fine-grained model is appropriate only if several thousands of processors are available.

Jess and Kees [20] introduced a model in which the structure of the task graph is essentially that of the elimination tree defined above. Thus, the nodes of the task graph are simply the columns of the Cholesky factor and the precedence relation defining the directed edges is given by the parent relation defined above. This large-grained model is most appropriate when only a relatively small number of processors is available.

For parallel dense Cholesky factorization, a medium-grained task model was introduced in [11] and then extended to the sparse case in [12] and [13] and many subsequent papers (see [19] for a survey). The scheduling of the medium-grained tasks for parallel sparse Cholesky factorization is studied in detail by Liu in [23]. Each computational subtask in this model is a column-oriented operation of one of the following two types:

1. *cdiv(j)*: division of column  $j$  by a scalar;
2. *cmod(j, k)*: modification of column  $j$  by column  $k$ ,  $j > k$ .

Specifically,  $cdiv(j)$  divides the nonzero entries in column  $j$  by the square root of its diagonal element, and  $cmod(j, k)$  subtracts a scalar multiple of column  $k$  from column  $j$ . The precedence relation among these column-oriented tasks is as follows:

$$cmod(j, k) \text{ with } k < j \rightarrow cdiv(j) \rightarrow cmod(i, j) \text{ with } j < i.$$

Thus,  $cdiv(j)$  cannot begin until  $cmod(j, k)$  has been completed, and  $cdiv(j)$  must finish before  $cmod(i, j)$  can begin. In terms of operations on individual matrix elements, some of the operations in  $cdiv(j)$  could in principle be executed without requiring that all operations in  $cmod(j, k)$  first be completed, and similarly for the relationship between  $cdiv(j)$  and  $cmod(i, j)$ . However, in the medium-grained model this potential fine-grained parallelism is not exploited, the rationale being that the communication and other overhead costs of exploiting parallelism at the level of individual floating point operations would be greater than the potential gain in execution time for the target architecture.

There is a one-to-one correspondence between the off-diagonal nonzero entries  $\ell_{jk}$  in the Cholesky factor matrix  $L$  and the  $cmod(j, k)$  operations. Thus, as observed by Liu [23], the medium-grained model based on column-oriented tasks, which he calls the *column task graph*, is structurally equivalent to the filled graph  $F(A)$  of the matrix  $A$ . Since each column division operation  $cdiv$  corresponds to a diagonal element of  $L$ , and each column update operation  $cmod$  corresponds to a nonzero off-diagonal element of  $L$ , the column task graph, which we will denote by  $C(A)$ , is simply the elimination tree  $T(A)$  with edges added to incorporate the additional nonzeros in the factor matrix  $L$ . The nodes of the graph  $C(A)$  correspond to the  $cdiv$  operations and the edges correspond to the  $cmod$  operations. To derive a true task graph, in which all tasks are represented by nodes and all edges represent precedence relations, we could merely insert a node representing each  $cmod$  operation within each “edge” in the above sense, but we will not make such a distinction, since no confusion should arise.

There is an intimate structural interplay between the elimination tree and the column task graph. The two graphs have the same node set, and the elimination tree is a spanning tree for the column task graph. Thus, the elimination tree serves as a convenient mechanism for traversing the column task graph, as required by some algorithms. Unfortunately, there is great potential for confusion in the terminology for referring to the relationships among nodes in the two graphs. In the standard terminology for a DAG representing a task graph, naturally enough, the ancestor tasks precede their descendants in time. In the standard terminology for trees, however, the parent/child relationship among immediate neighbors, or more generally the ancestor/descendant relationship among more distant nodes, places a parent or ancestor node between its child or descendant node and the root node. Thus, in the case of elimination trees, the leaf nodes are descendants of the other nodes in the tree yet are the first to be executed, while the root is an ancestor of the other nodes in the tree yet is last to be executed, which is precisely backward from the notion of ancestor and descendant in a DAG. Since the elimination tree terminology is much more established and pervasive in the sparse matrix field, we will use the terms parent/child and ancestor/descendant in the “tree” sense throughout this paper.

The structure of the elimination tree gives an indication of the potential parallelism in sparse Cholesky factorization. Roughly speaking, the height of the tree determines

the longest serial path in the column task graph and the width of the tree determines the degree of concurrency available (these notions will be made more precise later). Thus, a wide tree has many tasks that can be executed simultaneously, and a short tree has a relatively small parallel execution time. The structure of the elimination tree for a given matrix  $A$  is strongly affected by the particular ordering chosen for the matrix. For example, nested dissection orderings tend to produce short and wide elimination trees that are good for parallel factorization, whereas bandwidth or profile reducing orderings tend to produce relatively tall and narrow elimination trees that are poor for parallel factorization. Another desirable property of the elimination tree for parallel execution is that it be well balanced, by which we mean that subtrees at the same level are reasonably uniform in size and require roughly the same amount of work. For example, on highly regular problems such as  $k \times k$  grids, some nested dissection orderings produce well balanced binary trees. Minimum degree orderings, on the other hand, often produce unbalanced elimination trees. Having a well balanced elimination tree is helpful in scheduling the column task graph so that the computational load is well balanced across processors. However, Geist and Ng [9] have developed a method for partitioning the work in an unbalanced elimination tree and scheduling it so that the computational load is well balanced across the processors.

In the computational experiments to be reported below, the software package Sparspak [6,15] is used to perform the preliminary symbolic processing of our test matrices, which are derived from  $k \times k$  grid problems. Sparspak is a sparse matrix software package designed to order, factor, and solve sparse systems of linear equations. We did not use the standard orderings from Sparspak on our test problems, however. As mentioned above, minimum degree orderings tend to produce unbalanced elimination trees. Moreover, the automatic nested dissection ordering in Sparspak uses a level structure to find separators. While this approach is effective for many purposes on a wide range of problems, for some highly regular problems such as  $k \times k$  grids with a nine-point operator it fails to identify the ideal separators that produce an optimally short, wide, and balanced elimination tree. Therefore, we used instead a version of nested dissection patterned after [10] that takes advantage of the special structure of rectangular grid problems to yield the theoretically "correct" sequence of separators. After the ordering of the matrix  $A$  has been completed, a symbolic factorization is performed on the ordered matrix to obtain the structure of the Cholesky factor matrix  $L$ , from which we can construct the column task graph. We are then ready to begin our exploration of maximum speedup.

## 2.2. An Example

As an example of the ideas presented thus far, consider the  $10 \times 10$  matrix  $A$  whose nonzero entries are denoted by  $\times$  in Figure 2. Since the matrix  $A$  is symmetric, we concern ourselves only with its lower triangular structure, and we assume that it has already been ordered. Symbolic factorization yields the structure of the factor matrix  $L$  shown in Figure 3, where fill entries resulting from the factorization are denoted by  $+$ . The elimination tree for this example is shown in Figure 4. Using Liu's representation, we arrive at the column task graph shown in Figure 5. Next to each *cdiv* node and

$$\begin{pmatrix} \times & & & & & & & & & \\ & \times & & & & & & & & \\ & \times & \times & & & & & & & \\ \times & \times & & \times & & & & & & \\ & & & & \times & & & & & \\ & & & & \times & \times & & & & \\ & & & & & & \times & & & \\ & & & & & & \times & \times & & \\ & & & & & & \times & \times & \times & \times \\ & & \times & \times & \times & & & & & \times & \times \end{pmatrix}$$

Figure 2: Lower triangular structure of a  $10 \times 10$  symmetric matrix  $A$ .

$$\begin{pmatrix} \times & & & & & & & & & \\ & \times & & & & & & & & \\ & \times & \times & & & & & & & \\ \times & \times & + & \times & & & & & & \\ & & & & \times & & & & & \\ & & & & \times & \times & & & & \\ & & & & & & \times & & & \\ & & & & & & \times & \times & & \\ & & & & & & \times & \times & \times & \times \\ & & \times & \times & \times & + & & & & \times & \times \end{pmatrix}$$

Figure 3: Structure of factor matrix  $L$  for example in Figure 2.

*cm* edge in the column task graph is a value in parentheses that indicates the number of floating point operations involved in that column operation. The calculation of these values is discussed in chapter 3.

Now, consider the Cholesky factorization of the matrix  $A$ . From the figures we observe some important facts about the progression of the factorization of this matrix. Since  $l_{21}$  is zero, column 2 is not affected by column 1, and hence the computation of column 2 need not await the completion of column 1. On the other hand, since  $l_{32}$  is nonzero, column 3 depends on column 2, and therefore the computation of column 3 must await the completion of column 2. Similarly, we can continue this type of analysis for all columns of the matrix  $L$ . In terms of the elimination tree (Figure 4), we see that column  $i$  affects a subset of the columns that are ancestors of node  $i$ , and the *completion* of the columns (i.e., the *cdiv* operations) corresponding to the nodes along a path to the root must be computed sequentially in the given order. For nodes that are on independent branches of the tree and do not affect each other, such as nodes 1, 2, 5, and 7, the corresponding columns of  $L$  can be computed in parallel. Completing a column of  $L$  by performing its *cdiv* operation corresponds to removing that node from

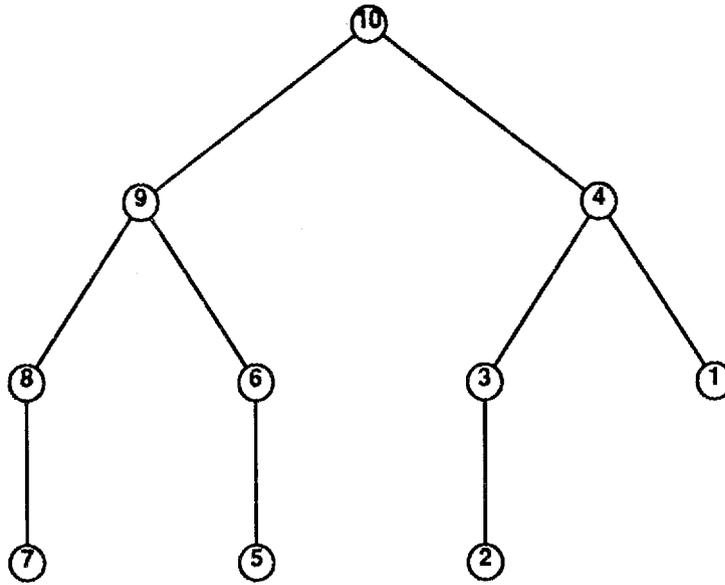


Figure 4: Elimination tree of example matrix.

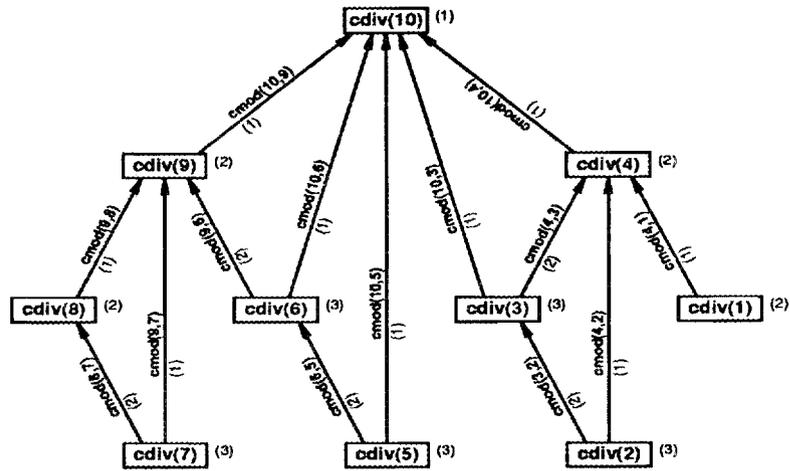


Figure 5: Column task graph of example matrix.

the tree. At the first step of the factorization, all of the leaf nodes are removed (recall that we are assuming an unlimited number of processors). This elimination results in the creation of more leaf nodes, and the factorization continues. At each stage, the *cdiv* operations for all of the current leaf nodes can be computed in parallel.

The parallel execution of multiple *cdiv* tasks is possible only in the sparse case; this type of parallelism is not available in factoring dense matrices (for a dense matrix, the elimination tree is a linear chain). For both sparse matrices and dense matrices, however, many *cmod* operations can potentially take place in parallel. Thus, in our example, *cmod*(3,2) and *cmod*(4,2) can take place simultaneously, even though *cdiv*(3) and *cdiv*(4) must be computed sequentially, with *cdiv*(3) preceding *cdiv*(4). These precedence relations among the column tasks are shown pictorially in the column task graph (Figure 5). To be fully effective, a parallel sparse factorization algorithm should exploit both types of parallelism: simultaneous *cdiv* operations on multiple leaf nodes and simultaneous *cmod* operations where possible.

### 3. DETERMINATION OF MAXIMUM SPEEDUP

After obtaining the structure of  $L$ , we can count the number of floating point operations required for each *cdiv* and *cmod* column operation. In the *cdiv* we consider each scalar division of an element of the column as one floating point operation. Thus, to calculate the number of operations required for *cdiv*( $k$ ), we simply count the number of nonzeros, including the diagonal element, in column  $k$  of the matrix  $L$ . Each scalar multiply/subtract pair in a *cmod* is also considered as one floating point operation. The number of floating point operations required for a *cmod*( $j, k$ ),  $j > k$ , is calculated by counting the number of nonzero entries in column  $k$  of  $L$  on and below row  $j$ . The total amount of work involved in the factorization is the sum of all of the *cdiv* and *cmod* column operations.

These individual floating point operation counts for the *cdiv* and *cmod* column operations are maintained as weights for the nodes and edges, respectively, of the column task graph. Three of the strategies to be described in the next section use these weights to calculate the length of the longest serial path in the column task graph representing the factorization. A fourth strategy, rather than using the individual weights, instead assumes a unit cost per *cdiv* or *cmod*, but also incorporates communication costs into the calculation of maximum speedup.

Once the length of the longest serial path and the total amount of work have been determined, we are ready to calculate our estimate of maximum speedup. We calculate maximum speedup according to the third definition for average parallelism of Eager, Zahorjan, and Lazowska [8]:

$$\text{maximum speedup} = (\text{total work}) / (\text{length of longest path})$$

This theoretical bound ignores the performance-degrading effects of communication delays, synchronization overhead, poor load balancing, etc. The effects of these factors will be examined in section 3.4.

The general problem of scheduling an arbitrary task graph for optimal parallel execution is another very difficult combinatorial problem (again, an NP-complete problem)

[26]. Thus, we seek heuristic scheduling strategies that provide an approximation to the longest serial path in the column task graph, whose length is required for computing maximum speedup. Initially, we consider three strategies that neglect communication costs, concentrating instead simply on the potential parallelism in executing the various column operations simultaneously, without regard for delays in propagating any data that might be required from other processors. Our strategies apply a depth-first search to either the elimination tree or the column task graph of  $L$ . The three different strategies result from different restrictions placed on the parallelism allowed in executing the column operation tasks. These restrictions result in different actions taken as each node is visited during the depth-first search. Two of the strategies serve as upper and lower bounds for the length of the longest serial path, while the third gives an intermediate estimate.

### 3.1. Strategy 1

Our first strategy is the most optimistic. It assumes that a given  $cdiv(j)$  task can be executed as soon as  $cmod(j, k)$  has been completed, where node  $k$  is the final descendant of node  $j$  in the elimination tree to be completed. Thus, this strategy assumes that any other required  $cmod(j, i)$  tasks, corresponding to any other descendants of node  $j$ , will have already been completed by this point in the execution of the algorithm, which may not be realistic in practice due to limited computational resources or communication delays. This strategy can be interpreted as placing no restriction on which processor can execute a given task. Thus, this optimistic strategy provides a lower bound on the length of the longest serial path, and hence an upper bound on maximum speedup.

Given the assumptions in Strategy 1, we compute the longest serial path by applying depth-first search to visit all nodes of the elimination tree. The total weighted path length is computed using the  $cdiv$  and  $cmod$  weights previously computed. The recursive depth-first search begins at the root of the tree. The following *visit* procedure is applied upon reaching a leaf node, and subsequently to the other nodes as the algorithm backtracks out of the recursion. We use the notation  $t(v)$  to denote the cumulative weight at node  $v$ .

```

visit(v)
    t(v) = 0
    let v have children  $v_1, \dots, v_k$ 
    for  $i = 1$  to  $k$ 
         $t(v) = \max(t(v), t(v_i) + cmod(v, v_i))$ 
    endfor
     $t(v) = t(v) + cdiv(v)$ 

```

Using this algorithm on our small example, we obtain the weighted elimination tree shown in Figure 6, where the cumulative weights  $t(v)$  of each node are shown in square brackets beside the node. Each of the brackets contains two values in the form  $[a, b]$ , where  $a$  is the incremental contribution of that node, and  $b$  is the length of the longest path at that point in the tree. By examining this weighted tree, we see that the length of the longest serial path for the example problem is 14.

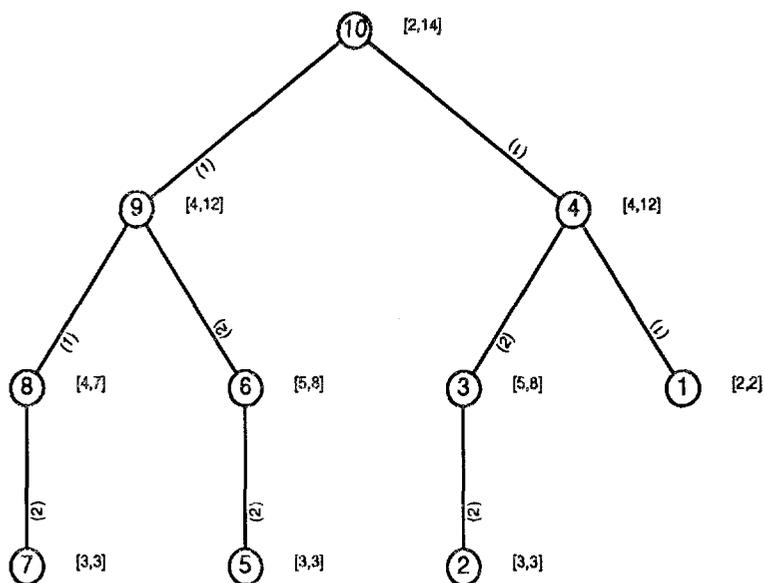


Figure 6: Weighted elimination tree for Strategy 1.

### 3.2. Strategy 2

Our second strategy is the most pessimistic, severely restricting the possible parallelism in the factorization. It assumes that a given  $cdiv(j)$  operation and all of its emanating  $cmod(i, j)$  operations must be done sequentially, which would be the case, for example, if the  $cdiv$  and resulting  $cmod$  operations were all done by the same processor. This extremely pessimistic approach foregoes one of the principal sources of parallelism in matrix factorization, namely the simultaneous execution of multiple  $cmod$  operations emanating from a single column, and thereby provides us with a lower bound on the speedup that can be expected in practice. In computing the longest serial path using Strategy 2, we apply depth-first search to visit the nodes of the column task graph, again using the  $cdiv$  and  $cmod$  weights previously computed. The *visit* procedure applied at each node is as follows, where again  $t(v)$  denotes the cumulative weight at node  $v$ .

```

visit(v)
  t(v) = 0
  let v have children v1, ..., vk then
  for i = 1 to k
    t(v) = max(t(v), t(vi))
  endfor
  t(v) = t(v) + cdiv(v) + Σtu,v ≠ 0 cmod(u, v)
  
```

The results of this strategy for our example are shown in Figure 7, where the bracket notation is as before. The length of the longest serial path obtained from this strategy is 16.

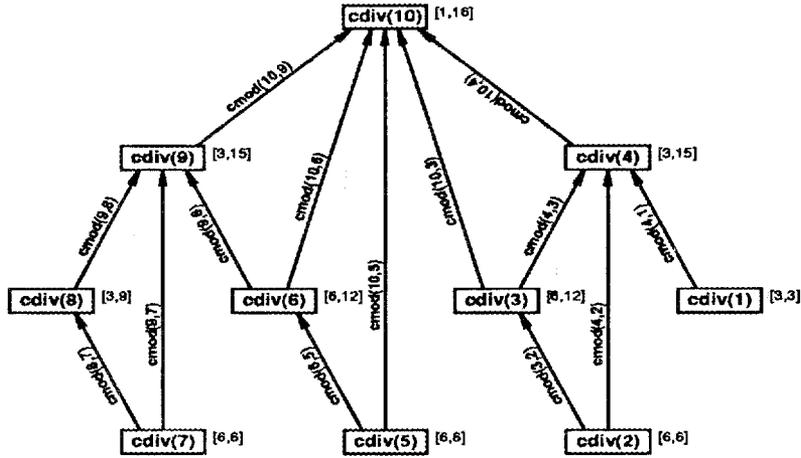


Figure 7: Weighted column task graph of Strategy 2.

### 3.3. Strategy 3

Our third strategy places only a mild restriction on the possible parallelism in the factorization. It assumes that a given  $cdiv(j)$  operation and all of its incoming  $cmod(j, i)$  operations must be done sequentially, which would be the case, for example, if the  $cdiv$  and immediately preceding  $cmod$  operations were all done by the same processor. Superficially, this third strategy may seem similar to the pessimistic Strategy 2, but it is in fact quite optimistic. In particular, since all of the incoming  $cmod(j, i)$  operations are updating the same column, they would have to be done sequentially anyway to maintain data integrity. Moreover, some of the  $cmod$  operations can be computed while waiting for the  $cdiv$  operations that provide the data for other  $cmod$  operations to be completed. Thus, this strategy provides an estimate for the longest serial path, and hence for maximum speedup, which should lie between those provided by the first two strategies. While Strategy 3 does not assume that all “earlier”  $cmods$  will have been completed before the final  $cdiv(i)$  upon which  $cdiv(j)$  depends, in practice this is often the case, so that Strategy 3 often gives similar results to the optimistic Strategy 1.

In implementing Strategy 3 we again apply depth-first search to the column task graph. If there are few ties in the path lengths as we proceed up the graph, then the results for Strategy 3 resemble those for Strategy 1. As before, we use the previously computed  $cdiv$  and  $cmod$  weights, and our depth first search begins at the root of the graph, with notation as before.

$visit(v)$

$t(v) = 0$

let  $v$  have children  $v_1, \dots, v_k$  then

sort children so that  $t(v_1) \leq t(v_2) \leq \dots \leq t(v_k)$

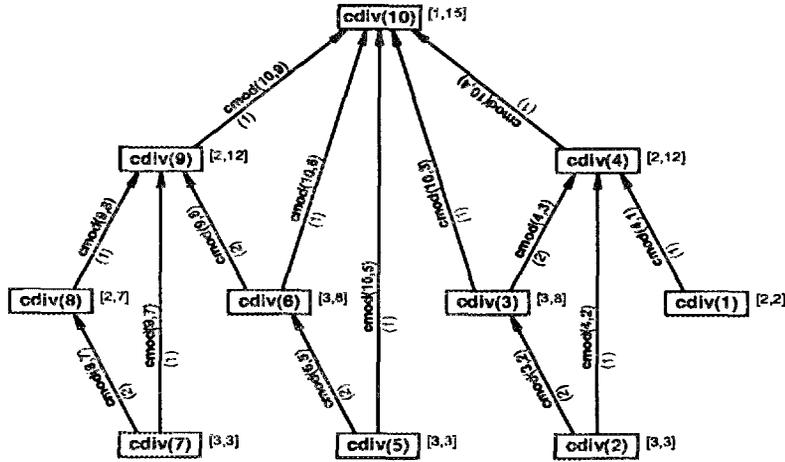


Figure 8: Weighted column task graph for Strategy 3.

```

for  $i = 1$  to  $k$ 
   $t(v) = \max(t(v), t(v_i)) + \text{emod}(v, v_i)$ 
endfor
 $t(v) = t(v) + \text{cdiv}(v)$ 
  
```

The results of this strategy for our example are shown in Figure 8. It reveals a longest serial path length of 15.

### 3.4. Strategy 4

Unlike the first three strategies, our fourth strategy takes into account communication delays in making results produced by earlier tasks available to later tasks that may require such data. The specific approach we use is due to Papadimitriou and Yannakakis [26]. The intent of [26] is to provide a simple, architecture-independent method for evaluating the performance of any algorithm on any parallel computer. The algorithm is represented by a directed acyclic graph (DAG), with the computational subtasks as its nodes and precedence constraints or data dependencies between tasks as its edges. As in [8], this analysis assumes that sufficiently many processors are available to handle the width of the DAG (i.e., the potential parallelism is not limited by any fixed number of processors).

As we have seen, a lower bound on the parallel completion time for a DAG is determined by its longest serial path. However, our earlier methods for determining the length of this path did not take into account communication delays in propagating results along the path before tasks that need these results can begin execution. In [26], the communication delay between tasks is measured in units of elementary processor steps, which is conveniently expressed as the “message-to-instruction” ratio, denoted

by  $\tau$ . Thus, the communication delay is expressed as a multiple of the time required for an elementary computational task. There are in fact two models given in [26], one in which all tasks are of unit size and the communication delay is a fixed constant given by  $\tau$ , and another in which both the task sizes and communication delays vary as a function of the amount of computation and sizes of messages, respectively.

The heart of the approach of [26] is an approximation algorithm for solving the problem of scheduling the DAG for parallel execution. This approximation algorithm is shown in [26] to produce a scheduling of the DAG that is within a factor of two of being optimal in the time to complete execution of the DAG. Our interest is not in the schedule itself, but in the longest serial path that it implies, thereby giving us an additional estimate of maximum speedup that, unlike our previous strategies, incorporates communication delays.

For the first model, with computational tasks of unit cost and constant communication delays, the approximation algorithm given in [26] is relatively simple, quite comparable in its complexity to the algorithms for implementing the previous three strategies given above. This simple model is mainly intended to address fine-grain computations, in which the computational tasks are individual arithmetic operations and messages consist of individual numbers. The second model, in which both computational tasks and messages are allowed to vary in cost, leads to a generalization of the approximation algorithm that is substantially more complex. The second model is intended for relatively coarse-grained computations in which the computational tasks require several arithmetic operations and the messages consist of several numbers, and both quantities vary in size from task to task. Strictly speaking, the second model is obviously more applicable to our medium-grained approach to sparse matrix factorization. However, we found the simplicity and elegance of the approximation algorithm for the first model to be much more appealing, and much more in keeping with the spirit of our first three strategies for estimating maximum speedup. Moreover, as we will see in section 4, the simpler model proved to be adequate for explaining the observed results for sparse Cholesky factorization. We therefore make some simplifying assumptions that enable us to apply the basic approximation algorithm to our problem.

The basic approximation algorithm of [26] assumes the following:

1. a directed acyclic graph whose nodes are computational tasks requiring equal execution time;
2. arcs in the graph representing time precedence and functional dependence; and,
3. a positive integer  $\tau$  that measures the communication delay relative to the cost of the computational tasks.

For the purposes of implementing this strategy, we take the column task graph of  $L$  as a representation of the DAG, which includes both the *cdiv* and *cmmod* column operations. We assume a uniform “average” cost per column task, and a constant communication delay  $\tau$  (implicitly assuming a fixed “average” message size). These assumptions would be significantly in error for dense matrix factorization, since the computational tasks and message sizes vary by a factor of  $n$  over the course of the factorization. However, they are not grossly in error for sparse matrix factorization, since

the columns get shorter but tend to become more dense as the factorization proceeds. In this setting, the proper choice of  $\tau$  is somewhat problematic. Given the composite nature of the “average” message, it is not clear that the basic “communication-to-computation” ratio of a given architecture (i.e., the time to send one floating point number relative to the time to compute one floating point operation) is applicable, since the start-up cost for sending the message is amortized over a larger message size. Moreover, there are various communication protocols and message packetizing effects that come into play. In addition, the rate at which floating point operations can be sustained in a sparse matrix code is dependent upon the amount of indexing and indirect addressing required by the compact storage scheme. Therefore, in applying this strategy we will consider a range of plausible values for  $\tau$ .

Since the approximation algorithm can be implemented by a depth-first search of the task graph, we can express this algorithm in terms similar to our previous three strategies, again using the same notation.

```

visit(v)
  let v have descendants  $v_1, \dots, v_k$ 
   $i = \min(\tau + 1, k)$ 
  if  $i = k$  then
     $t(v) = k + 1$ 
  else
    sort descendants so that  $t(v_1) \geq t(v_2) \geq \dots \geq t(v_k)$ 
     $t(v) = t(v_i) + i$ 
  endif

```

For illustrative purposes, we will use the value  $\tau = 2$  for our example problem. Taking  $\tau = 2$  in the above algorithm, we obtain the weighted column task graph shown in Figure 9, where we have represented the *cmop* operations, as well as the *cdiv* operations, explicitly as nodes of the DAG, and the cumulative weights of the nodes are shown in parentheses beside each node. The value given in parentheses indicates the length of the longest path at that node, specifying in “computational units” when the execution of that task is completed. Examining this weighted column task graph, we see that the length of the longest serial path is 9 for the example problem.

For any value of  $\tau$ , the path length determined by the approximation algorithm, as well as the total amount of work, must be multiplied by the average number of floating point operations required by each task, which is about 1.8 for our small example, in order to determine the cost in units comparable to those we have used previously. However, since speedup is a ratio of costs, the particular value of the average cost does not enter into the results.

### 3.5. Summary of Results for Example Problem

The results of the four strategies applied to the example problem are given in the Table 1. As expected, we see that Strategy 3 gives results lying between those given by Strategies 1 and 2, which serve as upper and lower bounds, respectively, on maximum

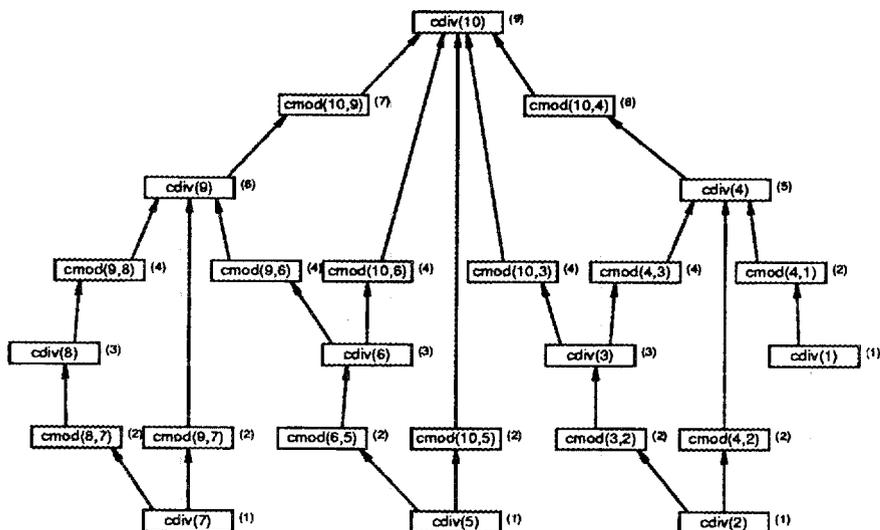


Figure 9: DAG for Strategy 4 ( $\tau = 2$ ).

Table 1: Summary of results for four strategies on example problem.

Approach	Total Work	Longest Path	Maximum Speedup
Strategy 1	43	14	3.0714286
Strategy 2	43	16	2.6875000
Strategy 3	43	15	2.8666667
Strategy 4	24	9	2.6666667

speedup, at least in the absence of communication delays. Given that it includes the additional time required for communication delays, Strategy 4 provides an estimate consistent with the other results.

#### 4. COMPARISON WITH OBSERVED SPEEDUPS

We now compare the estimates for maximum speedup determined by the four strategies outlined above to the speedups actually observed for sparse Cholesky factorization on an Intel iPSC/2 hypercube. For this purpose we obviously need some test problems and a parallel algorithm for computing the factorization. To date there have been three main types of approaches to developing practical parallel algorithms for sparse Cholesky factorization on distributed-memory architectures: fan-out [13], fan-in [3], and multifrontal [25]. For a direct comparison of these three schemes, see [5]. All three approaches are column-based and medium-grained, and therefore fit into the framework we have developed. The differences among the schemes amount to different ways of scheduling the work required for the factorization; in particular they amalgamate sub-

tasks and communications in different ways. The column task graph described earlier applies to each of the schemes in the sense that the temporal precedence relations hold among the subtasks it specifies. There is, however, no simple relationship between the arcs of the DAG and the messages actually sent when executing one of these parallel algorithms. For our numerical experiments, we have chosen to use the fan-in algorithm given in [3] and further refined in [4]. We have selected this algorithm because it is among the best performing available and it is the most easily accessible to us in the form of a working program for the Intel iPSC/2. The performance results we cite below were provided by Barry Peyton of Oak Ridge National Laboratory. The speedups cited are relative to the serial execution time on a single processor of the numeric factorization phase of Sparspak [6] for the same problem and ordering.

For our set of test problems we use a sequence of sparse matrices derived from a 9-point finite-difference operator on  $k \times k$  grids ordered by theoretical nested dissection. The  $k \times k$  grid problem is a standard model problem in sparse matrix computations because its sparsity pattern is representative of real (planar) applications and because its high degree of regularity lends itself to theoretical analysis. In addition, we have chosen to use grid problems and theoretical nested dissection orderings because they tend to yield better performance in parallel factorization than less regular problems and orderings, and we wish to understand the performance shortcomings of the factorization under the ideal conditions in which it should do best. Information about the test problems is given in the table 2. Symbolic factorization is used to determine the structure of the Cholesky factor  $L$ , from which we can determine the total number of floating point operations that are required for the factorization. The latter quantity is reported in the table as "total work."

Table 2: Characteristics of test problems.

grid size	no. of eqns.	nonzeros in $A$	nonzeros in $L$	total work
$3 \times 3$	9	49	30	70
$7 \times 7$	49	361	354	1592
$15 \times 15$	225	1849	2778	21552
$31 \times 31$	961	8281	17666	228312
$50 \times 50$	2500	21904	56949	1031185
$63 \times 63$	3969	34969	99450	2131928

Figure 10 shows the estimated maximum speedups for the sequence of test problems as a function of grid size using each of the four strategies discussed previously, together with the speedups observed for sparse Cholesky factorization on the Intel iPSC/2 hypercube. The speedups shown in the figure for the actual factorization are the *best* that were observed over several runs using various numbers of processors up to 64 (the best speedup is usually obtained for a larger number of processors as the problem size grows). We see that Strategies 1 and 2 indeed provide upper and lower bounds on the other speedups, except for very small grids whose cost is completely dominated by communication overhead, so that speedup is worse than predicted by even the most

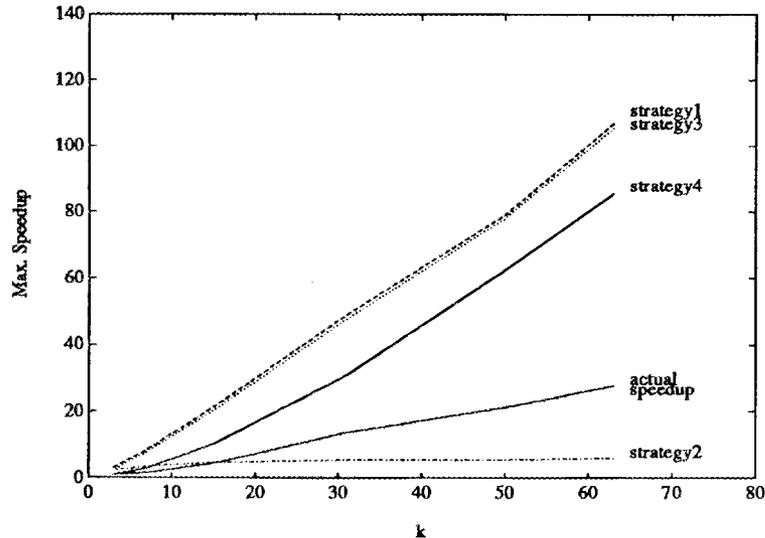


Figure 10: Comparison of results for various strategies with observed speedups for  $k \times k$  grid problems.

pessimistic strategy. We also note that Strategy 3 is almost as optimistic as Strategy 1.

The value used for  $\tau$  in generating the curve shown in Figure 10 for Strategy 4 was  $\tau = 59$ , which is the basic communication-to-computation ratio (sending one word to performing one flop) reported by Dunigan [7] for the Intel iPSC/2 hypercube. Since, as explained earlier, the choice of this value is somewhat arbitrary, we experimented with a range of choices for  $\tau$  in estimating speedups using Strategy 4. The results of this experiment are shown in Figure 11. The striking similarity of Figures 10 and 11 indicates that the model used in Strategy 4 is capable of subsuming all of the other models, as well as accurately modeling the observed speedups for the actual factorization, simply by choosing an appropriate value for the parameter  $\tau$ . These results suggest that the intent of Papadimitriou and Yannakakis in [26] to produce a model that effectively parameterizes this class of parallel architectures has been successfully realized, at least for this particular problem. Further, the results indicate that the Intel iPSC/2 executes the parallel sparse Cholesky factorization algorithm with an *effective* communication-to-computation ratio of about 500, which is an interesting fact about the machine itself, again for this specific problem.

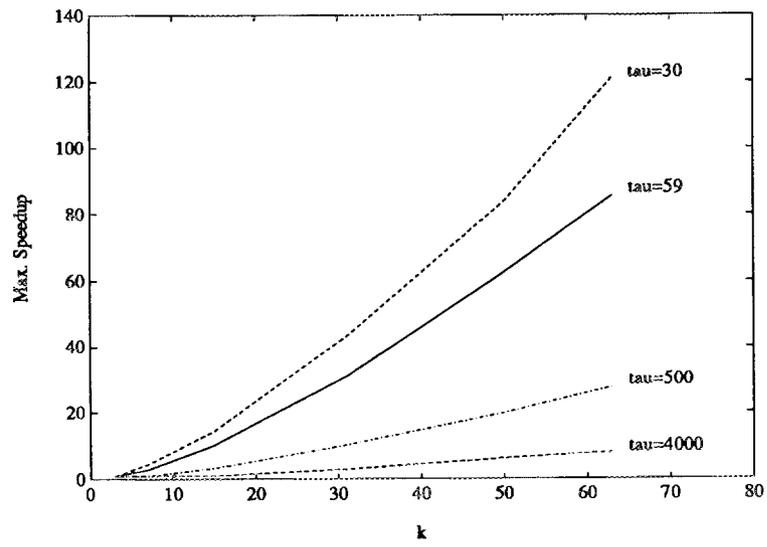


Figure 11: Estimated speedups for various values of  $\tau$ .

## 5. CONCLUSIONS

In this paper we have tried to explain the causes of the observed speedups in sparse Cholesky factorization on distributed-memory, message-passing parallel computers. To this end we developed and analyzed a number of theoretical models for determining the maximum speedup that could be expected for this problem. The first two strategies were based on rather extreme assumptions concerning the available parallelism, one very optimistic and the other very pessimistic, and these two strategies provided upper and lower bounds, respectively, on the maximum speedup. Unfortunately, the gap between these two bounds is too large for either to be of significant help in explaining the observed behavior of an actual parallel algorithm for sparse Cholesky factorization, which is not surprising considering that these models ignore any communication delays. A third strategy was based on assumptions of intermediate restrictiveness regarding possible parallelism, but still neglected communication costs and resulted in a very optimistic estimate of speedup.

A fourth strategy that takes explicit account of communication delays was based on an approximation algorithm given in [26] for scheduling an arbitrary DAG for parallel execution, which in turn leads to an estimate of the longest serial path and hence of maximum speedup. This model proved to be much more successful, and by appropriate choice of the communication parameter  $\tau$ , a full range of behaviors can be produced, including those of the previous theoretical models as well as the speedups observed for the actual parallel sparse Cholesky factorization algorithm. Since the parameter  $\tau$  enters the model specifically to characterize communication performance, this model indicates that a high degree of parallelism is attainable in solving this problem if communication is sufficiently fast. Moreover, the relatively poor performance observed in practice can be simulated in the model by assuming poor communication performance.

These results suggest that the answer to the question posed in the introduction is that the relatively poor performance of sparse Cholesky factorization to date on distributed-memory, message-passing parallel computers is primarily due to the poor communication performance of these machines relative to their floating point speed, rather than to insufficient parallelism in sparse factorization. For a number of reasons, however, this conclusion can only be regarded as tentative. First, we have experimented with a single algorithm (fan-in), a single class of highly regular test problems ( $k \times k$  grids), and a single ordering (theoretical nested dissection). Further experimentation with a wider variety of choices in all three areas is called for in future work. Second, the models and strategies we have employed involved a number of simplifying assumptions, heuristics, and approximations, and therefore cannot provide absolutely rigorous results. Finally, we observe that one should not read too much significance into the close match between observed performance and the model simulation. In particular, there is no necessary resemblance between the task schedule actually used by the fan-in algorithm and the schedule implicitly derived by the approximation algorithm of Strategy 4. Thus, we cannot rule out the possibility that the performance of the fan-in algorithm was determined as much by its choice of task schedule as by its communication requirements. Given this fact, one might ask why the approximation algorithm is not used in practical sparse factorization algorithms, and the answer is simple: al-

though the scheduling algorithm of Strategy 4 is simple to state, it is very expensive to execute for large problems. In fact, this scheduling algorithm is much more expensive than the factorization itself, and thus its value is for theoretical analysis rather than for practical computation.

Despite our inability to draw definitive conclusions based on our results thus far, we have nevertheless gained considerable insight into the factors affecting the performance of a complex and sophisticated algorithm on distributed-memory parallel architectures. Further analysis and experimentation along these lines should provide additional evidence to allow more rigorous conclusions, and may also help show the way to improved performance.

## 6. References

- [1] G.M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. *Proc. AFIPS*, 30:483–485, 1967.
- [2] G.M. Amdahl. Limits of expectation. *International Journal of Supercomputer Applications*, 2:88–94, 1988.
- [3] C. Ashcraft, S.C. Eisenstat, and J. W-H. Liu. A fan-in algorithm for distributed sparse numerical factorization. *SIAM Journal on Scientific and Statistical Computing*, 11:593–599, 1990.
- [4] C. Ashcraft, S.C. Eisenstat, J. W-H. Liu, B.W. Peyton, and A.H. Sherman. A compute-ahead implementation of the fan-in sparse distributed factorization scheme. Technical Report ORNL/TM-11496, Oak Ridge National Laboratory, Oak Ridge, TN, 1990.
- [5] C. Ashcraft, S.C. Eisenstat, J. W-H. Liu, and A.H. Sherman. A comparison of three column-based distributed sparse factorization schemes. Technical Report YALEU/DCS/RR-810, Yale University, New Haven, CT, 1990.
- [6] E.C.H. Chu, J.A. George, J. W-H. Liu, and E. G-Y. Ng. User's guide for SPARSPAK-A: Waterloo sparse linear equations package. Technical Report CS-84-36, University of Waterloo, Waterloo, Ontario, 1984.
- [7] T.H. Dunigan. Performance of the Intel iPSC/860 hypercube. Technical Report ORNL/TM-11491, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1990.
- [8] D.L. Eager, J. Zahorjan, and E.D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38:408–423, 1989.
- [9] G.A. Geist and E. Ng. Task scheduling for parallel sparse Cholesky factorization. *International Journal of Parallel Programming*, 18:291–314, 1989.
- [10] J.A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10:345–363, 1973.

- [11] J.A. George, M.T. Heath, and J. W-H. Liu. Parallel Cholesky factorization on a shared-memory multiprocessor. *Linear Algebra and Its Applications*, 77:165–187, 1986.
- [12] J.A. George, M.T. Heath, J. W-H. Liu, and E. G-Y. Ng. Solution of sparse positive definite systems on a shared memory multiprocessor. *International Journal of Parallel Programming*, 15:309–325, 1986.
- [13] J.A. George, M.T. Heath, J. W-H. Liu, and E. G-Y. Ng. Sparse Cholesky factorization on a local-memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9:327–340, 1988.
- [14] J.A. George and J. W-H. Liu. An automated nested dissection algorithm for irregular finite element problems. *SIAM Journal on Numerical Analysis*, 15:1053–1069, 1978.
- [15] J.A. George and J. W-H. Liu. The design of a user interface for a sparse matrix package. *ACM Transactions on Mathematical Software*, 5:134–162, 1979.
- [16] J.A. George and J. W-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.
- [17] J.A. George and J. W-H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31:1–19, 1989.
- [18] J.L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31:532–533, 1988.
- [19] M.T. Heath, E. Ng, and B.W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33, 1991. to appear.
- [20] J.A.G. Jess and H.G.M. Kees. A data structure for parallel L/U decomposition. *IEEE Transactions on Computers*, 31:231–239, 1982.
- [21] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5:308–371, 1979.
- [22] J. W-H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11:141–153, 1985.
- [23] J. W-H. Liu. Computational models and task scheduling for parallel sparse Cholesky factorization. *Parallel Computing*, 3:327–342, 1986.
- [24] J. W-H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.
- [25] R.F. Lucas. *Solving planar systems of equations on distributed-memory multiprocessors*. PhD thesis, Department of Electrical Engineering, Stanford University, 1987.

- [26] C.H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322–328, 1990.
- [27] O. Wing and J.W. Huang. A computational model of parallel solution of linear equations. *IEEE Transactions on Computers*, 29:632–638, 1980.
- [28] P.H. Worley. The effect of time constraints on scaled speedup. *SIAM Journal on Scientific and Statistical Computing*, 11:838–858, 1990.
- [29] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2:77–79, 1981.

**INTERNAL DISTRIBUTION**

- |        |                 |        |  |
|--------|-----------------|--------|--|
| 1.     | B. R. Appleton  | 23-27. | C. H. Romine   |
| 2-3.   | T. S. Darland   | 28.    | T. H. Rowan  |
| 4.     | E. F. D'Azevedo | 29-33. | R. C. Ward   |
| 5.     | J. J. Dongarra  | 34.    | P. H. Worley   |
| 6.     | G. A. Geist     | 35.    | A. Zucker  |
| 7-11.  | M. T. Heath     | 36.    | Central Research Library                             |
| 12.    | E. R. Jessup    | 37.    | ORNL Patent Office                                   |
| 13.    | M. R. Leuze     | 38.    | K-25 Plant Library                                   |
| 14.    | E. G. Ng        | 39.    | Y-12 Technical Library<br>Document Reference Station |
| 15.    | C. E. Oliver    | 40.    | Laboratory Records - RC                              |
| 16.    | G. Ostrouchov   | 41-42. | Laboratory Records Department                        |
| 17.    | B. W. Peyton    |        |  |
| 18-22. | S. A. Raby      |        |  |

**EXTERNAL DISTRIBUTION**

43. Cleve Ashcraft, Boeing Computer Services, P.O. Box 24346, M/S 7L-21 Seattle, WA 98124-0346
44. Donald Austin, 6196 EECS Bldg., University of Minnesota, 200 Union St., S.E., Minneapolis, MN 55455
45. Lawrence J. Baker, Exxon Production Research Company, P.O. Box 2189, Houston, TX 77252-2189
46. Jesse L. Barlow, Department of Computer Science, Pennsylvania State University, University Park, PA 16802
47. Edward H. Barsis, Computer Science and Mathematics, P. O. Box 5800, Sandia National Laboratory, Albuquerque, NM 87185
48. Chris Bischof, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 Cass Avenue, Argonne, IL 60439
49. Ake Bjorck, Department of Mathematics, Linkoping University, Linkoping 58183, Sweden
50. Jean R.S. Blair, Department of Computer Science, Ayres Hall, University of Tennessee, Knoxville, TN 37996-1301
51. James C. Browne, Department of Computer Sciences, University of Texas, Austin, TX 78712
52. Bill L. Buzbee, Scientific Computing Division, National Center for Atmospheric Research, P.O. Box 3000, Boulder, CO 80307

53. Donald A. Calahan, Department of Electrical and Computer Engineering, University of Michigan, Ann Arbor, MI 48109
54. John Cavallini, Acting Director, Scientific Computing Staff, Applied Mathematical Sciences, Office of Energy Research, U.S. Department of Energy, Washington, DC 20585
55. Ian Cavers, Department of Computer Science, University of British Columbia, Vancouver, British Columbia V6T 1W5, Canada
56. Tony Chan, Department of Mathematics, University of California, Los Angeles, 405 Hilgard Avenue, Los Angeles, CA 90024
57. Jagdish Chandra, Army Research Office, P.O. Box 12211, Research Triangle Park, NC 27709
58. Eleanor Chu, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
59. Melvyn Ciment, National Science Foundation, 1800 G Street NW, Washington, DC 20550
60. Tom Coleman, Department of Computer Science, Cornell University, Ithaca, NY 14853
61. Paul Concus, Mathematics and Computing, Lawrence Berkeley Laboratory, Berkeley, CA 94720
62. Andy Conn, IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598
63. John M. Conroy, Supercomputer Research Center, 17100 Science, Bowie, MD 20715-4300
64. Jane K. Cullum, IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598
65. George Cybenko, Center for Supercomputing Research & Development, 104 South Wright Street, Urbana, IL 61801-2932
66. George Davis, Department of Mathematics, Georgia State University, Atlanta, GA 30303
67. Tim A. Davis, CERFACS, 42 Avenue Gustave Coriolis, 31057 Toulouse Cedex, France
68. John J. Dornig, Department of Nuclear Engineering Physics, Thornton Hall, McCormick Road, University of Virginia, Charlottesville, VA 22901
69. Iain Duff, CSS Division, Harwell Laboratory, Didcot, Oxon OX11 0RA, England
70. Derek L. Eager, Department of Computational Science, University of Saskatchewan, Saskatoon, Saskatchewan, Canada
71. Patricia Eberlein, Department of Computer Science, SUNY at Buffalo, Buffalo, NY 14260
72. Stanley Eisenstat, Department of Computer Science, Yale University, P. O. Box 2158 Yale Station, New Haven, CT 06520
73. Lars Elden, Department of Mathematics, Linkoping University, 581 823 Linkoping, Sweden
74. Howard C. Elman, Computer Science Department, University of Maryland, College Park, MD 20742

75. Albert Erisman, Boeing Computer Services, P.O. Box 24346, M/S 7L-21, Seattle, WA 98124-0346
76. Geoffrey C. Fox, Booth Computing Center 158-79, California Institute of Technology, Pasadena, CA 91125
77. Paul O. Frederickson, RIACS, NASA Ames Research Center, Moffett Field, CA 94035
78. Fred N. Fritsch, L-300, Mathematics and Statistics Division, Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550
79. Robert E. Funderlic, Department of Computer Science, North Carolina State University, Raleigh, NC 27650
80. K. Gallivan, Computer Science Department, University of Illinois, Urbana, IL 61801
81. Dennis B. Gannon, Computer Science Department, Indiana University, Bloomington, IN 47401
82. Feng Gao, Department of Computer Science, University of British Columbia, Vancouver, British Columbia V6T 1W5, Canada
83. David M. Gay, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974
84. C. William Gear, Computer Science Department, University of Illinois, Urbana, IL 61801
85. W. Morven Gentleman, Division of Electrical Engineering, National Research Council, Building M-50 Room 344, Montreal Road, Ottawa, Ontario, Canada K1A 0R8
86. Alan George, Vice President, Academic and Provost, Needles Hall, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
87. John R. Gilbert, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304
88. Gene Golub, Computer Science Department, Stanford University, Stanford, CA 94305
89. Joseph F. Grear, Division 8331, Sandia National Laboratories, Livermore, CA 94550
90. John L. Gustafson, Ames Laboratory, 236 Wilhelm Hall, Iowa State University, Ames, IA 50011-3020
91. Per Christian Hansen, UCI\*C Lyngby, Building 305, Technical University of Denmark, DK-2800 Lyngby, Denmark
92. Richard Hanson, IMSL Inc., 2500 Park West Tower One, 2500 City West Blvd., Houston, TX 77042-3020
93. Robert M. Haralick, Department of Electrical Engineering, Director, Intelligent Systems Lab., University of Washington, 402 Electrical Engr. Bldg., FT-10, Seattle, WA 98915
94. Don E. Heller, Physics and Computer Science Department, Shell Development Co., P. O. Box 481, Houston, TX 77001
95. N. J. Higham, Department of Mathematics, University of Manchester, Gtr Manchester M13 9PL, ENGLAND

96. Charles J. Holland, Air Force Office of Scientific Research, Building 410, Bolling Air Force Base, Washington, DC 20332
97. Robert E. Huddleston, Computation Department, Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, CA 94550
98. Ilse Ipsen, Department of Computer Science, Yale University, P.O. Box 2158 Yale Station, New Haven, CT 06520
99. Lennart S. Johnsson, Department of Computer Science, Yale University, P. O. Box 2158 Yale Station, New Haven, CT 06520
100. Harry Jordan, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309
101. Barry Joe, Department of Computer Science, University of Alberta, Edmonton, Alberta T6G 2H1, Canada
102. Bo Kagstrom, Institute of Information Processing, University of Umea, 5-901 87 Umea, Sweden
103. Malvyn Kalos, Courant Institute for Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012
104. Hans Kaper, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439
105. Linda Kaufman, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974
106. Robert J. Kee, Applied Mathematics Division 8331, Sandia National Laboratories, Livermore, CA 94550
107. Kenneth Kennedy, Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77001
108. Tom Kitchens, Department of Energy, Scientific Computing Staff, Office of Energy Research, ER-7, Office G-236 Germantown, Washington, DC 20585
109. Richard Lau, Office of Naval Research, 1030 E. Green Street, Pasadena, CA 91101
110. Alan J. Laub, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106
111. Robert L. Launer, Army Research Office, P. O. Box 12211, Research Triangle Park, NC 27709
112. Charles Lawson, MS 301-490, Jet Propulsion Laboratory, 4800 Oak Grove, Pasadena, CA 91109
113. Peter D. Lax, Courant Institute of Mathematical Science, New York University, 251 Mercer Street, New York, NY 10012
114. Edward D. Lazowska, Department of Computer Science, Univ. of Washington, Seattle, WA
115. James E. Leiss, 13013 Chestnut Oak, Gaithersburg, MD 20878

116. John G. Lewis, Boeing Computer Services, P. O. Box 24346, M/S 7L-21, Seattle, WA 98124-0346
117. Jing Li, IMSL Inc., 2500 Park West Tower One, 2500 City West Blvd., Houston, TX 77042-3020
118. Heather M. Liddell, Director, Center for Parallel Computing, Department of Computer Science and Statistics, Queen Mary College, University of London, Mile End Road, London E1 4NS, England
119. Joseph Liu, Department of Computer Science, York University, 4700 Keele Street, Downsview Ontario, Canada M3J 1P3
120. Robert F. Lucas, Supercomputer Research Center, 17100 Science ve, Bowie, MD 20715-4300
121. Franklin Luk, Electrical Engineering Department, Cornell University, Ithaca, NY 14853
122. Thomas A. Manteuffel, Computing Division, Los Alamos National Laboratory, Los Alamos, NM 87545
123. Paul Messina, Mail Code 158-79, California Institute of Technology, 1201 E. California Blvd., Pasadena, CA 91125
124. James McGraw, Lawrence Livermore National Laboratory, L-306, P. O. Box 808, Livermore, CA 94550
125. Neville Moray, Department of Mechanical and Industrial Engineering, University of Illinois, 1206 West Green Street, Urbana, IL 61801
126. Cleve Moler, The Mathworks, 325 Linfield Place, Menlo Park, CA 94025
127. Brent Morris, National Security Agency, Ft. George G. Meade, MD 20775
128. Dianne P. O'Leary, Computer Science Department, University of Maryland, College Park, MD 20742
129. James M. Ortega, Department of Applied Mathematics, Thornton Hall, University of Virginia, Charlottesville, VA 22901
- 130-134. L. Susan Ostrouchov, Department of Computer Science, Ayres Hall, University of Tennessee, Knoxville, TN 37996-1301
135. Chris Paige, Department of Computer Science, McGill University, 805 Sherbrooke Street W, Montreal, Quebec, Canada H3A 2K6
136. Christos H. Papadimitriou, Department of Computer Science and Engineering, University of California, San Diego, CA 92093
137. Roy P. Pargas, Department of Computer Science, Clemson University, Clemson, SC 29634-1906
138. Beresford N. Parlett, Department of Mathematics, University of California, Berkeley, CA 94720
139. Merrell Patrick, Department of Computer Science, Duke University, Durham, NC 27706

140. Robert J. Plemmons, Departments of Mathematics and Computer Science, North Carolina State University, Raleigh, NC 27650
141. Jesse Poore, Department of Computer Science, Ayres Hall, University of Tennessee, Knoxville, TN 37996-1301
142. Alex Pothén, Department of Computer Science, Pennsylvania State University, University Park, PA 16802
143. Yuanchang Qi, IBM European Petroleum Application Center, P.O. Box 585, N-4040 Hafslund, Norway
144. Giuseppe Radicati, IBM European Center for Scientific and Engineering Computing, via del Giorgione 159, I-00147 Roma, Italy
145. John K. Reid, CSS Division, Building 8.9, AERE Harwell, Didcot Oxon, England OX11 0RA
146. Werner C. Rheinboldt, Department of Mathematics and Statistics, University of Pittsburgh, Pittsburgh, PA 15260
147. John R. Rice, Computer Science Department, Purdue University, West Lafayette, IN 47907
148. Garry Rodrigue, Numerical Mathematics Group, Lawrence Livermore National Laboratory, Livermore, CA 94550
149. Donald J. Rose, Department of Computer Science, Duke University, Durham, NC 27706
150. Edward Rothberg, Department of Computer Science, Stanford University, Stanford, CA 94305
151. Axel Ruhe, Department of Computer Science, Chalmers University of Technology, S-41296 Gotesborg, Sweden
152. Joel Saltz, ICASE, MS 132C, NASA Langley Research Center, Hampton, VA 23665
153. Ahmed H. Samel, Center for Supercomputing R&D, 1384 W. Springfield Avenue, University of Illinois, Urbana, IL 61801
154. Michael Saunders, Systems Optimization Laboratory, Operations Research Department, Stanford University, Stanford, CA 94305
155. Robert Schreiber, RIACS, MS 230-5, NASA Ames Research Center, Moffett Field, CA 94035
156. Martin H. Schultz, Department of Computer Science, Yale University, P. O. Box 2158 Yale Station, New Haven, CT 06520
157. David S. Scott, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006
158. Lawrence F. Shampine, Mathematics Department, Southern Methodist University, Dallas, TX 75275
159. Andy Sherman, Department of Computer Science, Yale University, P.O. Box 2158, Yale Station, New Haven, CT 06520

160. Kermit Sigmon, Department of Mathematics, University of Florida, Gainesville, FL 32611
161. Horst D. Simon, Mail Stop 258-5, NASA Ames Research Center, Moffett Field, CA 94035
162. Danny C. Sorensen, Department of Mathematical Sciences, Rice University, P. O. Box 1892, Houston, TX 77251
163. G. W. Stewart, Computer Science Department, University of Maryland, College Park, MD 20742
164. Paul N. Swartztrauber, National Center for Atmospheric Research, P.O. Box 3000, Boulder, CO 80307
165. Philippe Toint, Department of Mathematics, University of Namur, FUNOP, 61 rue de Bruxelles, B-Namur, Belgium
166. Hank Van der Vorst, Department of Techn. Mathematics and Computer Science, Delft University of Technology, P.O. Box 356, NL-2600 AJ Delft, The Netherlands
167. Charles Van Loan, Department of Computer Science, Cornell University, Ithaca, NY 14853
168. James M. Varah, Centre for Integrated Computer Systems Research, University of British Columbia, Office 2053-2324 Main Mall, Vancouver, British Columbia V6T 1W5, Canada
169. Robert G. Voigt, ICASE, MS 132-C, NASA Langley Research Center, Hampton, VA 23665
170. Phuong Vu, Cray Research Inc., 1408 Northland, Mendota Heights, MN 55120
171. Daniel D. Warner, Department of Mathematical Sciences, 0-104 Martin Hall, Clemson University, Clemson, SC 29631
172. Mary F. Wheeler, Rice University, Department of Mathematical Sciences, P.O. Box 1892, Houston, TX 77251
173. Andrew B. White, Los Alamos National Laboratory, P. O. Box 1663, MS-265, Los Alamos, NM 87545
174. Margaret Wright, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974
175. Mihalis Yannakakis, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974-2070
176. David Young, University of Texas, Center for Numerical Analysis, RLM 13.150, Austin, TX 78731
177. John Zahorjan, Department of Computer Science, University of California, Santa Barbara, CA 93106
178. Earl Zmijewski, Department of Computer Science, University of California, Santa Barbara, CA 93106
179. Office of Assistant Manager for Energy Research and Development, U.S. Department of Energy, Oak Ridge Operations Office, P.O. Box 2001, Oak Ridge, TN 37831-8600
- 180-189. Office of Scientific & Technical Information, P. O. Box 62, Oak Ridge, TN 37831