Sparse Cholesky Factorization on a Multiprocessor

Earl Zmijewski
Ph.D. Thesis

87-856
August 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

SPARSE CHOLESKY FACTORIZATION ON A MULTIPROCESSOR

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Earl Edward Zmijewski

August 1987

SPARSE CHOLESKY FACTORIZATION ON A MULTIPROCESSOR

Earl Edward Zmijewski, Ph.D.

Cornell University 1987

Systems of linear equations of the form $Ax = b$, where A is a large sparse symmetric positive definite matrix, arise frequently in science and engineering. The sequential computation of the solution vector x is well understood and many algorithms for this problem employ the following steps. First, try to reorder the rows and columns of A so that its Cholesky factor L is sparse. Next, determine the structure of L by symbolically factoring A and allocate storage for L. Finally, numerically factor A and then compute x by solving the triangular systems $Ly = b$ and $L^T x = y$.

In this thesis, we present parallel algorithms for the different steps of this computation. We design our algorithms for message-passing multiprocessors. The algorithms limit communication overhead and can solve problems that are too large to reside in the memory of any single processor. We provide numerical results based upon an implementation on an Intel hypercube.

We begin by presenting a parallel column-oriented sparse numeric Cholesky factorization algorithm. Then, viewing A as a graph, we develop a parallel graph partitioning algorithm that we use to order the columns of A and partition them among the processors. In addition to producing a sparse L, the resulting ordering and partitioning allows for parallelism and reduces communication overhead during the remaining phases of the computation. The parallel graph partitioning algorithm is based on the sequential Kernighan-Lin algorithm for finding small edge separators.

Since the computation of a particular column of L may depend on columns stored on several processors, the processors cannot operate independently. The elimination forest of A captures these dependencies and allows for efficient numeric factorization. We provide a parallel algorithm for computing the forest and prove its correctness. We also develop a parallel row-oriented symbolic factorization algorithm that uses the elimination forest.

Finally, we describe fast parallel forward and backward triangular solve algorithms. These algorithms solve for the components of x requiring information from other processors by using a variant of Li and Coleman's dense triangular solve algorithms.

To my parents, Irene and Milton, and to Laurie

Acknowledgements

I thank my adviser, John Gilbert, for his support and encouragement over the last three years. John was always able to suggest new avenues when I reached dead ends. His enthusiasm and depth of knowledge were inspiring. John taught me a lot about writing and contributed greatly to the readability of this thesis.

I thank Tom Coleman and Richard Shore for serving on my committee and for suggesting improvements to this thesis. Tom served as my unofficial adviser while John was on leave and assisted during my initial encounters with the Intel hypercube.

I am also grateful to Alan George, Michael Heath, Joseph Liu, and Esmond Ng for providing their sparse parallel matrix factorization code and for some interesting conversions when I was just starting this work. Tom Dunigan supplied the Oak Ridge hypercube simulator, which proved invaluable in debugging my code.

My parents, Irene and Milton, provided news from home during my long absences and moral support throughout my academic career. I was often able to preserve my sanity by visiting my parents, my sisters and brother, Barbara, Stephanie and Milton, and especially my nieces and nephews, Chrissy, Jackie, Jessica, Jenny, Michael, Patrick, Richard, and Teri.

A special thanks goes to my longtime officemate, Todd Knoblock. Todd taught me to appreciate Ithaca winters by teaching me how to downhill ski and often loaned me school supplies when I couldn't find my own.

Finally, I thank Laurie Hulbert for her love, support, and patience. Laurie read drafts of this thesis many times and suggested countless improvements and clarifications. She also enriched my life greatly by teaching me how to cook, cross country ski, bicycle tour, and numerous other things.

I also wish to express my gratitude for the generous financial support provided by a graduate fellowship from AT&T Bell Laboratories.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many problems in science and engineering require solving systems of linear equations. In matrix notation, this entails computing the solution vector x to the system

$$Ax = b,$$

where the coefficient matrix A and the right hand side vector b are both given. For many practical applications, A is an $n \times n$ large sparse symmetric positive definite matrix. In this case, we can use Cholesky factorization to factor A into a product of the form LL^T, where L is a lower triangular matrix. Then, we can easily compute x by solving the two triangular systems $Ly = b$ and $L^T x = y$. If most of the entries in A and L are zero, we can save considerable computer time and storage by manipulating and storing only the nonzeros. Unfortunately, a sparse A does not guarantee a sparse L. The set of positions that are nonzero in L and zero in A is known as *fill*. To reduce the amount of fill, one generally solves the equivalent system

$$(PAP^T)(Px) = Pb$$

for some $n \times n$ permutation matrix P. Since A is positive definite, no pivoting is required to maintain numerical stability and, hence, we are free to choose P solely on the basis of fill and algorithm design considerations.

The different aspects of Cholesky factorization have been extensively studied with regard to single processor systems [22] and many algorithms for this problem employ the following four steps. First, find the matrix P and form PAP^T. This operation is known as a *reordering* of A. Next, determine the fill by symbolically factoring A and then use it to allocate storage for L. Using this storage, numerically factor A and then find x by solving the appropriate triangular systems.

With the speeds of sequential machines reaching their theoretical limits, computer manufacturers are increasingly turning to parallel architectures to obtain higher execution rates. In recent years, dozens of new parallel machines have been built using a wide variety of architectures and numbers of processors, ranging from the 4 processor Cray X-MP to the 65,536 processor Thinking Machines hypercube. Algorithm designers face the enormous problem of redesigning sequential codes to effectively exploit the parallel processing capabilities of these machines.

In this thesis, we develop parallel algorithms for all four phases of sparse Cholesky factorization. We design our algorithms for message-passing multiprocessors and implement them on an Intel hypercube. On message-passing multiprocessors, computations are performed by dividing the work into several tasks and then assigning the tasks to the processors. We define a task as the computation of a single column of L. If a processor is assigned column i of A, it is responsible for computing column i of L and then sending it to all the processors that need it. We design all of our algorithms to limit communication overhead and to solve problems that are too large to reside in the memory of any single processor.

In this chapter, we provide background material and introduce some terminology; we conclude with a column-oriented sparse numeric Cholesky factorization algorithm.

Before performing the factorization, we must decide how to partition the columns among the processors and how to order them. We would like an ordering and partitioning that not only allows for parallelism, but also reduces the fill and communication overhead that occurs during the factorization. In Chapter 2, we develop a parallel algorithm for this problem that is based on the sequential Kernighan-Lin algorithm for finding small edge separators. We use the separators in both ordering the columns and partitioning them among the processors. The sizes and graph theoretic properties of these separators determine the amount of communication required during the remaining phases of the computation.

Of course, the processors cannot operate independently, since the computation of a particular column may depend on columns of L stored on several processors. In Chapter 3, we see that these dependencies are given by the elimination forest of A. We provide a parallel algorithm for computing this forest and prove its correctness.

Chapter 4 compares two very different parallel symbolic factorization algorithms. Their behavior and running times depend on the size of the problem and the way the columns are distributed among the processors.

Finally, Chapter 5 provides fast forward and backward triangular solve algorithms. These algorithms solve for the components of x requiring information from other processors by using a variant of Li and Coleman's dense triangular solve algorithms [35,36].

1.1 Graph Theory and Cholesky Factorization

We begin this section by defining a few graph theoretic terms that we use throughout this thesis. Then, we briefly describe the relation between graph theory and sparse Cholesky factorization. In examining the various phases of sparse Cholesky factorization, we cast many of the problems encountered in graph theoretic terms and then use graph algorithms to attack them. Parter [46] was among the first to suggest this approach and since then many others have used it to solve a wide variety of sparse matrix problems. For a more detailed treatment of the graph theoretic approach to sparse Cholesky factorization, the reader is referred to George and Liu [22]. Harary [28] provides a general introduction to graph theory.

An ***undirected graph*** $G = (V, E)$ consists of a set V of ***vertices*** and a set E of ***edges***. An edge (v, w) is an unordered pair of distinct vertices. Vertices v and w are ***adjacent*** if $(v, w) \in E$. Edge (v, w) is ***incident*** on vertices v and w. The ***adjacency set*** of a vertex v is $adj(v) = \{w \in V \mid (v, w) \in E\}$. Furthermore, if S is a set of vertices, its adjacency set is $adj(S) = \{w \in V - S \mid (v, w) \in E \text{ for some } v \in S\}$.

A graph $G' = (V', E')$ is a *subgraph* of the graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. The set of edges with both end points in V' is denoted by $E(V')$. If $E' = E(V')$ then G' is the subgraph of G *induced* by V'.

A *path of length k* from v to w in G is a sequence of vertices $v = v_0, v_1, \ldots, v_k = w$ such that $(v_i, v_{i+1}) \in E$ for $i = 0, 1, \ldots, k-1$ and the vertices $v_1, \ldots, v_k$ are all distinct. A *cycle* is a path with $v_0 = v_k$. If there exists a path between every pair of vertices of a graph, the graph is *connected.* The *connected components* of G (or just *components*) are the maximal connected subgraphs of G.

If $G = (V, E)$ is a graph, $V' \subseteq V$ is a *vertex separator* of G if removing V' from G divides G into two or more components. Likewise, $E' \subseteq E$ is an *edge separator* if its removal divides G into two or more components.

A *directed graph* $G = (V, E)$ consists of a set V of vertices and a set E of edges. Unlike undirected graphs, an edge $\langle v, w \rangle$ is an ordered pair of distinct vertices and is said to be directed from v to w. The same definitions concerning paths and cycles apply to directed graphs. A vertex v has *in-degree* equal to the number of edges $\langle w, v \rangle$ and *out-degree* equal to the number of edges $\langle v, w \rangle$.

A *directed tree* $T = (V, E)$ is an acyclic directed graph in which exactly one vertex, called the *root*, has out-degree 0 and the rest have out-degree 1. This implies that, for every vertex v, there is a unique path from v to the root. If $\langle v, w \rangle$ is an edge of T then v is a *child* of w and w is the *parent* of v. A vertex with no children is called a *leaf.* The *height* of vertex v is the length of the longest path from a leaf to v. The height of tree T is the height of its root. If there is a path from v to w then v is a *descendant* of w and w is an *ancestor* of v. An *induced*

$$A = \begin{bmatrix} \times & \times & & \times & \times & \\ \times & \times & & & \times & \\ & & \times & \times & \times & \\ \times & & \times & \times & & \times \\ \times & \times & \times & & \times & \\ & & & \times & & \times \end{bmatrix}$$

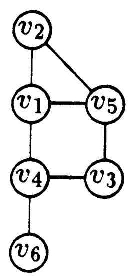
$G =$ 

Figure 1.1: A matrix A and its corresponding graph G.

subtree of T is an induced subgraph that happens to be a tree. The *subtree of T rooted at k* is a subtree of T induced by k and some of its descendants in T. We say that T is *heap ordered* if the vertices are linearly ordered in such a way that w is less than v whenever w is a proper descendant of v. If $G = (V, E)$ is a directed graph, a directed tree $T = (V, E')$ is a *spanning tree* of G if $E' \subseteq E$.

A *directed forest* is a directed graph that consists of one or more vertex-disjoint directed trees.

In this thesis, we concentrate on solving linear systems $Ax = b$, where A is an $n \times n$ large sparse symmetric positive definite matrix. Positive definiteness ensures that A has a nonzero diagonal. Since A is also symmetric, we can represent its structure using an undirected graph $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ and $(v_i, v_j) \in E$ if and only if $a_{ij} \neq 0$ and $i \neq j$. An *elimination order* of G is an ordering of the vertices of G, which we will write as a one-to-one function $\alpha : V \to 1, \ldots, n$. We define the graph $G_\alpha = (V, E_\alpha)$, where $(v_{\alpha(v_i)}, v_{\alpha(v_j)}) \in E_\alpha$ if and only if $(v_i, v_j) \in E$. Graph G_α is just G with its vertices relabeled by α. Since G is the graph of A, we can view α as a symmetric permutation of A. More formally, if for all i, column i of the permutation matrix P has its single nonzero in row $\alpha(i)$, then G_α is the graph of PAP^T. Hence, finding an elimination order of G corresponds to finding a symmetric permutation of A. This correspondence allows us to equate the structure of a matrix with its graph and solve structural problems with graph algorithms. Figure 1.1 contains a symmetric matrix structure, where each nonzero is denoted by an X, and its corresponding graph.

One way to compute the structure of L is to carry out a structural version of numeric Cholesky factorization on the structure of PAP^T. We get such a version of the algorithm by replacing numerical operations with Boolean ones. (For more information on structural matrix algorithms, see Coleman, Edenbrandt, and Gilbert [6,11].) We define the *filled graph* $G^*_\alpha = (V, E^*_\alpha)$ of G to be the graph of the structure of L computed by a structural factorization of PAP^T. Rose, Tarjan, and Lueker [50] provided a way of determining the edges in G^*_α directly from G and α without actually performing the structural factorization.

Lemma 1.1 Edge $(u, v) \in E^*_\alpha$ if and only if there is a path $u = v_1, v_2, \ldots, v_k = w$ in G with $\alpha(v_i) < \min(\alpha(u), \alpha(w))$ for $1 < i < k$.

That is, there is an edge (u, v) in G^*_α if and only if there is a path in G_α from u to v through vertices numbered lower than both u and w. An immediate corollary of Brayton, Gustavson, and Willoughby's work [2] is that for every symmetric matrix structure, there are numerical values that can be assigned to the nonzeros that make the matrix positive definite and result in no numerical cancellation during factorization. For such matrices, the structural factorization algorithm correctly predicts the structure of L, and $L + L^T$ is the adjacency matrix of G^*_α. Thus, finding a P to reduce the amount of fill in L corresponds to finding an α to reduce the number of edges in G^*_α.

1.2 Message-passing Multiprocessors

In what follows, we develop a number of parallel algorithms for sparse Cholesky factorization. All of them are designed for a class of message-passing multiprocessors typified by the currently available hypercube machines of Ametek, Intel, and NCUBE [56]. These machines consist of several identical processors, each containing some local memory. They coordinate their activities by passing messages along a network of communication links. On these machines, the number of processors is typically quite a bit smaller than the size of the problem we want to solve, and communication is considerably slower than computation. Therefore we seek algorithms that do as much computation as possible locally, and use the least possible amount of communication. Our only assumption about the topology of the communication network is that any processor can communicate efficiently with any other processor. See Feng [13] for a survey of network topologies.

We assume that the multiprocessor system supports two message passing primitives: *send* and *recv*. A processor executes a send to ship information to another processor. A processor uses a recv to receive information it has been sent. When a processor is sent a message, that message remains in a buffer until the processor executes a recv. If a processor executes a recv and no message is currently available, it waits for one to appear. Since processors will automatically forward

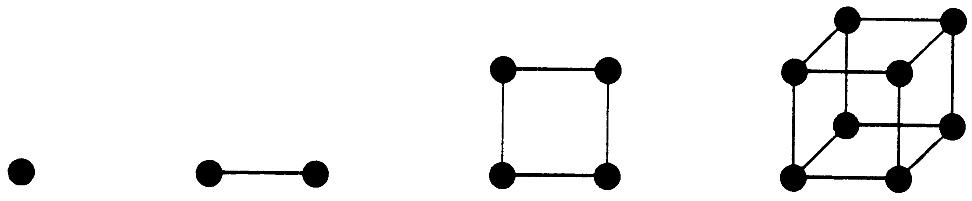


Figure 1.2: Hypercubes of dimension 0, 1, 2, and 3.

messages, two processors that want to communicate need not be directly linked in the network.

Hypercubes are one important class of message-passing multiprocessors that are both commerically available and relatively inexpensive. A 0-dimensional hypercube or 0-cube consists of a single processsor. To construct a 1-cube, take two processors (0-cubes) and place a single communication link between them. In general, to construct an $(n+1)$-cube, take two n-cubes and find a one-to-one correspondence between the processors in the two cubes. Then, place one communication link between each pair of corresponding processors. Figure 1.2 contains examples of low dimension hypercubes drawn as graphs, where each vertex represents a processor and each edge a communication link. Typically, at least one of the processors in the hypercube is connnected to an additional processor called the host. The user accesses the hypercube through the host, and although she can use the host in a computation, she commonly employs it only to send data to the processors of the hypercube and collect results.

It is easy to see that an n-cube contains 2^n processors, where each processor is linked to exactly n other processors. In an n-cube, the shortest distance between any two processors is at most n, and for any processor ϕ, we can find a spanning tree where ϕ is the root and n is the maximum distance from a leaf to ϕ. Using such a spanning tree, processor ϕ can broadcast a message to every other processor in time proportional to that required to sequentially send n messages between two adjacent processors. As each processor receives the broadcast message, it sends the message to its immediate descendants in the spanning tree in order of decreasing height. We will refer to this as a *fan-out* approach to broadcasting. In a similar manner, we can use a *fan-in* approach to collect information from all the processors onto a single processor. See Saad and Schultz [51,52,53] for more topological properities of hypercubes and communication algorithms. See Seitz [55] for a description of the first hypercube, the Caltech Cosmic Cube, and Wiley [56] for some characteristics of existing hypercubes.

All of the algorithms in this thesis have been coded in Fortran and run on the Cornell Theory Center's 16 processor Intel hypercube under XENIX 286 release 3.4 of the host operating system and iPSC release 3.0 of the node operating system. The algorithms also run on a Vax 780 under Berkeley Unix, using the Oak Ridge National Laboratories' hypercube simulator [10]. Although we present numerous timing results for the Intel hypercube in what follows, one should not read too much into these numbers. The Intel hypercube is an experimental machine whose hardware and software are in a constant state of flux. During the months we coded the algorithms in this thesis, there were several software releases and some of them had a dramatic effect on running times. In addition, since the communication primitives are at a very low level, running times are sensitive to the way the user incorporates communication in her programs. Hopefully, higher level communication primitives will eventually be built into the system. On the hardware side, Intel will soon release a version of the hypercube where each processor has a communications co-processor. This should decrease running times of any algorithm requiring nontrivial amounts of communication.

1.3 Dense Numeric Factorization

There are many ways of computing the Cholesky factorization of a symmetric matrix [22]. One method is known as ***column*** or ***inner product Cholesky***. In this scheme, the columns of L are computed in order. The j^{th} column is determined by first subtracting multiples of columns 1 through $j-1$ of L from the j^{th} column of A. The resulting column is then divided by the square root of its diagonal element to obtain the j^{th} column of L. The computation requires only the lower triangle of A. Note that once column j has been computed, appropriate multiples of it can be subtracted from columns $j+1$ through n of A at the same time. We will use this idea to develop a parallel version of column Cholesky for sparse matrices. We will not attempt to exploit parallelism within column operations; rather, the parallelism will result from computing the columns of L simultaneously on different processors. We assume that number of available processors p is no greater than the number of columns of A, namely n. We expect such multiprocessor systems to be used to factor large matrices, and hence, in practice, we expect $n \gg p$.

Parallel column Cholesky algorithms for dense matrices have already been developed for various architectures. George, Heath, and Liu [17] have implemented such an algorithm on a Denelcor HEP, a shared memory multiprocessor, and Geist and Heath [15,16] have done the same for an Intel hypercube. Experimental results on both machines indicate that the column Cholesky approach is a reasonable way to exploit the parallelism inherent in the computation.

Our sparse column Cholesky factorization algorithm is similar to Heath's algorithm [29] for dense matrices. Figure 1.3 contains a version of his algorithm, which we call *DenseCholesky*. It uses the following two subroutines.

- cmod(j,i) : subtracts the appropriate multiple of column i from column j where $i < j$.
- cdiv(j) : divides column j by the square root of its diagonal element.

For ease of presentation, *DenseCholesky* assumes that $n = p$ and each processor is assigned exactly one column of A. We let θ_k denote the processor assigned column k of A; it is this processor that is responsible for computing column k of L. Of course, in an actual implementation, we would allow more than one column on each processor, and we would ensure that each processor sends at most one copy of any column of L to another processor. However, there is little value in presenting the algorithm at this level of detail. To compute L, each processor executes the *DenseCholesky* algorithm. We assume that each processor knows the location of each column of A, i.e., each processor knows θ_k for all k. Note that a processor can receive its needed columns in any order.

```
processor $\theta_k$
    {Modify column $k$ by all preceding columns.}
    for $i := 1$ to $k - 1$ do
        recv col $j$;  {$j$ defined by sender}
        cmod($k, j$);
    od
    cdiv($k$);
    {Send column $k$ of $L$ to the required processors.}
    for $i := k + 1$ to $n$ do
        send the entries of col $k$ in rows $i$ through $n$ to processor $\theta_i$;
    od
```

Figure 1.3: The *DenseCholesky* algorithm.

```
processor $\theta_k$
    {Col $k$ is the data structure for column $k$ of $L$.}
    initialize col $k$ to contain the values in column $k$ of $A$;
    for $i := 1$ to nLrow[$k$] $- 1$ do
        recv col $j$;  {$j$ defined by sender}
        cmod($k$,$j$);
    od
    cdiv($k$);
    {Send column $k$ of $L$ to the required processors.}
    for each nonzero $i$ in (nzcol $k$ $-$ $\{k\}$) do
        send the nonzeros of col $k$ in rows $i$ through $n$ to processor $\theta_i$;
    od
```

Figure 1.4: The *SparseCholesky* algorithm.

1.4 Sparse Numeric Factorization

As in the dense case, we will use a column-oriented approach when A is sparse. (See Liu [40] for an examination of this and other approaches to sparse parallel Cholesky factorization.) We can use the *DenseCholesky* algorithm for sparse matrices; however, if processor θ_j executes cmod(j,i) and the j^{th} element of column i is zero, then column j will remain unchanged. That is, the cmod will multiply column i by 0 and subtract the result from column j. Under these circumstances, θ_j does not need to wait for column i in order to compute column j. In addition, θ_i need not send column i to θ_j. We can modify *DenseCholesky* to exploit this observation provided each processor knows the exact number of columns for which it must wait for each of its assigned columns. Namely, θ_j must know how many nonzeros are in the j^{th} row of L. For typical sparse matrices, computing this information first should speed up the computation of L and result in reduced message traffic.

Figure 1.4 contains the *SparseCholesky* algorithm that uses the above observation, and is similar to code developed by George, Heath, Liu, and Ng [18]. We define

Definition 1.2 $Lrow(k) = \{j \mid l_{kj} \neq 0 \text{ and } j < k\}$.

The *SparseCholesky* algorithm assumes $nLrow[k]$ equals the size of $Lrow(k)$. For ease of presentation, we again assume that there is exactly one column per processor. In the worst case, *SparseCholesky* requires time proportional to that needed to multiply L and L^T, which is proportional to the time needed to factor A on a single processor. It passes at most $O(|L|)$ messages, containing a total of $O(|L|)$ entries of L and their corresponding locations. In Chapter 3, we define the elimination forest for A and show how to use it and the structure of A to compute the row structure of L, i.e., $Lrow(k)$ for each row k. Chapter 4 contains algorithms for computing the column structure of L and the entries of $nLrow$.

Chapter 2

Ordering

As noted in Chapter 1, the first step in computing the Cholesky factorization of an $n \times n$ symmetric positive definite matrix $A = (a_{ij})$ is to find a permutation matrix P to reorder A. Equivalently, we want to find an elimination order α for the graph $G = (V, E)$ of A. On single processor systems, one typically selects P solely to reduce fill. This is a good strategy since reducing fill, besides reducing the needed storage, also reduces the factorization time. On message-passing multiprocessors, defining a good ordering is more complicated. We want all of the processors to be busy throughout the factorization; that is, we want an ordering that allows for parallelism. Also, all hypercubes currently on the market require significantly more time to communicate a byte of data than to perform a floating point operation on that byte. Therefore, we also want to reduce the amount of communication needed during the factorization, perhaps even at the expense of more fill. Both the parallelism and communication in the computation depend not only on P but also on the placement of A on the processors. As we shall see in this chapter, it is possible to find a reordering of A and an assignment of its nonzeros to processors that results in good processor utilization during the factorization, while reducing both the fill and the communication.

Nested dissection is an ordering heuristic that reduces fill [22] and allows for parallelism [40,48]. Nested dissection begins by finding a vertex separator S of G whose removal would disconnect G into at least two components $C_1, \ldots, C_k$. It orders the vertices of S after those in $C_1, \ldots, C_k$. Then no edge in G^*_α can connect two vertices in different C_i, since any path in G between two such vertices must go through S. Besides reducing fill, this property also allows us to compute columns of L corresponding to vertices in different C_i in parallel [48]. To order the remaining vertices in V, we apply this procedure recursively to the subgraphs $C_1, \ldots, C_k$. Nested dissection orderings produce low fill if each separator is small and the components it divides its subgraph into are all roughly the same size. For example, planar graphs, two-dimensional finite element graphs, and graphs of bounded genus all have nested dissection orderings that produce at most $O(n \log n)$ fill [25,37].

In what follows, we define narrow and wide vertex separators and develop a parallel ordering algorithm that uses either type of separator to order the columns of A and assign them to processors. In a different setting, Liu [43] used both of these separators to order grid graphs and analyzed the parallelism that results during outer product Cholesky factorization. Our parallel ordering algorithm is based on a simple modification of the Kernighan-Lin algorithm [31] for finding edge separators on a single processor. Gilbert and Zmijewski [26] first reported on the ideas in this chapter.

George, Liu, and Ng [18,19,20,23] independently made many of the observations in this section and implemented a different parallel ordering algorithm. In their algorithm, each processor uses Sparspak's nested dissection routines [22] to order a part of the matrix; the host then computes the elimination forest and uses it to reassign columns to processors. Unlike our algorithm, each processor must have enough memory to store the adjacency structure of all of A.

2.1 The Kernighan-Lin Algorithm

In this section, we briefly review the Kernighan-Lin algorithm [31] for finding small edge separators on a single processor. We assume $G = (V, E)$ is an arbitrary graph with $2n$ vertices numbered from 1 to $2n$. Each edge (i, j) has a *cost* c_{ij}. Let $C = (c_{ij})$ be the cost matrix of G, where c_{ij} is the cost of (i, j) if it exists, and is 0 otherwise. We want to partition the vertices of G into two sets A and B of equal size, such that the total cost of all edges connecting vertices of A and B is minimized. In other words, we want to find a *minimum cost edge separator* that divides the vertices of G into two equal-sized sets. Note that if the costs are all one then a solution to this problem is an edge separator with the minimum number of edges. Although this problem is NP-complete, Kernighan and Lin have devised an iterative algorithm that works well in practice. In the remainder of this section, we will describe the central idea behind their algorithm.

Suppose the vertices of G are initially partitioned into two equal-sized sets, A and B, in some manner. Call an edge connecting a vertex of A to one of B an *external* edge; call any other edge an *internal* edge. Let T be the total cost of all the external edges. Kernighan and Lin's algorithm reduces T by repeatedly swapping equal-sized subsets of A and B. It selects the subsets to guarantee that T decreases at each iteration of the algorithm. Hopefully, the algorithm will quickly converge to a solution near the optimum.

Before explaining how the subsets to be swapped are chosen, we will need some notation. Define the *external cost* E_a of a vertex $a \in A$ to be the total cost of its incident external edges,

$$E_a = \sum_{x \in B} c_{ax}.$$

Similarly, define the *internal cost*

$$I_a = \sum_{x \in A} c_{ax}.$$

Let $D_a = E_a - I_a$. Following Kernighan and Lin, we will refer to D_a as "the D value of vertex a." Define the corresponding quantities for the vertices of B.

If we swap $a \in A$ and $b \in B$ then we can update T by subtracting

$$g = D_a + D_b - 2c_{ab},$$

where g is called the *gain* in swapping a and b. Swapping a and b may alter the D values of other vertices incident on a and b. These D values can be recalculated as follows.

$$D'_x = D_x + 2c_{xa} - 2c_{xb}, \quad x \in A - \{a\} \tag{2.1}$$

$$D'_y = D_y + 2c_{yb} - 2c_{ya}, \quad y \in B - \{b\} \tag{2.2}$$

Using these definitions, we can state the Kernighan-Lin algorithm as follows. First, unmark all the vertices of G and compute their initial D values with respect to the current partition, A and B. Then locate two unmarked vertices, $a \in A$ and $b \in B$, that would produce the largest gain if swapped. Do not swap these vertices, but simply mark them and update the D values of the unmarked vertices using Equations 2.1 and 2.2. Repeat this process of marking vertices and updating D values until no unmarked vertices remain. The result is a sequence of pairs $(a_i, b_i) \in A \times B$ of vertices and their associated gains g_i, for $i = 1, \ldots, n$. Note that the gains g_i can be positive or negative and that $\sum_{i=1}^{n} g_i = 0$. Finally, determine which vertices of A and B to swap by finding the smallest k that maximizes $G = \sum_{i=1}^{k} g_i$. If $G > 0$, swap vertices $a_1, \ldots, a_k$ of A with $b_1, \ldots, b_k$ of B and repeat this entire process. Otherwise, stop. Since $G = 0$, no further improvements are possible using this approach.

One important feature of this algorithm is that it does not terminate upon encountering a negative gain. Hence, during a single iteration, it may consider the effect of swapping a pair of vertices that would increase T. The algorithm will only swap these two vertices if it can locate other pairs of vertices that can be swapped to produce an overall decrease in T. Thus, negative gains are tolerated provided they ultimately result in a better edge separator.

In a straightforward implementation, one iteration of this algorithm requires $O(n^3)$ time on a single processor. If C is stored as a dense matrix, the time to compute the initial D values is $O(n^2)$. Since there are $O(n^2)$ possible pairs of vertices, locating the pair with the maximum gain takes $O(n^2)$ time. Updating the remaining D values also takes at most $O(n)$ time. Since the process of locating pairs of vertices of maximum gain and updating D values is repeated n times, one iteration of the entire algorithm requires at most $O(n^3)$ time.

Kernighan and Lin implemented two faster methods for selecting pairs of vertices with large gains. In the first, not all vertices are considered, but rather some small number of the vertices with the largest D values. Using this idea, one iteration of the algorithm requires $O(n^2)$ time, but will not always select the pair of vertices with the largest possible gain at each step. Another approach sorts the D values of all the vertices before looking for the best pair. Employing this method, one iteration still requires $O(n^3)$ time in the worst case; however, for nonnegative edge costs, the actual running time will typically be $O(n^2 \log n)$, the time required for n sorts.

Both methods perform well in practice. Kernighan and Lin tested a variety of graphs with up to 360 vertices and various edge densities. Using random initial partitions for these problems, they found that both implementations of their algorithm almost always converged in 2 to 4 iterations, and that the probability of a single iteration finding an optimal solution was approximately $2^{-n/30}$, where n was the number of vertices in the graph.

We conclude this section by noting that Kernighan and Lin proposed variants of their basic algorithm that can be used to partition the vertices of a graph into sets of different sizes or into more than two sets. In fact, the parallel algorithm in the next section is just a parallel version of one of their algorithms for partitioning the vertices of a graph into d sets, where d is a power of 2.

2.2 A Parallel Kernighan-Lin Algorithm

In this section, we assume that $G = (V, E)$ is an arbitrary graph whose vertices have been partitioned among $p \geq 2$ processors of a message-passing multiprocessor in some roughly even manner. We present a simple parallel version of the Kernighan-Lin algorithm for partitioning the vertices of G into p roughly equal-sized sets, each set residing on its own processor. Our goal is to produce a partition with few edges connecting vertices in different sets. Since our graphs will correspond to sparse matrices, each such edge represents a data dependency between two processors and, hence, a communication that must take place during subsequent phases of the computation. Reducing the number of these edges will reduce communication overhead and perhaps increase parallelism. Since we are primarily interested in large sparse graphs, we assume that G is stored as a collection of adjacency lists. A processor is assigned vertex $v \in V$ if it has the list of vertices adjacent to v stored in its local memory. Finally, since we are interested in finding edge separators with the minimum number of edges, we assume that the edges all have cost one.

Our algorithm begins by dividing the p processors into two sets P_1 and P_2 with sizes different by at most one. Sets P_1 and P_2 induce a roughly even division of the vertices. Our initial goal is to reduce the number of edges connecting vertices in P_1 to those in P_2. If $P_1 = \emptyset$ or $P_2 = \emptyset$ then there is nothing to do, so we stop. Otherwise, we perform the following procedure. First, we select one processor in each part, say $\phi_1 \in P_1$ and $\phi_2 \in P_2$, to be the *leader* of that part. If $\phi \in P_i$ then we will say that the leader of ϕ is ϕ_i. The leaders execute the simplified version of the Kernighan-Lin algorithm described below.

Each processor in $P_1 \cup P_2$ computes the D values of all its vertices, and reports these values to its leader. Each leader unmarks all of the vertices in its half of the partition. Next, each leader selects the unmarked vertex with the largest D value. The leaders mark these two vertices and save them on a list along with their gain. The leaders update the D values of the unmarked vertices using Equations 2.1 and 2.2. From these equations, we see that they need the adjacency lists of both selected vertices. The leaders request this information from the processors assigned the selected vertices and, upon receiving it, update the relevant D values. The leaders repeat this process of marking vertices and updating D values until all the vertices assigned to the processors in P_1 or P_2 have been marked.

The leaders now decide what vertices to swap using the same procedure as the Kernighan-Lin algorithm. They inform the processors of their decision, and the processors swap the selected adjacency lists. After swapping vertices, each processor still has the same number of vertices it had originally. The processors repeat this entire algorithm until the number of external edges between P_1 and P_2 cannot be decreased. Then, in parallel, P_1 and P_2 each apply this algorithm recursively. In Figure 2.1, the *ParallelKL* algorithm outlines the entire procedure.

To reduce the number of messages passed between processors in P_1 and P_2, we select vertices a and b to swap that maximize $D_a + D_b$; that is, we ignore a possible edge between a and b. Thus we may choose vertices whose actual gain is less than maximum by at most 2.

1. The processors divide themselves into two groups P_1 and P_2 with sizes different by at most one. If either group is empty, they stop. Otherwise, they select one processor in each group, say $\phi_1 \in P_1$ and $\phi_2 \in P_2$, as the leader of that group.
2. Each processor in $P_1 \cup P_2$ computes the D values of its vertices.
3. Each processor reports its D values to its leader. Each leader unmarks all of the vertices in its half of the partition.
4. Each leader ϕ_i selects the vertex v_i with the largest D value.
5. The leaders request the adjacency lists of v_1 and v_2 from their assigned processors and update the D values of the unmarked vertices.
6. If at least one vertex in each half of the partition is unmarked, the processors repeat from step 4.
7. Using the list of vertex pairs and gains, the leaders decide which vertices to swap, and tell the other processors in their groups.
8. The processors carry out the swapping of vertices.
9. Beginning at step 2, the processors repeat until no further improvement is possible.
10. In parallel, P_1 and P_2 each apply the algorithm recursively, from step 1.

Figure 2.1: The *ParallelKL* algorithm.

As it stands, the algorithm requires a lot of message passing; each processor repeatedly sends all of its adjacency lists to the current leader. Since we want to solve problems too large for a single processor, some of this message passing is unavoidable. However, we can reduce it by allowing a pair of leaders to stop marking vertices when further improvement seem unlikely. In our implementation, leaders stop marking vertices when the sum of all the gains computed so far becomes too negative or when they have encountered too many consecutive nonpositive gains. Since we are primarily interested in sparse graphs, once the sum of all the currently computed gains becomes very negative, it will likely remain negative. In addition, given a good initial assignment of vertices to processors, once a pair of leaders have seen several consecutive nonpositive gains, it is likely that no further improvement is possible using this approach. These modifications should improve the algorithm's running time without significantly affecting the sizes of the resulting edge separators. We will say more about the initial assignment of vertices to processors in Sections 2.6 and 2.7.

As the leaders execute the algorithm, the other processors are mostly idle. Although there is little parallelism at the beginning of this algorithm, more processors become engaged in active work as the algorithm proceeds, i.e., more processors become leaders.

2.3 An Implementation and Analysis

To analyze the computational and communication complexity of the parallel Kernighan-Lin algorithm, we will need some additional notation. Suppose G has n vertices, numbered from 1 to n, and m edges. Let p be the total number of available processors. To simplify the analysis, we assume that p is a power of two, n is a multiple of p, and $p \leq n \leq m$. We also assume that each processor initially has n/p vertices and knows the initial location of every vertex. Then each processor will have exactly n/p vertices throughout the computation. Let q be the maximum storage required by any processor for its vertices at any point during the computation. We call the execution of line 1 of *ParallelKL* a *level-k cut*, where k is the depth of the recursion. The first execution of line 1 is a level-0 cut. If $k \leq \log p$, there are 2^k level-k cuts, all of which can take place in parallel. After making a cut, the relevant processors try to generate a small separator by repeatedly executing lines 2–8 of *ParallelKL*. We refer to a single execution as a level-k *iteration*, where k is the level of the cut. We will assume that the number of level-k iterations after any cut is bounded by some constant. Kernighan and Lin's experiments [31] support this assumption.

We begin by describing an implementation of the algorithm along with an analysis of its computational complexity. For now, we will ignore the message passing. Performing the initial level-0 cut takes $O(p)$ time. Then, in parallel, each processor in each half of the partition computes the D values of all of its assigned vertices in $O(q)$ time and reports them to its leader. Each leader constructs a heap out of the D values it receives. The heap is a balanced binary tree with the maximum D value stored at the root; see Aho, Hopcroft, and Ullman [1] for details of algorithms to construct and maintain a heap. A leader stores its heap as two n-vectors, one containing the D values and the other containing the vertices corresponding to these values. Each leader also maintains an n-vector of pointers from vertices of G to their D values in the heap. The leaders need these pointers to update D values efficiently as vertices are marked. Constructing the heap and the pointers into it takes $O(n)$ time.

A leader removes the vertex with largest D value from the heap (which corresponds to marking it) and remakes the heap, in $O(\log n)$ time. After receiving the adjacency lists of the current pair of marked vertices, a leader modifies the D values of their neighbors and adjusts the heap accordingly, using $O(\log n)$ time per modification. Since there are m edges in G, constructing the complete list of vertex pairs and gains for a level-0 iteration takes $O(m \log n)$ time. Determining the vertices to swap requires $O(n)$ time and (again ignoring message passing time) these vertices can be swapped in $O(m)$ time. Hence, the time for a single level-0 iteration is $O(m \log n)$, since $q \leq m$ and $p \leq n$. Since we have assumed that the number of iterations after any particular cut is bounded by some constant, the time required to find the level-0 edge separator is also $O(m \log n)$. At level $k > 0$, we find the 2^k edge separators in parallel. Thus, the entire algorithm takes

$$O(m \log n \log p)$$

time, ignoring the time for message passing.

To measure the communication complexity, we will count both the total number of messages and the total volume of message traffic, that is, the total number of integers passed in messages. We assume that each processor has an integer label which is known to every other processor. The processors use this labelling to partition processors and select leaders and, hence, require no message passing to perform a cut.

Now consider a single level-k iteration Let P' be the set of processors in one half of the current partition. In line 3 of *ParallelKL*, each processor in P' reports its D values (and corresponding vertex labels) to the leader of P' in a fan-in fashion. The set P' contains $p/2^{k+1}$ processors, so this step requires $p/2^{k+1} - 1$ messages. The total number of integers passed is

$$\sum_{i=1}^{\log p'} \frac{p'}{2^i}(2^i \frac{n}{p}) = \frac{n}{2^{k+1}} \log \frac{p}{2^{k+1}},$$

where $p' = p/2^{k+1}$.

Each cut produces two leaders, both requiring a fan-in report of D values. For $0 \leq k \leq \log p$, there are 2^k level-k cuts, each requiring at most some constant number of iterations. Thus, execution of the entire algorithm produces

$$\sum_{k=1}^{\log p} O\big(2^{k+1}(\frac{p}{2^{k+1}} - 1)\big) = O(p \log p)$$

D value messages containing a total of

$$\sum_{k=1}^{\log p} O\big(2^{k+1}(\frac{n}{2^{k+1}} \log \frac{p}{2^{k+1}})\big) = O(n \log^2 p)$$

integers.

We could have implemented the reporting of D values by simply having each processor send a message containing its D values directly to its leader. In this approach, there would still be $O(p \log p)$ messages, but they would only contain a total of $O(n \log p)$ integers. We use the fan-in method because, on a hypercube, it can be implemented so that only adjacent processors need to communicate. The total number of integers sent over single links is the same —$O(n \log^2 p)$—in the fan-in and direct-to-leader methods; the total number of messages sent over single links is $O(p \log p)$ for fan-in and $O(p \log^2 p)$ for direct-to-leader. Since the machines we are interested in have a significant minimum cost per message, fan-in is more efficient. (Chamberlain and Powell [4,5] examine the fan-in approach to communication in the context of LU and QR factorization.)

To calculate the message traffic required for the remainder of the algorithm, first consider a single level-0 iteration. After constructing heaps of D values, the two leaders request adjacency lists from other processors, communicate vertices of maximum D value to each another, and tell processors what vertices to swap. The other processors send adjacency lists to leaders, and all of the processors carry out the swapping of vertices. All of this communication requires $O(n)$ messages containing a total of $O(m)$ integers. Hence, the entire algorithm requires $O(n \log p)$ messages containing a total of $O(m \log p)$ integers to carry out the communication not involving D values.

There is one subtle point concerning the swapping of vertices. At the start of the algorithm, each processor knows the location of each vertex. Thus, after the initial level-0 cut, the two leaders know what vertices are assigned to each processor. To tell each processor which of its vertices it must send to some other processor, the leaders send a total of $O(n) \leq O(m)$ integers in $O(p) \leq O(n)$

messages. After the swap, the leaders at the next level iteration will not necessarily know the location of each vertex. We can remedy this during the fan-in of D values by including the processor of origin with every D value. This will not change the complexity of fan-in. Therefore the entire parallel separator algorithm requires

$$O(n \log p)$$

messages containing a total of

$$O(\max(n \log^2 p, m \log p))$$

integers.

2.4 A Parallel Ordering Algorithm

Here we use our parallel edge separator algorithm to find nested dissection orderings of A. First, the processors run *ParallelKL* on the graph G of A. We then use each edge separator to define a vertex separator as follows. Suppose some edge separator divides a subset of the processors into two groups, say P_1 and P_2. We can partition the vertices incident on the edge separator into two groups V_1 and V_2, depending on whether they reside in P_1 or P_2. Both V_1 and V_2 are vertex separators for a subgraph of G. We can select the smaller of the two sets, say V_1, as the vertex separator defined by this edge separator. We will call V_1 a *narrow separator*. Let V be the set of all vertices assigned to P_1 and P_2. If the vertices in V_1 are ordered after the vertices in $V - V_1$, no communication across the cut, i.e., between processors in P_1 and those in P_2, is required to compute the columns of L

corresponding to the vertices in $V - V_1$. However, as these columns are computed, they will be sent to processors assigned vertices of V_1. Thus, no matter where the vertices of the narrow separator reside, communication across the cut will take place as the columns that are not in the separator are computed.

Another possibility is to take all of $V_1 \cup V_2$ as the separator of the subgraph, since this guarantees that processors in P_1 and P_2 will not need to communicate until they begin computing columns of L corresponding to vertices in $V_1 \cup V_2$. This is because no fill can occur between a vertex assigned to a processor P_1 and one assigned to a processor in P_2 until the first of these columns is computed. We will refer to such vertex separators as *wide separators*. Since these separators are larger than narrow separators, they will give more fill. However, the number of columns of L that must be communicated across the cut is bounded by $|V_1 \cup V_2|$, the size of the wide separator. For narrow separators, the number of columns crossing the cut is bounded only by $|V|$, the number of columns assigned to processors in P_1 and P_2. Thus, for wide separators, one may hope that the increase in computation time will be more than offset by the decrease in communication time. Section 2.5 contains numerical factorization times using both narrow and wide separators to find orderings.

After defining vertex separators, each processor orders all of its vertices, beginning with those not contained in any separator. In our implementation, the processors use Sparspak's nested dissection routine [22] to order these vertices. Finally, the processors order the vertices contained in the vertex separators after all the other vertices, in such a way that vertices in level-k separators come after those in level-(k+1) separators. The result is a nested dissection ordering whose first $\lceil \log p \rceil$ levels of vertex separators are based on the edge separators from the parallel Kernighan-Lin algorithm. Note that a vertex can belong to more than one separator, provided the separators are at different levels. We consider such a vertex to reside in the separator with the smallest level number.

After all the vertices are numbered, those contained in the vertex separators are redistributed among the processors to balance the computational load during the factorization. In the case of a wide separator, we could *wrap* the vertices in V_1 onto the processors in P_1. That is, if $V_1 = \{v_1, \ldots, v_k\}$ and $P_1 = \{\phi_0, \ldots, \phi_{l-1}\}$, then we would reassign vertex v_i to processor ϕ_j, where $j = (i - 1) \bmod l$. Similarly, we could wrap the vertices in V_2 onto the processors in P_2. This redistribution of vertices would not change the edge separator between P_1 and P_2; however, it could increase the number of edges crossing higher numbered cuts and hence, increase the number of vertices incident on those edges. To avoid this problem, we reassign a vertex only if there is no increase the number of edges crossing other cuts. Whenever a given vertex cannot be reassigned, we note that its assigned processor has an extra vertex and skip this processor the next time around. More formally, if vertex v_i is assigned to processor ϕ_j, we try to reassign it to processor ϕ_k, where ϕ_k is the first processor in the sequence $\phi_{(i-1)\bmod l}, \phi_{i \bmod l}, \phi_{(i+1)\bmod l}, \cdots$ that is assigned no more than $\lfloor i/l \rfloor$ vertices from the set $\{v_0, \ldots, v_{i-1}\}$.

In the case of a narrow separator, V_1 is wrapped onto all the processors in P_1 and P_2. Unlike wide separators, narrow separators are not designed to limit the number of columns communicated across cuts. Thus, we do not need to take the same precautions in wrapping them that we did with wide separators.

Vertex separators correspond to dense submatrices of L, which are more time consuming to compute. Hence, redistributing these vertices evenly among all the processors should give better processor utilization. If we succeed in finding small separators, each processor will end up with roughly the same number of vertices. Since the separator vertices are wrapped, the load will be fairly well balanced. In our experiments, reassigning vertices required very little time and, in many cases, significantly reduced the running time of the remaining phases of the computation. Note that using either narrow or wide separators, at most $p/2^k$ processors need to communicate in order to compute the columns of L corresponding to a level-k vertex separator. On a hypercube, this implies that these columns can be computed in a k-dimensional subcube. Not until the very end of the computation, when the columns of L associated with the level-0 vertex separator are being computed, do all the processors need to communicate.

2.5 Numerical Results

We have implemented the wide and narrow ordering algorithms of Section 2.4 that use the parallel Kernighan-Lin algorithm. We have added this code to the parallel elimination forest, symbolic factorization and triangular system solver codes discussed in Chapters 3, 4, and 5. Together with George, Heath, Liu, and Ng's parallel numeric factorization code [18,20,23], we have a collection of routines that perform all phases of sparse Cholesky factorization in parallel. We have used

Table 2.1: Test problems.

Problem	Equations	Nonzeros	Density (%)
1	346	3226	2.69
2	512	3502	1.34
3	758	5994	1.04
4	878	7448	0.97
5	918	7384	0.88
6	1005	8621	0.85
7	1007	8575	0.85
8	1242	10426	0.68
9	1561	10681	0.44
10	1882	12904	0.38

the Oak Ridge hypercube simulator [10] to generate communication statistics and the Cornell Theory Center's 16 processor Intel hypercube to measure running times.

We have compared three algorithms for ordering the columns of a matrix and assigning them to processors: the narrow and wide algorithms of Section 2.2, and a simple sequential strategy we will call *seq-wrap*. The seq-wrap method orders the matrix sequentially on the host using Sparspak's nested dissection routine and then distributes the columns to all the processors of the hypercube in a wrap fashion. Thus, this method orders the columns to reduce fill and distributes them in a way that should result in good processor utilization, but it ignores the issue of communication. We ran these three algorithms on the 10 finite element problems listed in Table 2.1. The first eight problems represent various physical structures and are described by Everstine [12]; the last two are derived from L-shaped triangular meshes and are described by George and Liu [21].

Table 2.2: Ordering times (seconds).

Problem	Seq-wrap	Narrow	Wide
1	1.88	7.85	7.38
2	1.74	8.34	8.57
3	4.90	13.23	13.28
4	5.00	8.95	8.70
5	5.80	16.10	15.18
6	7.42	23.12	23.30
7	5.88	9.22	9.56
8	7.96	15.67	15.64
9	9.60	12.48	12.94
10	11.78	11.13	10.90

Table 2.3: Message traffic during numeric factorization.

Problem	Seq-wrap	Narrow	Wide
1	3526 (2.13)	2402 (1.93)	1580 (1.74)
2	3166 (2.11)	1087 (1.57)	778 (1.39)
3	6689 (2.11)	2745 (1.84)	2070 (1.63)
4	8836 (2.13)	4558 (1.85)	3321 (1.64)
5	8849 (2.13)	4951 (1.95)	4230 (1.83)
6	10123 (2.13)	5694 (1.97)	4884 (1.83)
7	10438 (2.13)	4843 (1.83)	3318 (1.60)
8	12897 (2.13)	7534 (2.00)	6312 (1.82)
9	15825 (2.13)	7982 (1.97)	6334 (1.76)
10	18917 (2.13)	9537 (1.95)	7368 (1.75)

Table 2.4: Flops during numeric factorization.

Problem	Seq-wrap	Narrow	Wide
1	127294	110344	142954
2	36190	37197	45574
3	93408	112187	166011
4	192834	266103	500606
5	239274	277298	638908
6	458580	470572	897429
7	258793	315552	608565
8	558595	781817	1437048
9	629271	693393	1109808
10	841932	983679	1866030

Table 2.5: Numeric factorization times (seconds).

Problem	Seq-wrap	Narrow	Wide
1	5.06	4.39	3.88
2	3.13	2.22	2.33
3	6.81	5.99	6.53
4	9.91	7.43	10.13
5	10.51	9.18	14.30
6	14.29	17.72	23.46
7	11.25	7.99	12.17
8	17.99	19.51	28.73
9	19.81	16.65	23.40
10	25.45	22.02	34.66

Table 2.2 lists the time required to perform the orderings. Under seq-wrap, we list the time the host uses to order the matrix, ignoring the time required to send the columns to the nodes of the hypercube. Under narrow and wide, we list the times for the parallel Kernighan-Lin algorithm. These include the time to swap columns among processors during the algorithm and the time needed to wrap the columns of the resulting separators. As with seq-wrap, we do not include the time to initially send the columns to the nodes. The initial orderings of problems 3, 5, 6, and 8 were very poor. Due to message-passing delays, narrow and wide both require more time then seq-wrap in all cases except the last. However, as we shall see below, narrow and wide orderings usually succeed in reducing the numeric factorization time. On single-processor machines, numeric factorization is the most time consuming step in solving sparse linear systems. For larger problems, we can expect narrow and wide to require less time than seq-wrap, provided the problem has a reasonable initial ordering. The parallel ordering algorithms also allow us to solve problems that are too large to reside in the memory of any one processor.

After ordering a matrix with one of the algorithms above and symbolically factoring it, we used George, Heath, Liu, and Ng's parallel numeric factorization code [18,20,23] (in an experimental version from February 1987) to compute the Cholesky factor. For each problem, Table 2.3 lists the total number of messages the processors pass during numeric factorizaton. Each message contains the nonzero values of a single column of the Cholesky factor, along with the positions of its nonzeros. Table 2.3 also lists, in parentheses after each total, the average distance travelled by the messages. Since we used a 4-dimensional cube, a message makes at most 4 hops. On the Intel hypercube, messages are broken up into packets of 1024 bytes, and the smallest message is 1024 bytes. Since almost all of the messages passed were smaller than 1024 bytes, we have listed only the total number of messages. Only the wide approach produced messages longer than 1024 bytes and just for problems 8 and 10. In both of these cases, 1% of the messages were longer than 1024 bytes and all were less than 2048 bytes. As expected, the wide approach results in both the lowest total message traffic and lowest average distance travelled per message. For our test problems, narrow requires 32% to 66% fewer messages than seq-wrap, while wide requires 14% to 34% fewer messages than narrow.

Table 2.4 lists the total number of flops the processors perform during the numeric factorization. For most of the problems, the narrow method performs almost as well as Sparspak's nested dissection routine. Due to the large separators, the wide method requires about twice as many flops as the narrow method for the larger problems. For a fixed number of processors, the relative difference between the narrow and wide flop requirements will decrease as the sizes of the problems increase, since the percentage of the columns belonging to wide separators will decrease. Our test problems are all relatively small, and the percentage of columns belonging to wide separators range from 35%, for the largest problem, to 80%, for the smallest.

Table 2.5 lists the factorization times for the three methods. Even though the narrow approach requires somewhat more flops and the wide approach considerably more flops than seq-wrap, both methods frequently require less time. Fac-

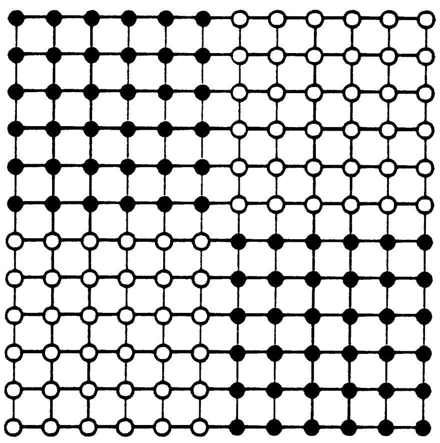


Figure 2.2: A partitioning of a 12 × 12 grid graph.

torization time depends not only on the number of flops, but also on the amount of communication and on how well the load is balanced. Narrow and wide require significantly less communication than seq-wrap, but may not do as well at balancing the load. For example, if the graph is irregular, interprocessor separators at the same level may be of very different sizes and, in fact, this happens with problems 6 and 8. As a result, seq-wrap produces the best factorization times for these problems. Overall, narrow is the best method in terms of factorization time, and hence, it is used to order our test matrices in all the numerical experiments that follow.

2.6 Remarks on the Kernighan-Lin Algorithm

We have seen that using either narrow or wide vertex separators to reorder large sparse symmetric positive definite matrices can decrease the factorization time by lowering the total volume of message traffic. Since both the amount of fill and message traffic depend on the size of these separators, our hope is that we can find small ones for certain types of graphs. In particular, we would like to know if the sequential version of the Kernighan and Lin algorithm presented in Section 2.1 will always find minimum edge separators for a particular class of graphs, regardless of the initial partition.

Let G be an $n \times n$ grid graph where n is even. Suppose it is initially partitioned as in Figure 2.2. The total number of external edges in G is $2n$, twice the minimum. (One minimum edge separator divides the first $n/2$ rows from the others.) The Kernighan-Lin algorithm will not necessarily find a partitioning with the minimum number of external edges. At each step of the algorithm, it must mark a pair of vertices that produces the maximum gain, and, due to the regularity of the graph, it usually has more than one choice. By carefully selecting the vertices to be marked at each step, we can force the algorithm to stop after one iteration without swapping a single pair of vertices.

To see this, think of actually swapping the vertex pairs as they are marked. In Figure 2.2, we can choose the sequence of pairs so that the black vertices in the upper left move to the right, trading places with the white vertices in the upper right. The black vertices in the lower right move to the left, trading places with the white vertices in the lower left. Figure 2.3 shows the partition after swapping the first 30 pairs of vertices. The black vertices in the lower half of Figure 2.3 resemble the letter L. As the swapping progresses from here, the vertical part of this L grows wider, while the horizontal part grows thinner. The upper black vertices behave similarly. The total number of external edges is never less than $2n$. Therefore

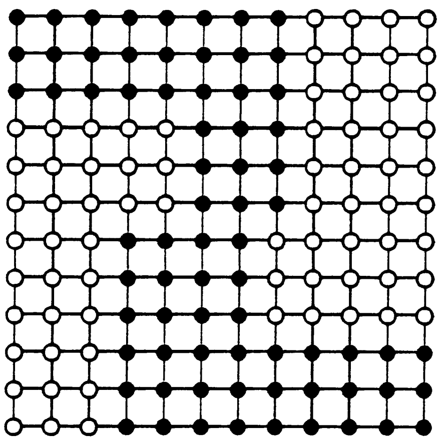


Figure 2.3: The partitioning after swapping the first 30 pairs of vertices.

the sum of gains is never positive, so the algorithm will not actually swap any vertices. Thus, the Kernighan-Lin algorithm does not necessarily find minimum edge separators even for grid graphs. It is important to note, however, that the algorithm can find a minimum edge separator for a grid graph partitioned as in Figure 2.2, if it chooses to mark the vertices in the proper order. We do not know if such an order exists for every initial partition of the graph.

2.7 Discussion

Many graph algorithms are based on finding a small set of vertices or edges whose removal divides the graph into two or more nearly equal parts. Examples include layout of circuits in a model of VLSI [34], efficient sparse Gaussian elimination [25,27,37], and solving various graph problems [38]. Lipton, Rose, and Tarjan [37] have shown that random graphs do not contain good separators. However, many graphs one encounters in practice do have good separators, since most real-world problems have considerable structure. Therefore, finding good separators of graphs is important. Our experience with the Kernighan-Lin algorithm is that it always converges quickly, regardless of the initial partition, but that the quality of this partition affects the size of the resulting edge separator. We are currently examining ways to improve the Kernighan-Lin algorithm. One possiblity is to develop a parallel heuristic for finding good initial partitions, such as a technique for finding highly connected subgraphs of a graph. We could then use this partitioning as input to the algorithm. Another approach is to modify the Kernighan-Lin algorithm so that it uses global knowledge about the graph in breaking ties between the vertices of maximum gain. This could eliminate the problem with the grid graph in Section 2.6.

As we saw in our experiments, the parallel Kernighan-Lin algorithm can produce an assignment of columns to processors and an ordering that results in poor processor utilization during numeric factorization. This happens when interprocesor separators at the same level are of very different sizes. One approach to this problem is to assign weights to the vertices and then partition them in a way that gives each processor roughly the same total weight. If we made the weight of a vertex proportional to the number of edges incident on it, this approach would allow denser parts of the graph to be distributed over more processors and may result in a more uniform distribution of the separators. Kernighan and Lin have suggested a modification of their algorithm that will handle this case for positive integer weights. Namely, if a vertex has weight $k > 1$, replace it with a cluster of k vertices of weight 1, bound together by edges of appropriately high cost.

At the top level of the parallel Kernighan-Lin algorithm, the two leaders perform the entire computation, once the initial D values have been computed. Here, the only advantage of using more than two processors is that more memory is available for storing the graph, so bigger problems can be solved. Of course, as more processors become leaders, more processors become actively involved in the computation. Designing a more parallel algorithm for finding separators is an interesting problem.

For a fixed number of processors, as the problem size increases, the percentage of columns belonging to interprocessor separators will typically decrease. As a result, we expect numeric factorization time differences between the narrow and wide approaches to decrease as the problem size increases. Since each processor in the Cornell Theory Center's Intel hypercube contains four megabytes of memory, we have enough memory to solve problems much larger than our test problems. However, the Intel hypercube limits the maximum length of a message and the total message buffer space. To solve significantly larger problems, we need to explicitly break long messages into several pieces and to consume messages waiting in buffers as quickly as possible. These revisions are currently underway, and when they are complete, we expect to be able to solve problems at least five times larger than our largest test problem.

In general, a parallel algorithm will perform better if it first decomposes the problem it is solving into parts that have high locality and require low communication overhead. Thus, finding good graph partitionings should be a useful first step for a wide variety of parallel problems. For example, in LU factorization with partial pivoting, if we use wide separators to partition the columns of the matrix, then our pivot searches will be confined to single groups of processors. We can also use wide separators in iterative methods, e.g., Jacobi and Gauss-Seidel splitting methods, to reduce the amount of communication.

Simulated annealing is another promising iterative approach to finding small edge separators. At each step, this algorithm creates a new partition from the current one and replaces the current partition with the new one if it is better. If the new partition is worse, the algorithm probabilistically decides whether to accept it as the current partition, with the probability of acceptance decreasing as the number of iterations increase. This feature will hopefully allow the algorithm to move away from a local minimum that is far from a global minimum. Kirkpatrick, Gelatt, and Vecchi [32,33] discuss applications of simulated annealing to optimization problems and provide experimental results for various graph problems, including graph partitioning.

For a class of regular graphs with small optimal edge separators, Bui, Chaudhuri, Leighton, and Sipser [3] develop a polynomial time algorithm that they prove almost always finds optimal edge separators. The algorithm uses the maxflow-mincut algorithm to find edge separators and, in polynomial time, either finds the optimal separator along with a proof of its optimally or halts without output. In experiments with graphs in this class with small vertex degree, the authors found that their algorithm usually locates the optimal edge separator. They also found that the Kernighan-Lin and the simulated annealing algorithms often find poor separators when the graphs in this class have vertex degree equal to 3.

The problem of finding a good ordering along with a good assignment of nonzeros to processors deserves further study. Peters examines aspects of this issue for shared memory multiprocessors [48] and message-passing multiprocessors [47]. Liu and Mirzaian [41,42,44] use tree rotations to create more evenly balanced elimination forests and thereby, better parallel orderings. Fox and Otto [14] suggest two different approaches to automatic load balancing. The first method uses simulated annealing to rebalance the computation when it becomes imbalanced enough to warrant the extra computation. The second method simply partitions the problem into relatively large pieces; it then breaks each large piece into smaller pieces and evenly divides them among all of the processors. Smaller final pieces result in better load balancing, at the expense of more communication.

Chapter 3

Elimination Forests

After reordering A, we must allocate storage for L and then numerically factor A. As we saw in Chapter 1, we must compute the size of $Lrow(k)$ for each column k, in order to efficiently factor A. In this chapter, we define the elimination forest for A and show that it contains the row structure of L. In Chapter 4, we use the elimination forest to allocate storage for L and show how it can be used to compute the sizes of the $Lrow(k)$.

Liu [39] has provided an efficient algorithm for computing elimination forests on a single processor system. This algorithm could be employed here by collecting all the columns of A onto a single processor. There are two obvious drawbacks to this approach. First, since only one of the p processors would be used for this computation, we would not be exploiting all of the available computational power. Second, since the entire matrix would have to fit in the memory available to a single processor, the size of the problems we could solve would be severely limited.

In this chapter, we develop an efficient algorithm for computing the elimination forest for A without moving the columns of A off their assigned processors and without excessive message passing. This work was originally reported in Gilbert and Zmijewski [57,58]. George, Heath, Liu, and Ng [18,19,20,23] have independently suggested the use of elimination forests in sparse factorization and have implemented symbolic and numeric factorization codes that compute the needed elimination forest on a single processor.

3.1 Background

An elimination forest for A is a directed forest $F = (V, E)$ where $V = \{1, \ldots, n\}$ and $\langle i, j \rangle \in E$ if and only if

$$j = \min\{k \mid l_{ki} \neq 0 \text{ and } k > i\}.$$

That is, $\langle i, j \rangle \in E$ if and only if the first nonzero below the diagonal in column i of L occurs in row j. (Figure 3.1 contains an example of a matrix and its elimination forest. In L, an original nonzero is denoted by an X, and a fill is denoted by an ⊠. In this case F happens to be a tree; in fact, each connected component of the graph of A corresponds to a tree in the forest.) If $\langle i, j \rangle \in E$ then a nonzero multiple of column i of L is used to compute column j. Thus, the computation of column j cannot finish before that of column i. Similarly, if there is a path from i to j in F, column i must be computed before column j. We will say that column j *depends* on column i if the computation of column i must be completed before that of column j.

$$L = \begin{bmatrix} \times & & & & & \\ \times & \times & & & & \\ & & \times & & & \\ \times & \otimes & \times & \times & & \\ \times & \times & \times & \otimes & \times & \\ & & & \times & \otimes & \times \end{bmatrix}$$

$F =$

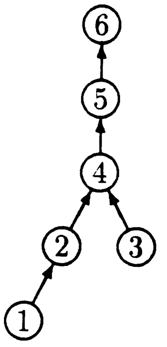


Figure 3.1: A Cholesky factor L and its corresponding elimination forest F.

Parent: $\begin{array}{|c|c|c|c|c|c|}\hline 2 & 4 & 4 & 5 & 6 & 0 \\ \hline\end{array}$

Figure 3.2: A vector represention of the forest F in Figure 3.1.

Schreiber [54] and Liu [39] applied elimination forests to different aspects of Cholesky factorization. They proved several results relating the forest to the nonzero structure of L. We will restate some of these results here and provide somewhat simpler proofs. In particular, we will show that F allows us to compute $Lrow(k)$ for $k = 1, \ldots, n$. Our proofs are in terms of the nonzero structures of A and L, rather than the graphs of these structures, and hence, we must assume that no cancellation occurs while factoring A.

From the definition of F, there is at most one edge leaving any given node and all edges are from lower numbered nodes to higher numbered ones. Thus, we have the following result.

Lemma 3.1 [54] F is a heap ordered forest. $\square$

Since F is a forest, we can represent it as a vector ***Parent*** of pointers. The i^{th} element of ***Parent*** is j if and only if $\langle i, j \rangle \in E$ and is 0 otherwise. We will refer

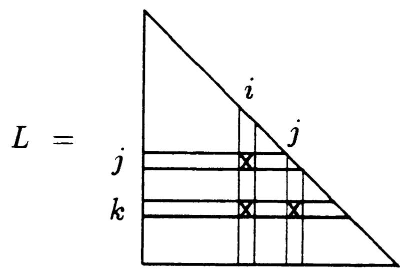


Figure 3.3: Part of the nonzero structure of L.

to j as the *parent* of i in F. Figure 3.2 depicts *Parent* for the elimination forest in Figure 3.1. This simple representation will prove quite useful in Section 3.2, where we give an algorithm for computing F on a multiprocessor system.

Our first two theorems relate the row structure of L to F.

Theorem 3.2 [54] If $i \in Lrow(k)$ then i is a descendant of k in F.

Proof: The proof is a simple backwards induction argument on the columns of L. For column n, there is nothing to prove. Assume the theorem holds for every nonzero in columns $i+1$ through n of L. Suppose $i \in Lrow(k)$, i.e., $l_{ki} \neq 0$ and $k > i$. If the first nonzero below the diagonal in column i of L occurs in row k then $\langle i, k \rangle \in E$ by definition. In this case, i is a descendant of k in F. Now, assume the first nonzero below the diagonal in column i of L is in row j, where $i < j < k$. Thus, $\langle i, j \rangle \in E$. Since $l_{ji} \neq 0$, a nonzero multiple of column i must be added to column j in computing L. Given $l_{ki} \neq 0$, this implies that $l_{kj} \neq 0$ (see Figure 3.3). But $j > i$, so the induction hypothesis tells us that j is a descendant of k in F. Since $\langle i, j \rangle \in E$, we conclude that i is a descendant of k in F. □

From this theorem, it is easy to see that F captures every dependency that exists between the columns of L and only those dependencies. Thus, F can be

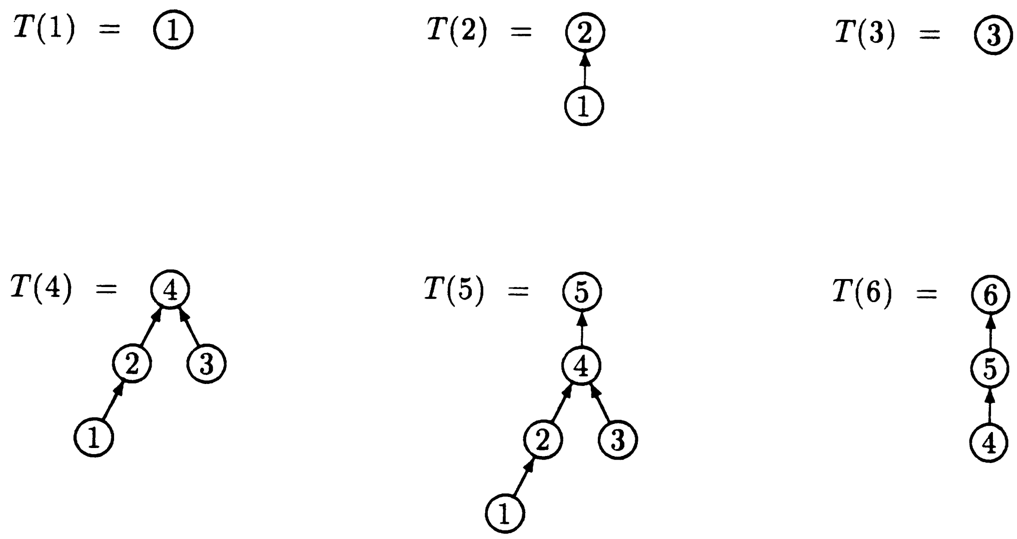


Figure 3.4: Subtrees of the forest F in Figure 3.1.

thought of as the *set* of dependencies that exist between the columns of L. Using an argument similiar to the proof of the theorem, we can establish the following stronger result. Its proof is left to the reader.

Theorem 3.3 [54] If $i \in Lrow(k)$ then there is a path from i to k in F and for all v on this path, $v \in Lrow(k)$.

Corollary 3.4 [54] The set $Lrow(k) \cup \{k\}$ induces a subtree of F rooted at k.

The corollary tells us that the row structure of L is given by certain subtrees of the elimination forest F. Following Liu [39], we let $T(k)$ denote the subtree of F induced by $Lrow(k) \cup \{k\}$. Figure 3.4 depicts these subtrees for the matrix L in Figure 3.1, and the next theorem tells us how to find them using the matrix A.

Theorem 3.5 [39] If $j \in Lrow(k)$ then there exists $d \leq j$ such that $a_{kd} \neq 0$ and d is a descendant of j in F.

Proof: The proof is an induction on the elements of $Lrow(k)$. Let j be the smallest element of $Lrow(k)$. It follows immediately from the column Cholesky algorithm that, since l_{kj} is the first nonzero in row j, it cannot be fill. Thus, $a_{kj} \neq 0$ and the statement of the theorem holds with $d = j$. Now, let j be an arbitrary element of $Lrow(k)$ and assume the theorem holds for all smaller elements. If $a_{kj} \neq 0$ then there is nothing to prove. So, assume $a_{kj} = 0$. Since $l_{kj} \neq 0$, a nonzero multiple of some column of L, say i, is added to column j of A causing a fill at position (k,j) (see Figure 3.3). Thus, l_{ji} and l_{ki} must both be nonzero. Since $l_{ki} \neq 0$ and $i < j$, the induction hypothesis tells us that there is some $d \leq i$ such that $a_{kd} \neq 0$ and d is a descendant of i in F. In addition, since $l_{ji} \neq 0$, it follows from Theorem 3.2 that i is a descendant of j in F. We conclude that d is a descendant of j in F and, hence, the theorem holds for j. □

We can think of $T(k)$ as consisting of the root k along with two other types of nodes, those corresponding to original nonzeros in row k and those corresponding to fill in row k. The theorem tells us that a node representing fill has as a descendant a node representing an original nonzero. Thus, the set of nodes representing original nonzeros of row k contains all of the leaves of $T(k)$ (Figure 3.4). This allows use to easily compute the structure of the k^{th} row of L, given F and the structure of the k^{th} row of A.

We now turn our attention to computing F on a single processor system. Note that F was defined in terms of the matrix L. It would be useful if we could compute F directly from A without explicitly forming L. We could then use F and A to determine the structure of L. As shown in Figure 3.5, the *SeqElimForest*

```
for $j := 1$ to $n$ do
    parent[$j$] := 0;
    for each $a_{ji} \neq 0$ where $i < j$ do
        {Find the root of the tree containing $i$.}
        $r := i$;
        while parent[$r$] $\neq 0$ do
            $r :=$ parent[$r$];
        od
        {Make $j$ the root of this tree.}
        if $r \neq j$ then
          parent[$r$] := $j$;
    do
do
```

Figure 3.5: The *SeqElimForest* algorithm.

algorithm [39] computes F directly from $A = (a_{ij})$, generating a vector *parent* that represents F.

The algorithm scans A by rows. For each nonzero a_{ji} in the lower triangle of A, it makes j the root of the subtree containing i by adding at most one edge to F. Note that the algorithm requires only the nonzero structure of the lower triangle of A. We can represent such a structure as a set of ordered pairs of integers. If $\hat{A}$ represents the nonzero structure of the lower triangle of A, then $\langle j, i \rangle \in \hat{A}$ if and only if $i < j$ and $a_{ji} \neq 0$. On occasion, we will execute *SeqElimForest* using a set of ordered pairs in place of a matrix. We show that *SeqElimForest* correctly computes the elimination forest in Section 3.3.

Depending on the structure of F, the loop for finding the root of a subtree can be the most time consuming part of *SeqElimForest*. Liu [39] has suggested speeding up this search by using *path compression* and *weighted union* [1]. The basic idea is to maintain two forests, the first being the desired elimination forest and the second a *compressed* version of the first that is used for finding roots of subtrees quickly. Using this idea, we can compute F in $O(m\alpha(m,n))$ time, where m is the number of nonzeros in A below the diagonal, n is the number of rows in A, $1 \leq n \leq m$, and $\alpha(m,n)$ is related to the inverse of Ackerman's function. The function α grows extremely slowly. In fact, for any values of m and n that one would encounter in practice, $\alpha(m,n) \leq 4$. We assume *SeqElimForest* is implemented in this manner.

3.2 A Parallel Elimination Forest Algorithm

We saw in Chapter 1 that, in order to develop an efficient sparse column Cholesky factorization routine for a multiprocessor, each processor must know the size of $Lrow(k)$ for each column k of A assigned to it. From Section 3.1, we know that we can easily obtain this information from the elimination forest F and the nonzero structure of A. Here we develop an algorithm for computing F on a multiprocessor where each processor possesses only part of the lower triangle of A. For this algorithm, A need not be partitioned by columns.

The algorithm for computing F has two main parts.

1. Each processor ϕ_i computes an elimination forest F_i based solely on the nonzeros of A assigned to it, using *SeqElimForest*.

2. The processors merge the F_i to obtain the correct elimination forest F for A.

We will examine both of these steps in detail.

The *SeqElimForest* algorithm computes an elimination forest F for A by scanning the rows of A. Each processor ϕ_i computes an elimination forest F_i by executing *SeqElimForest* on the nonzeros of A it has been assigned. Because ϕ_i only knows some of A, F_i is not, typically, the correct elimination forest, F, for the entire matrix. However, F_i is related to F. Recall that a path in an elimination forest corresponds to a dependency between two columns of L. If ϕ_i discovers such a dependency given its incomplete knowledge of A, that dependency will still hold if one considers all of A. Thus, F_i can be thought of as a *subset* of F.

The next step is to merge the various F_i to obtain F. The merging is carried out pairwise and in parallel. For example, suppose we have four processors $\phi_1, \ldots, \phi_4$ containing the elimination forests $F_1, \ldots, F_4$, respectively. Processor ϕ_1 sends F_1 to ϕ_2 and ϕ_2 then merges F_1 and F_2 to obtain a new elimination forest $F_{1,2}$. At the same time, ϕ_3 sends F_3 to ϕ_4 and ϕ_4 then merges F_3 and F_4 to obtain a new elimination forest $F_{3,4}$. After ϕ_2 is finished, it sends $F_{1,2}$ to ϕ_4 and ϕ_4 merges $F_{1,2}$ and $F_{3,4}$ to obtain the correct elimination forest, F, for the entire matrix A. At the end of this merging process, one processor contains F and broadcasts it to the remaining processors. Since sending a forest means sending an n-vector, exactly $2p-2$ messages, each consisting of n integers, are sent during the creation of F and its broadcast.

We now describe the process of merging two elimination forests. Suppose we want to merge $F_1 = (V, E_1)$ and $F_2 = (V, E_2)$. Both forests represent sets of column dependencies. To create a new forest that represents both sets, we run

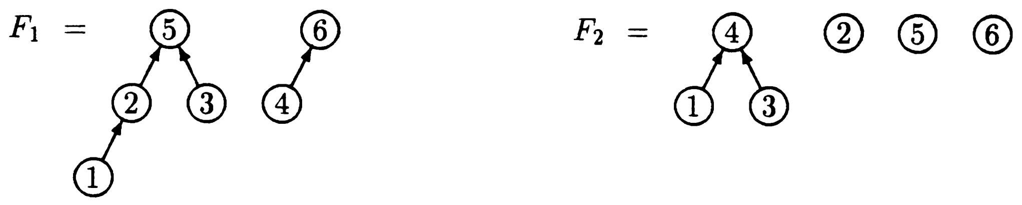


Figure 3.6: The elimination forests computed by ϕ_1 and ϕ_2.

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

Figure 3.7: The matrix representation of F_1 and F_2.

SeqElimForest on U, where $\langle j,i\rangle \in U$ if and only if $\langle i,j\rangle \in E_1 \cup E_2$. Whenever the algorithm examines an edge $\langle j,i\rangle \in U$, it makes j the root of the tree containing i in the forest it is constructing.

Let us consider the example in Figure 3.1, and assume processor ϕ_1 has been given rows 2, 5, and 6, and processor ϕ_2 has been given rows 1, 3, and 4. Processors ϕ_1 and ϕ_2 compute elimination forests $F_1 = (V, E_1)$ and $F_2 = (V, E_2)$ (Figure 3.6). Processor ϕ_1 then sends F_1 to ϕ_2, and ϕ_2 merges F_1 and F_2 by running *SeqElimForest* on U. Since we can view the edges in U as the nonzero positions of a matrix, this is equivalent to running *SeqElimForest* on the matrix in Figure 3.7. The result is the elimination forest for the entire matrix, as shown in Figure 3.1.

Figure 3.8 contains the *Union* algorithm for constructing U from F_1 and F_2 in

```
for $i := 1$ to $n$ do
    nz[$i$] := nil;
od
for $i := 1$ to $n$ do
    if parent1[$i$] $\neq 0$ then
        append $i$ to nzrow[parent1[$i$]];
    if parent2[$i$] $\neq 0$ then
        append $i$ to nzrow[parent2[$i$]];
od
```

Figure 3.8: The *Union* algorithm.

$O(n)$ time. It scans *parent1* and *parent2*, the vector representations of F_1 and F_2, and creates an n-vector of pointers *nzrow*. The pointer *nzrow*$[j]$ points to a linked list of nondecreasing integers, where i is an element of the list if and only if $\langle j, i \rangle \in U$.

We first analyze the parallel running time needed to compute F, ignoring the time required to pass messages. Processor ϕ_i computes F_i in

$$O(q_i \alpha(q_i, n) + n)$$

time, where q_i is the total number of nonzeros of A assigned to ϕ_i. Once the processors have computed all the F_i, they can begin merging them. Since a processor can merge two F_i in $O(n\alpha(n,n))$ time, and the F_i are merged pairwise and in parallel, we can construct F in

$$O(q\alpha(q,n) + n\alpha(n,n)\log p)$$

time, where q equals the largest q_i.

Message-passing to compute and broadcast F consists of $2 \log p$ rounds of n-word messages. On a hypercube architecture, the messages are all between adjacent processors, so message-passing time is $O(n \log p)$.

3.3 A Proof of Correctness

In this section, we prove that our algorithm for computing the elimination forest on a multiprocessor is correct. We assume that the nonzeros of $A = (a_{ij})$ have been partitioned into p sets, $N_0, \ldots, N_{p-1}$, and assigned to processors $\phi_0, \ldots, \phi_{p-1}$, respectively. The N_k need not correspond to columns of A. Recall that the algorithm for generating the elimination forest consists of two parts. First, each ϕ_k produces a forest F_k by running *SeqElimForest* on N_k. Then, the processors merge the F_k into a single forest F, again using *SeqElimForest*. Our object is to show that F is indeed the correct elimination forest for A as defined in Section 3.1. Toward this end, we will require some additional notation. Let $G = (V, E)$ be the graph of A and $G^* = (V, E^*)$ be the corresponding filled graph, where $V = \{1, \ldots, n\}$ for notational simplicity. Let F^A be the correct elimination forest for A. We will show that $F = F^A$.

Lemma 3.6 Every forest created by *SeqElimForest* is heap ordered.

Proof: Suppose we construct a forest H by running *SeqElimForest* on a set B of ordered pairs. At stage j of the algorithm, we consider those $\langle j, i \rangle \in B$, where $i < j$. At the beginning of this stage, j is an isolated vertex in H. During it, we make j the root of various trees in H consisting of nodes numbered less than j. Thus, each step of the algorithm maintains heap order. □

As we saw in Section 3.2, each F_k can be viewed as a set of dependencies between the columns of L. We would like each F_k to be a subset of F. That is, we don't want any information contained in the F_k to be lost during the merging process. The next lemma tells us that this does not happen.

Lemma 3.7 For all k, if i is a descendant of j in F_k then i is a descendant of j in F.

Proof: Let $F_{1,2}$ be the result of merging $F_1 = (V, E_1)$ and $F_2 = (V, E_2)$. We will show that all the descendant relationships in F_1 and F_2 are also in $F_{1,2}$. Suppose i is a descendant of j in F_1. Let $T : i = k_0, \ldots, k_l = j$ be the path from i to j in F_1. The proof will be by induction on l, the length of T. If $l = 1$ then $\langle i, j \rangle$ is an edge of F_1, and hence, $\langle j, i \rangle \in U$. When $\langle j, i \rangle$ is examined during the merge of F_1 and F_2, i is made a descendant of j in the partially constructed $F_{1,2}$. Since edges are never removed during the construction of $F_{1,2}$, i is a descendant of j in $F_{1,2}$. Now assume $l > 1$. By the induction hypothesis, i is a descendant of k_{l-1} and k_{l-1} is a descendant of k_l in $F_{1,2}$. Thus, i is a descendant of j in $F_{1,2}$. Since F_2 can replace F_1 in this argument, we conclude that merging F_1 and F_2 preserves the dependencies found in both forests. Since F is built up by a series of such merges, the lemma holds. □

Now that we have seen that F embodies the information in each of the F_k, we will show that F and G^* are closely related. The following two lemmas and two corollaries prove that an edge in G^* corresponds to a descendant relationship in F and an edge in F corresponds to an edge in G^*.

Lemma 3.8 If $(i,j) \in E^*$, where $i < j$, then i is a descendant of j in F.

Proof: Since $(i,j) \in E^*$, Lemma 1.1 tells us that there exists a path in G from i to j through lower numbered vertices. Let $T : i = k_0, k_1, \ldots, k_l = j$ be the shortest such path. The proof is by induction on l, the length of T. If $l = 1$ then $(i,j) \in E$. Therefore element a_{ji} of A is nonzero and is assigned to some processor ϕ_r. During the construction of F_r, ϕ_r examined a_{ji} and made j the root of the tree containing i. Thus, i is a descendant of j in F_r. From Lemma 3.7, i is a descendant of j in F.

Now, suppose $l > 1$. Let k_r be the largest vertex on T less than i. Since $i = k_0, k_1, \ldots, k_r$ is a path in G from i to k_r through lower numbered vertices, $(k_r, i) \in E^*$. Furthermore, this path is of length less than l, so k_r is a descendant of i in F by the induction hypothesis. A similar argument shows that k_r is also a descendant of j in F. Since i and j have a common descendant and F is a forest, either i is a descendant of j or j is a descendant of i. But $i < j$, so Lemma 3.6 allows us to conclude that i is a descendant of j in F. □

Lemma 3.9 Let Z be a set of directed edges, where $\langle i,j \rangle \in Z$ implies $(i,j) \in E^*$ and $j > i$. If *SeqElimForest* is run on Z then for every edge $\langle i,j \rangle$ in the resulting forest, edge (i,j) is in E^*.

Proof: Suppose some processor ϕ constructs forest H by running *SeqElimForest* on Z. The proof is by induction on the construction of H. Suppose that at some step of the construction ϕ examines $\langle j,i\rangle \in Z$. At this point, ϕ makes j the root of the tree containing i in the partially constructed forest. This is accomplished as follows. Let $i = i_0, i_1, \ldots, i_t$ be the path from i to the root i_t of its subtree. If $i_t \neq j$ then $\langle i_t, j\rangle$ is added to the forest. Otherwise, no edge is added.

Assume $i_t \neq j$. We must show that $(i_t, j) \in E^*$. That is, we must show that the undirected form of the edge added at this stage of the construction is in the filled graph G^*. If $t = 0$ then we have added the edge $\langle i,j\rangle$ to the forest. Since $\langle j,i\rangle \in Z$, we have $(i,j) \in E^*$ by definition of Z.

Now, assume $t > 0$. Since heap order is maintained at all times during the construction, we know that $i_0 < i_1 < \cdots < i_t$. Furthermore, the edges $\langle i_0,i_1\rangle, \langle i_1,i_2\rangle, \ldots, \langle i_{t-1},i_t\rangle$ were all added to the forest earlier in the construction and so, by the induction hypothesis, the undirected forms of all these edges are in E^*. Thus, there exist paths in G from i_k to i_{k+1} through lower numbered vertices for $k = 0, 1, \ldots, t-1$. Patching all of these paths together, we get a path in G from $i_0 = i$ to i_t through vertices numbered less than i_t. Since $(i,j) \in E^*$ and $i < i_t < j$, there is a path in G from j to i_t through lower numbered vertices. We conclude that $(i_t, j) \in E^*$. □

Corollary 3.10 For all k, if $\langle i,j\rangle$ is an edge of F_k then $(i,j) \in E^*$.

Proof: Processor ϕ_k constructs F_k by running *SeqElimForest* on N_k. Since $N_k \subseteq E \subseteq E^*$, the corollary holds. □

Corollary 3.11 If $\langle i,j\rangle$ is an edge of F then $(i,j) \in E^*$.

Proof: Consider merging F_1 and F_2 on ϕ_2, and let $F_{1,2}$ be the resulting forest. Processor ϕ_2 merges F_1 and F_2 by running *SeqElimForest* on the edges in F_1 and F_2. The undirected forms of all of these edges are in E^*, and hence, the same is true of the edges in $F_{1,2}$. Since F is built up by a series of such merges, the corollary is true. □

Our last lemma gives us a condition for deciding if two arbitrary forests are equal. We will use it and the relationship between F and G^* to prove $F = F^A$.

Lemma 3.12 Let $H_1 = (V, E_1)$ and $H_2 = (V, E_2)$ be two directed forests on the same set of vertices. If for all $\langle i, j\rangle \in E_1$, i is a descendant of j in H_2, and for all $\langle i, j\rangle \in E_2$, i is a descendant of j in H_1, then $H_1 = H_2$.

Proof: Under the assumptions of the lemma, an easy induction argument shows that i is a descendant of j in H_1 if and only if i is a descendant of j in H_2. Now assume $\langle i, j\rangle \in E_1$. By hypothesis, i is a descendant of j in H_2. Suppose $\langle i, j\rangle \notin E_2$ and let k be some vertex on the path from i to j in H_2, excluding the end points. Since i is a descendant of k in H_2, we know that i is a descendant of k in H_1. Likewise, we know that k is a descendant of j in H_1. Thus, there is a path in H_1 from i to j through k. Noting that $k \neq i, j$ and $\langle i, j\rangle \in E_1$, we conclude that there are two distinct paths from i to j in H_1. But this is impossible since H_1 is a forest. This contradiction proves that if $\langle i, j\rangle \in E_1$ then $\langle i, j\rangle \in E_2$. Since E_1 and E_2 can be interchanged in the above argument, we have shown that $E_1 = E_2$. □

Finally, we come to the main result of this section. Using the above lemmas, we show that our parallel elimination forest algorithm is correct.

Theorem 3.13 $F = F^A$.

Proof: If $\langle i,j \rangle$ is an edge of F^A then (i,j) is also an edge of G^* by the way F^A is defined. Thus, from Lemma 3.8, i is a descendant of j in F.

If $\langle i,j \rangle$ is an edge of F then Corollary 3.11 tells us that $\langle i,j \rangle$ is in G^*. Theorem 3.2 implies that i is a descendant of j in F^A. Since each edge in one forest represents a descendant relationship in the other, Lemma 3.12 proves the theorem. $\Box$

Corollary 3.14 *SeqElimForest* for computing F^A on a single processor is correct.

Proof: For a single processor system, no forest merges are required. The lone processor simply runs *SeqElimForest* using all the nonzeros of A. $\Box$

3.4 Numerical Results

We implemented the parallel elimination forest algorithm of Section 3.2, assuming that the nonzero structure of A is partitioned among the processors by rows. Note that since A is symmetric, the nonzero structures of row k and column k are the same. We obtained an implementation of Liu's sequential elimination forest from George, Heath, Liu, and Ng [19]. The implementation of Liu's algorithm uses path compression but not weighted union. After ordering our test matrices and assigning their columns to processors using the narrow ordering algorithm of Chapter 2, we compared the parallel and sequential elimination forest algorithms. The sequential elimination forest algorithm runs on the host; the parallel algorithm runs on the nodes of the hypercube.

Table 3.1: Elimination forest times (seconds).

Problem	Elimination Forest	
	Sequential	Parallel
1	0.281	0.560
2	0.281	0.720
3	0.500	1.035
4	0.621	1.230
5	0.602	1.290
6	0.719	1.410
7	0.719	1.390
8	0.879	1.790
9	0.922	2.120
10	1.121	2.535

Table 3.1 provides running times for the computation of the elimination forests. Although the parallel elimination forest algorithm takes about twice as long as the sequential algorithm for all of our test problems, the running times of both algorithms are small when compared to the other phases of the computation. For problems at least twice as large or dense, we expect the parallel algorithm to take less time than the sequential one. The main advantage of the parallel elimination forest algorithm is that it can solve problems that are too large to reside on a single processor.

3.5 Discussion

The parallel elimination forest algorithm carries out its computation without moving the columns of A off their assigned processors and without requiring excessive message passing. This allows us to solve problems that are too large to reside on a single processor. For problems small enough to reside on the single host processor, the sequential algorithm is faster than the parallel one. This is mainly because the pairwise merge uses fewer processors as it proceeds, so utilization is low. It is not clear how to overcome this. Gilbert and Hafsteinsson [24] give a shared-memory symbolic factorization algorithm with good processor utilization throughout, but a message-passing implementation of that algorithm would waste huge amounts of time passing small messages.

Chapter 4

Symbolic Factorization

After we have computed the elimination forest F and distributed it to all of the processors, we can begin the computation of L. First, we must allocate storage for the columns of L. That is, if a processor contains column i of A, it must allocate storage for column i of L. As noted earlier, F contains the row structure of L; however, F cannot readily be used to compute the column structure of L. Thus, although every processor contains a copy of F, the processors must perform a symbolic factorization of A to determine the column structure of L. In this chapter, we discuss two different symbolic factorization algorithms. Gilbert and Zmijewski [58] first reported on the ideas in this chapter.

4.1 Column Symbolic Factorization

To determine the column structure of L, we could simulate the numeric factorization on the nonzero structure of A. However, as noted by George, Heath,

```
processor $\theta_k$
    {Col $k$ is the data structure for column $k$ of $L$.
     Nzcol $k$ is the data structure for the nonzero structure of column $k$ of $L$.}
    {Compute the nonzero structure of column $k$ of $L$.}
    initialize nzcol $k$ to contain the nonzero positions in column $k$ of $A$;
    for $i := 1$ to nchild[$k$] do
        recv nzcol $j$;                {$j$ defined by sender}
        merge nzcol $j$ into nzcol $k$;
    od
    {Nzcol $k$ now contains the nonzero structure of column $k$ of $L$ and must
     be sent to the appropriate processor.}
    send (nzcol $k - \{k\}$) to processor $\theta_{\text{parent}[k]}$;
    {Allocate storage for column $k$.}
    set up the data structure for col $k$ using nzcol $k$;
```

Figure 4.1: The *ColSymFact* algorithm.

Liu, and Ng [19], a faster algorithm employs the following observation. If $\langle i, k\rangle$ is an edge in F then a nonzero multiple of the i^{th} column of L is needed in the computation of column k. Thus, if we ignore the i^{th} element of column i, the sparsity pattern of column i is contained in the sparsity pattern of column k. In general, if we disregard elements of column i in rows less than k, the sparsity pattern of column k contains the sparsity pattern of column i whenever i is a descendant of k in F. The columns of L used to compute column k are those in $Lrow(k)$ and, as we saw in Section 3.1, they form a tree in F. It follows that the nonzero structure of column k of L can be found by merging the nonzero structures of each column of L that is a child of k in F and the nonzero structure of the k^{th} column of A. For a single processor system, this observation leads to a way to carry out the symbolic factorization in time proportional to the number of nonzeros in L. Since numeric factorization requires time proportional to that needed to multiply L and L^T, this approach can save considerable time. It is used in Sparspak [22].

In order to implement a parallel column-oriented symbolic factorization algorithm, each processor must determine the number of children node k of F has, for each of its columns k. This is accomplished in $O(n)$ time by scanning the vector representation of F. We assume that the results are saved in a vector *nchild*, where *nchild*$[i]$ is the number of children of node i in F, and that F is stored as a vector *parent*. Figure 4.1 contains the code for the column-oriented symbolic factorization algorithm *ColSymFact*. For ease of presentation, we again assume one column per processor. Ignoring the time required to pass messages, this approach requires at most $O(|L|)$ time. In the worst case, *ColSymFact* sends $O(n)$ messages containing a total of $O(|L|)$ integers. For a narrow nested dissection ordering, the $O(n)$ messages contain a total of $O(z_r)$ integers, where z_r is the number of nonzeros in the rows of L corresponding to interprocessor separators. For a wide nested dissection ordering, *ColSymFact* sends only $O(n_s)$ messages containing a total of $O(z_c)$ integers, where n_s is the number of columns of L belonging to the interprocessor separators and z_c is the total number of nonzeros in these columns. Depending on the assignment of columns to processors, the height of the trees in the elimination forest, and the communication speed, the processors could spend a lot of time waiting for messages, and hence, *ColSymFact* could have low processor utilization. In general, *ColSymFact* will perform better if the elimination forest contains trees that are short and wide, rather than long and narrow.

4.2 Row Symbolic Factorization

A different approach to symbolic factorization is to use F to compute the row structure of L and then transpose the result. We will call this the *RowSymFact* algorithm. Note that since A is symmetric, typical Cholesky factorization routines for single processor systems (e.g., Sparspak [22]) would save only the nonzero structure of the lower triangle of A along with the corresponding numeric values. However, for this approach, we assume that for each column of A, its assigned processor saves the sparsity pattern of the entire column. This allows us to determine $Lrow(k)$ from F. To compute $Lrow(k)$, we need the sparsity pattern of row k of A (Theorem 3.5). More precisely, we need to know where the nonzeros occur in columns 1 through $k-1$ of row k of A. Since A is symmetric, the sparsity pattern of the k^{th} column of A is also the sparsity pattern of the k^{th} row. Thus, the processor assigned column k will be able to compute $Lrow(k)$ from F. Since L will, in general, be much more dense than A, the additional storage required should not be significant. After the $Lrow(k)$ are computed, this overhead storage can be reclaimed and used for the nonzeros of L. Saving sparsity patterns of entire columns also simplifies the implementation of the parallel elimination forest algorithm, since we can scan rows by scanning columns.

In our implementation of *RowSymFact*, we assume that the matrix has been ordered using *ParallelKL*. Recall that this algorithm finds *interprocessor* separators that partition the columns in groups that reside on single processors. These separators are then ordered last. In *RowSymFact*, each processor computes $Lrow(k)$

```
for $k$ := 1 to $n$ do
    visited[$k$] := 0;
    Lrow[$k$] := empty list;
    Lcolumn[$k$] := empty list;
od

for each column $k$ in an interprocessor separator on this processor do
    {Compute $Lrow(k)$.}
    for each $a_{ik} \neq 0$ where $i < k$ do
        {Examine the path from $i$ to $k$ in $F$.}
        $t := i$;
        while $t \neq k$ and visited[$t$] $\neq k$ do
            append $t$ to Lrow[$k$];
            visited[$t$] := $k$;
            $t$ := parent[$t$];
        od
    od
od

for each column $k$ in an interprocessor separator on this processor do
    {Transpose $Lrow(k)$.}
    for each $i$ in Lrow[$k$] do
        append $k$ to Lcolumn[$i$];
    od
od

send the partial column structures to the appropriate processors;
compute the remaining row structures and transpose them;
receive the partial column structures and allocate storage for $L$;
```

Figure 4.2: The *RowSymFact* algorithm.

in decreasing order of k for each of its assigned columns k. Once a processor has computed $Lrow(k)$ for each of its columns k belonging to an interprocessor separator, it transposes these row structures and sends out the resulting (partial) column structures to the appropriate processors. Each processor then computes and transposes $Lrow(k)$ for its remaining columns k. Each of these additional column structures resides on the processor assigned that column. Finally, the processors receive the column structures they have been sent and allocate storage for the columns of L. Figure 4.2 contains the ***RowSymFact*** algorithm. Ignoring the time required to pass messages, ***RowSymFact*** requires at most $O(|L|)$ time. The algorithm sends $O(p^2)$ messages containing at most $O(|L|)$ integers, and unlike *ColSymFact*, always generates the same number of messages. As with *ColSymFact*, ***RowSymFact*** sends messages containing a total of $O(z_r)$ integers for narrow orderings and $O(z_c)$ integers for wide orderings. Provided each processor has enough columns not belonging to any interprocessor separator, no processor will have to wait for messages and ***RowSymFact*** will have good utilization.

4.3 Numerical Results

We have implemented ***RowSymFact*** and obtained an implementation of *ColSymFact* from George, Heath, Liu, and Ng [19]. We ran both algorithms on our test problems and have provided the results in Table 4.1. For these problems, ***RowSymFact*** was significantly slower than *ColSymFact*. Relative to the number of processors, the test problems are all small and, as a result, a large percentage

Table 4.1: Symbolic factorization times (seconds).

Problem	Symbolic Factorization	
	Column	Row
1	0.48	1.17
2	0.27	0.80
3	0.57	1.66
4	0.68	1.97
5	1.00	1.92
6	1.64	4.32
7	0.65	2.29
8	2.05	4.71
9	1.31	3.95
10	1.63	5.06

of the columns belong to interprocessor separators. In addition, these columns are typically denser than those not belonging to an interprocessor separator. During the *RowSymFact* algorithm, the processors do not have enough work left once they have processed the columns belonging to the interprocessor separators. Hence, they become idle before their needed partial column structures arrive. Since *ColSymFact* communicates in a more uniform fashion than *RowSymFact*, it does not encounter this problem. For problems at least twice as large or on machines with faster communication, we expect *RowSymFact* to be competitive with *ColSymFact*.

Recall from Chapter 1 that we need to know the sizes of the $Lrow(k)$ to carry out the numeric factorization. We compute these sizes using code very similar to that in *RowSymFact* for computing $Lrow(k)$. We perform this computation prior to the symbolic factorization, and since we have saved the entire sparsity pattern of each column, no message passing is required. In code from George, Heath, Liu, and Ng, the processors carry out this computation after the symbolic factorization. Each processor scans its columns of L, determining the total number of nonzeros in each row, and then broadcasts this information to the other processors. For our test problems, both algorithms required a relatively insignificant amount of time. With a narrow nested dissection ordering, running times for our approach ranged from 0.06 to 0.28 seconds, and running times for the George, Heath, Liu, and Ng approach ranged from 0.22 to 1.16 seconds.

Chapter 5

Triangular System Solving

After computing the Cholesky factor L of A, we can find the solution to the system of linear equations $Ax = b$ by solving the triangular systems $Ly = b$ and $L^Tx = y$. On a single processor machine, solving triangular systems is trivial. However, as we shall see in this chapter, developing a good triangular solver for a message-passing multiprocessor requires some care. Even given a fixed assignment of columns to processors, we have a lot of flexibility in choosing the frequency and volume of message traffic during the solve. Since the solve requires relatively little computation ($O(|L|)$ flops), we must be careful to ensure that the processors spend most of their time doing useful work, rather than passing or waiting for messages. As with all our algorithms, we want the computation to mask the communication. A number of different algorithms for solving dense triangular systems have been proposed [4,30,35,36,49]. If we order a sparse matrix using a nested dissection algorithm, e.g., the parallel ordering algorithm in Chapter 2, then the separators will correspond to dense blocks in L. For these blocks, we can use dense triangular solve techniques. Li and Coleman [35,36] have devised one of the most promising dense triangular solve algorithms. In this chapter, we review their algorithm and then show how to incorporate it in a sparse triangular solver.

5.1 Dense Triangular System Solving

In this section, we will concentrate on solving $Ly = b$ on a message-passing multiprocessor, where L is dense. We assume that the multiprocessor contains an embedded ring and the columns of L are wrapped around this ring. We also assume that each processor maintains a list *mycols* of its assigned columns in increasing order. At the start of the triangular solve, processor θ_1 contains the values of the right-hand side b; all other processors initialize b to be a vector of zeros. Figure 5.1 contains a sequential algorithm *SeqSolve* for computing the solution to $Ly = b$ by columns. The algorithm overwrites b with y and requires $O(n^2)$ time. After θ_k computes y_k and uses it to update b, it sends b to θ_{k+1}. Thus, only one processor is active at any time. The entire solution vector ends up on θ_n.

To introduce some parallelism into the *SeqSolve* algorithm, we can have the processors issue sends before they complete updating b. Since the columns of L are wrapped onto the processors, after computing y_k, processor θ_k is next responsible for computing y_{k+p}. Thus, after computing y_k, processor θ_k need only modify $b_{k+1}, \ldots, b_{k+p-1}$ before issuing the send. It can modify the remaining components of b after the send. Li and Coleman [36] used this idea to develop a parallel

```
for each $j \in$ mycols do
    if $j > 1$ then
      recv $b$;
    fi
    $b_j := b_j / l_{jj}$;
    for $i := j + 1$ to $n$ do
        $b_i := b_i - b_j * l_{ij}$;
    od
    send $b$ to processor $\theta_{j+1}$;
od
```

Figure 5.1: The *SeqSolve* algorithm.

```
for each $j \in$ mycols do
    if $j > 1$ then
      recv $x_1, \ldots, x_{p-1}$;
      for $i := 1$ to $p - 1$ do
          $b_{j+i-1} := b_{j+i-1} + x_i$;
      od
    fi
    $b_j := b_j / l_{jj}$;
    $m := \min(j + p - 1, n)$;
    for $i := j + 1$ to $m$ do
        $b_i := b_i - b_j * l_{ij}$;
    od
    send $b_{j+1}, \ldots, b_m$ to processor $\theta_{j+1}$;
    for $i := m + 1$ to $n$ do
        $b_i := b_i - b_j * l_{ij}$;
    od
od
```

Figure 5.2: The *ParallelSolve* algorithm.

version of the *SeqSolve* algorithm. Figure 5.2 contains their algorithm, which we call *ParallelSolve*. The algorithm uses a vector x of length $p-1$ to receive contributions to b. When *ParallelSolve* terminates, component y_k of the solution will reside on processor θ_k. In practice, *ParallelSolve* works well when $n \gg p$. For a 16 processor Intel hypercube and $n = 1000$, Li and Coleman report that *SeqSolve* requires 1122.6 seconds, but *ParallelSolve* needs only 41.7 seconds.

Li and Coleman analyzed the running time of *ParallelSolve* in terms of flops, assuming that the columns of L are wrapped around an embedded ring of processors. To account for message passing times, they defined t to be the maximum number of flops that can be executed during the time required to send p or less double precision words between two adjacent processors. If $n > p(t+p)$ then the running time of the algorithm is proportional to

$$\frac{1}{2}\left\{\frac{n^2}{p} + n + p(t+p)^2 - pt - p^2 + p\right\} - t.$$

Asymptotically, this is the best possible, namely, $O(n^2/p)$. Except at the very beginning and end of the computation, the processors spend no time waiting for messages.

For $n \leq p(t+p)$, *ParallelSolve* is communication rather than computation bound, with the processors idly waiting for each message they receive. The algorithm requires $O(p(t+p))$ time and is essentially sequential when n is only moderately larger than p. Li and Coleman [35] devised an improvement to their algorithm for this case. They partitioned the $p-1$ length vector that they pass around the ring into q roughly equal sized segments, where $1 \leq q \leq p$. After a processor updates one of these q segments, it sends the segment directly to the processor that needs it. One of the q segments travels around the ring as before, while the rest travel across the ring. Thus, each processor receives q updates to b before computing another component of the solution vector. For $n \leq p(t+p)$, this algorithm is faster than the original one. Once θ_k computes y_k, it updates at most $\lceil p/q \rceil$ components of b before issuing a send to θ_{k+1}, allowing θ_{k+1} to compute y_{k+1} sooner.

To simplify the analysis of the improved algorithm, Li and Coleman assumed that q divides p and that a cross-ring message of size $\overline{p} = p/q$ takes time at most $t \log p$ flops. Here, t is the maximum number of flops that can be executed during the time required to send $\overline{p}$ or less double precision words between two adjacent processors. Li and Coleman also assumed that the processors forward messages immediately at zero cost. Under these assumptions, the running time of this new algorithm is proportional to

$$(t+\overline{p})n - \frac{1}{2}\overline{p}(\overline{p}-1) - t,$$

when $n \leq p(t+p)$. Since $p(t+p)$ can be rather large, this improvement to the algorithm is significant. Li and Coleman have estimated t at 40 for an Intel hypercube. Thus, for a 16 processor Intel hypercube, $p(t+p)$ is 896 and, for 64 processors, it is 6656.

The choice of q is important. For $p = 128$ and $n = 1000$, Li and Coleman report solving times that vary between 9.6 seconds and 46.3 seconds as q ranges over the powers of 2 from 1 to 128. For this particular problem, the plot of q

versus running time resembles an upright parabola with smallest time at $q = 8$. In general, we want to select a value for q just large enough to ensure that the processors do not idly wait for messages during the computation. Selecting a larger value will only increase the message passing overhead, and thereby increase the total running time. Under the same assumptions used to determine the running time, Li and Coleman determined that the desired value of q is given by $q^*(n,p,t)$, where $q^*(n,p,t)$ is

$$\max\left\{1, \min\left\{\frac{p}{2}, \frac{2p}{(3-2t)+\sqrt{(3-2t)^2+8(t\log p+1)}}, \frac{p(p+1)}{n-p(t-1)}\right\}\right\},$$

for $n \le p(t+p)$, and is 1, otherwise. Thus, for $n > p(t+p)$, we have the original *ParallelSolve* algorithm. For fixed p and t, $q^*(n,p,t)$ is constant for all n outside of a certain range. In particular, if $p = 16$ and $t = 40$, then $q^*(n,p,t) = 4$ for all $n \le 691$ and $q^*(n,p,t) = 1$ for all $n > 876$. Experimently, Li and Coleman found that $q^*(n,p,t)$ overestimates the best choice for q by roughly a factor of 2. This is not surprising, since the processors require time to forward messages and do not necessarily do so immediately. All of this suggests that, rather than recompute $q^*(n,p,t)$ for each n, it may be reasonable in practice to set q equal to $q^*(n,p,t)/2$ for all $n \le p(t+p)$, and equal to 1, otherwise.

To solve for $L^T x = y$, Li and Coleman suggested an algorithm very similar to *ParallelSolve*. Here, we must solve a triangular system by rows. Instead of passing around a $p-1$ length vector of updates to y, we pass around $p-1$ components of the solution vector, x. When a processor ϕ receives such a message, it immediately computes the next component of x and sends it and the $p-2$ earlier components to the next processor. Processor ϕ then updates the right hand side y by scanning its rows and subtracting off multiples of the p most recently computed components of x. As with *ParallelSolve*, we can break the $p-1$ length messages into q segments when n is small relative to p.

5.2 Sparse Triangular System Solving

We can use dense triangular solve techniques in solving sparse triangular systems. In the rest of this chapter, we assume that A is sparse and has been ordered by the nested dissection algorithm in Chapter 2. We further assume that if a processor is assigned column k of A it also has b_k, the k^{th} component of b. Otherwise, it initializes b_k to zero. If L is the Cholesky factor of A, we can solve $Ly = b$ as follows. First, each processor computes y_k for each of its assigned columns k that are not part of any interprocessor separator. These components of y can be computed without any communication. After each component is computed, it is used to update b. Next, the processors compute the remaining components of y using a variant of the dense algorithms described in the last section. These components correspond to the interprocessor separators, and each such separator is a dense block in L.

To use the improved version of the *ParallelSolve* algorithm for sparse matrices, we have made the following two modifications. First, note that the blocks of L corresponding to the interprocessor separators need not consist entirely of nonzeros. Since the computation of y_k depends only on those y_i where $i < k$ and

$l_{ki} \neq 0$, the *ParallelSolve* algorithm could generate unneeded messages. We can avoid this as follows. After updating a segment of b, a processor sends the segment to processor θ_j, where j is the first nonzero in the segment. If no such j exists, no message is sent. Analogous to numeric factorization, a processor must know when it has received enough messages to compute the next component of y. To compute y_k, the processors must modify b_k exactly $|Lrow(k)|$ times. Since any processor can modify b_k, the messages we send contain both components of b and the number of times each has been modified. Each processsor computed the sizes of the appropriate $Lrow(k)$ for the numeric factorization phase, and so, each can determine how long to wait before computing the next component of y.

Our second modification to the improved *ParallelSolve* allows for a general mapping of columns to processors. This is required for wide separators, since the corresponding columns of L do not necessarily follow a strict wrap mapping. As in the dense algorithm, for each column k of L, some processer ϕ updates b using column k. However, rather than send a message after updating each of q segments, processor ϕ stops sending messages regarding updates using column k only after it reaches a component b_j, where column j is assigned to ϕ. Of course, when we do not have a wrap mapping of columns, Li and Coleman's analysis of the running time of the triangular solve no longer applies. However, this approach should still work well provided the mapping does not deviate too much from a wrap mapping. The sparse forward solver we have outlined above will pass at most $O(qn_s)$ messages, where n_s is the total number of columns in interprocessor separators. Each message contains at most $\lceil p/q \rceil$ components of b along with their indices and the number of times they have been modified.

As noted in the last section, the value of $q^*(n, 16, 40)$ is 4, for all $n \leq 691$, and this is about twice the optimal value of q. When we order our test problems using the narrow method of Chapter 2, the largest interprocessor separator consists of 83 columns. Hence, in our implementation of the forward solve, we have fixed q at 2. Note that according to Li and Coleman's analysis, we should decrease q as p decreases, i.e., as we solve for components correponding to higher level separators. In practice, decreasing q from 2 to 1, for the appropriate separators, produces no significant change in running times.

Once we have completed the forward solve $Ly = b$, we must perform the backward solve $L^T x = y$. In order to implement a backward solver similar to our forward solver, we would have to know the column structure of L^T for all those columns in interprocessor separators. That is, we would have to know the structure of row i of L for each column i in an interprocessor separator. The *RowSymFact* algorithm computes these structures, but then transposes them to determine the column structure of L. We could save the row structures for use during the backsolve, or we could recompute them as needed from the elimination forest. However, we elected to use neither the extra time or storage and implemented a much simpler algorithm that performs well in practice.

In solving $L^T x = y$, we must first compute those components of x corresponding to interprocessor separators. We use a variant of the method suggested at the end of Section 5.1 that communicates a $p-1$ length vector consisting of components of x. The only difference is that we allow for non-wrap mappings by having the processors send all the known values of x, rather than just the $p-1$ most recently computed components. Initially, all the processors use this algorithm to compute those x_k corresponding to the level-0 separator. After some processor computes the last such component, it sends the known components of x to the two processors assigned the last column of the two level-1 interprocessor separators. In parallel, these two processors then initiate the computation of those x_k corresponding to the two level-1 separators. Once the processors have computed all the x_k corresponding to interprocessor separators, they can compute its remaining values of x without any communication. This backward solver generates at most $O(n_s)$ messages, each containing at most $O(n_s)$ components of x along with their indices.

5.3 Numerical Results

We have implemented the sparse forward and backward solve algorithms of Section 5.2 and obtained implementations of different solvers from George, Heath, Liu, and Ng. We will refer to our algorithms as the *Separator* solvers, since they depend on the use of interprocessor separators, and those of George, Heath, Liu, and Ng as the *General* solvers, since they do not depend on the ordering. The General forward solver is similar to the Separator forward solver in that it performs its computation in a manner analogous to numeric factorization. Processor θ_k is responsible for computing y_k and waits until b_k has been modified $|Lrow(k)|$ times before doing so. Once θ_k has computed y_k, it multiplies each component of column

k by y_k and subtracts the result from the corresponding component of b. As each component b_j of b is updated, procesor θ_k determines if this is the last update it is going to make to b_j. If so, it immediately sends b_j, its index, and the number of times it has modified b_j to processor θ_j. For dense matrices, this algorithm is essentially the improved *ParallelSolve* algorithm with $q = p$. The General forward solver generates at most $O(pn_s)$ messages, each consisting of 2 integers and a component of b. Depending on the value of q used in the Separator forward solver, the General algorithm may send many more messages.

The General backward solver is quite straightforward. When a processor computes x_k, it broadcasts x_k to all the other processors. After receiving x_k, a processor scans its rows and updates its right hand side. Thus, the algorithm generates $O(n^2)$ messages, each containing component of x and its index. Unlike the Separator backward solver, the General algorithm does not exploit the assignment of columns to processors or the ordering. As a result, the processors spend a lot of time waiting for and broadcasting unneeded messages.

Table 5.1 lists the running times for all four triangular solve algorithms on our test matrices ordered by the narrow method. The Separator and General forward solvers require almost the same amount of time for all of the problems. However, with large enough problems, the Separator algorithm should run faster than the General algorithm. For the backward solve, the Separator approach is 4.1 to 11.4 times faster than the General approach. Comparing Tables 5.1 and 2.5, we see that the General backward solver requires more time than the numeric factorization for all but two of the test problems.

Table 5.1: Triangular solve times (seconds).

Problem	Forward Solve		Backward Solve	
	Separator	General	Separator	General
1	0.74	0.72	0.92	5.00
2	0.36	0.37	0.50	7.22
3	0.76	0.69	1.12	11.13
4	1.01	0.91	1.65	13.07
5	1.41	1.28	2.27	13.50
6	1.95	1.85	3.61	15.15
7	0.93	0.83	1.33	15.13
8	2.12	2.02	4.54	18.60
9	1.76	1.69	3.31	23.67
10	1.95	1.89	3.95	28.81

5.4 Discussion

In general, solving a triangular system requires much less work than numerically factoring a matrix. As a result, it is harder to design a parallel triangular system solver that masks communication with computation. The General backward solve algorithm requires $O(n^2)$ messages to perform only $O(|L|)$ work, and hence, message-passing overhead dominates its running time. Using this algorithm, the backward system solving phase, which is trivial on sequential machines, can be the most timing consuming part of the entire parallel computation. By exploiting both the sparsity of the matrix and the assignment of its columns to processors, we were able to develop a fast backward triangular system solver whose running times are small compared to numeric factorization times.

The General forward solver sends $O(pn_s)$ messages, while the Separator forward solver sends $O(qn_s)$ messages, where n_s is the total number of columns in interprocessor separators. In general, for fixed p, as the problem size increases, n_s will increase and q will decrease. Hence, for larger problems, we expect the Separator forward solver, with its lower message-passing overhead, to require less time than the General forward solver. For problems that have very good separators, n_s may be insignificant compared to n, and in this case, the two forward solvers may require roughly the same amount of time.

Recently, Heath and Romine [30] developed new dense column- and row-oriented triangular system solving algorithms that they call *wavefront* algorithms. These algorithms are more general versions of Li and Coleman's algorithms that do not require a wrap mapping of columns or rows to processors. In the column-oriented wavefront algorithm, each column is divided into segments of length k, for some fixed $1 \leq k \leq n$. After a processor updates the right hand side using a column segment, it sends the corresponding portion of the right hand side to the processor assigned the next column of L. Depending on the segment size, this algorithm can generate many more messages than the improved *ParallelSolve* algorithm. Heath and Romine's experiments indicate that wavefront algorithms perform best under a wrap mapping with large p and n moderately larger than p. Like the *ParallelSolve* algorithm, these algorithms could be adapted for sparse matrices. However, because of their higher message-passing overhead, they might only prove superior for mappings that deviate significantly from a wrap mapping or for limited ranges of p and n.

Chapter 6

Conclusion

Below we briefly discuss the main contributions of this thesis and provide possible directions for future research.

Our emphasis was on developing parallel algorithms for the four phases of sparse Cholesky factorization: ordering, symbolic factorization, numeric factorization, and triangular system solving. We designed our algorithms for a class of message-passing multiprocessors typified by hypercube machines. Since passing a message on these machines is typically much slower than performing a floating point operation, we developed parallel algorithms that require low communication overhead. Even with faster message passing, it would still be important to reduce communication overhead. We want the processors to spend as much time as possible doing useful work (i.e., flops), and to solve large problems, we want the message buffers to consume as little memory as possible. We also designed our algorithms so that the original matrix and its Cholesky factor always remain partitioned among the processors in a roughly even manner. This allows us to solve problems that are too large to reside in the memory available to any single processor.

We began by presenting a column-oriented sparse numeric factorization algorithm. We then developed a parallel algorithm for ordering and partitioning that orders the columns and assigns them to processors in a way that reduces fill and communication overhead during numeric factorization. This algorithm is a simple parallel version of the sequential Kernighan-Lin algorithm for finding small edge separators. We used this parallel Kernighan-Lin algorithm to find small interprocessor separators that evenly divide subgraphs of the original matrix onto subsets of processors. The sizes and the graph theoretic properties of these separators determine the amount of communication required during the remaining phases of the computation. Designing a more parallel algorithm for finding separators is an interesting open problem.

In our experiments, the Kernighan-Lin algorithm always converged quickly, regardless of the initial partition, but the quality of this partition affected the size of the resulting edge separator. To find smaller separators, we might want to develop a parallel heuristic for finding good initial partitions, such as a technique for finding highly connected subgraphs of a graph. To improve the final partition, we might want to assign weights to vertices and partition the graph into pieces of roughly equal total weight. If the weight of a vertex is proportional to the number of edges incident on it, this change could allow for a more even distribution of the columns belonging to the interprocessor separators and, hence, result in better load balancing.

In general, a parallel algorithm will perform better if it first decomposes the problem it is solving into parts that have high locality and require low communication overhead. Thus, applying a parallel graph partitioning algorithm, such as the parallel Kernighan-Lin algorithm, could be a useful first step for a wide variety of parallel problems. For Cholesky factorization, as well as other matrix problems, the problem of finding a good ordering along with a good assignment of nonzeros to processors deserves further study.

After examining the ordering phase, we showed that a column-oriented numeric factorization algorithm could be implemented more efficiently if each processor knew the elimination forest for the matrix. We presented a parallel algorithm for computing elimination forests and proved that it works correctly for any assignment of nonzeros to processors. Although the running times of both the parallel elimination forest algorithm and its sequential counterpart were small relative to the other phases of the computation, designing a more parallel and faster elimination forest algorithm for a message-passing multiprocessor is an interesting problem.

We then described a row-oriented parallel symbolic factorization algorithm. For large enough problems, this algorithm may generate fewer messages and have better processor utilization than an alternative column-oriented approach. Finally, we developed parallel forward and backward triangular solve algorithms that employ Li and Coleman's techniques for solving dense triangular systems. The backward solver is significantly faster than a more straightforward algorithm. In both the symbolic factorization and triangular system solving phases, the amount of work required is in general significantly less than that needed for numeric factorization. As a result, running times for these phases are very sensitive to how we map the problem on the processors and carry out the communication. Our algorithms for both phases depend on the existence of interprocessor separators. Faster, more general algorithms for these phases can probably be developed.

We designed our algorithms under the assumption they would be used to factor large matrices; however, all of our test problems are relatively small. On a Vax 780, Sparspak can numerically factor the largest of these problems in less than 30 seconds and requires a total time of under 3 minutes to perform all four phases of the computation on all ten test problems. Message-passing multiprocessors are not needed to solve problems of this size. Thus, we plan on experimenting with much bigger problems and much bigger hypercubes. To do so, we must revise some of our code to overcome limitations of the Intel hypercube, e.g., limits on message length, message buffer space, and total memory. These revisions are currently underway.

Overall, it appears that sparse Cholesky factorization can be implemented efficiently on a message-passing multiprocessor. As on sequential machines, the numeric factorization phase is the most time consuming phase for large problems. For our test problems ordered with the narrow method, the speed-up for this phase increased from 2.9 to 7.2 as the problem size increased. Since we use interprocessor separators to partition problems into independent parts, we expect the numeric factorization speed-up to continue to increase with the problem size, given a fixed number of processors and sufficiently sparse problems. Provided the processors are fast enough and contain enough memory, message-passing multiprocessors have the potential to solve large problems much more quickly than conventional sequential machines.

The current Ametek, Intel, and NCUBE hypercubes all have relatively slow processors and communication [9]. For an Intel hypercube without the optional vector boards, Moler [45] reported a single processor execution rate of 0.033 million floating point operations per second (megaflops), and Dunigan [9] reported that communicating an eight-byte message between adjacent processors is 26 times slower than performing a floating-point operation. Thus, the largest Intel hypercube, a 128 processor machine, has a maximum megaflop rate of 4.224. Unfortunately message-passing overhead and imperfect load balancing can result in much lower rates in practice. Using a full precision parallel version of LINPACK to perform an LU factorization of a dense matrix with 1890 equations, Moler [45] achieved 3.020 megaflops during the factorization phase and 0.079 megaflops during the triangular system solving phase. For comparison, in solving a system of linear equations of order 100 with full precision LINPACK [8], Dongarra [7] achieved 33 megaflops on a Cray X-MP, 2.5 megaflops on an IBM 370/195, and 0.13 megaflops on a Vax 11/780 with a floating point accelerator. Hence, even if present day hypercubes could achieve their maximum megaflop rates, they might not be worth the enormous effort required to rewrite existing sequential codes. As a result, one should view these machines simply as tools for studying parallel algorithms, rather than as high-speed state-of-the-art supercomputers. Hopefully, future hypercubes will have much faster processors and communication.

If hypercubes are to achieve widespread use, they must become easier to program. For example, on the Intel hypercube, all messages are vectors of some type. The user provides the vector and its length in bytes and opens a "communications channel" for the operating system to use in transmitting the message. An obvious improvement is to allow the user to send and receive arbitrary data structures without packing and unpacking them into vectors and without computing their sizes and opening channels.

Currently, hypercubes are programmed using conventional sequential programming languages (e.g., Lisp, C, Fortran) that have been extended with message-passing primitives. An important and difficult problem is to develop and implement new parallel programming languages to relieve the user of the burden of explicitly providing message-passing and synchronization details. Providing such details can easily be the most difficult aspect of programming these machines. For example, a sequential implementation of sparse forward triangular solve consists of 6 or 7 lines of code within two nested loops. In our parallel implementation, we have over 150 additional lines of code that is not directly related to computing flops. Most of this extra code involves sending and receiving data and interpreting data that have been received.

In designing a parallel language for a messsage-passing multiprocessor, we might allow the user to write programs that consist of blocks of sequential code. With each block, we could associate a list of data dependencies. When a processor assigned a particular block of code received all of the block's dependent data, it could execute the code. Of course, a processor might need to execute the same code for different data, so we would have to allow several data-dependency lists to be associated with a single block of code. The operating system could automatically handle the message passing. Whenever a processor signaled the completion of some computation (e.g., by setting a Boolean flag associated with the results), the operating system could automatically send the results to all the other processors that need them. As with computing elimination forests in parallel, the user might still have to explicitly compute the data dependencies, but perhaps even this computation could be automated in some cases.

In summary, message-passing multiprocessor programming currently resembles assembly language programming. The primitive operations are available, but the process of translating even simple ideas into working code is long and arduous. If these machines are going to establish themselves in the scientific computing community, the tools to program them at a higher level must be developed.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[2] Robert K. Brayton, Fred G. Gustavson, and Ralph A. Willoughby. Some results on sparse matrices. *Mathematics of Computation*, 24:937–954, 1970.

[3] Thang Bui, Soma Chaudhuri, Tom Leighton, and Mike Sipser. Graph bisection algorithms with good average case behavior. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, pages 181–192, 1984.

[4] R. M. Chamberlain. *An algorithm for LU factorization with partial pivoting on the hypercube.* Technical Report CCS 86/11, Chr. Michelsen Institute, 1986.

[5] R. M. Chamberlain and M. J. D. Powell. *QR factorization for linear least squares problems on the hypercube.* Technical Report CCS 86/10, Chr. Michelsen Institute, 1986.

[6] Thomas F. Coleman, Anders Edenbrandt, and John R. Gilbert. Predicting fill for sparse orthogonal factorization. *Journal of the Association for Computing Machinery*, 33:517–532, 1986.

[7] J. J. Dongarra. *Performance of various computers using standard linear equations software in a Fortran environment.* Technical Report 23, Argonne National Laboratory, 1984.

[8] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide.* SIAM Press, 1979.

[9] T. H. Dunigan. Hypercube performance. In Michael T. Heath, editor, *Hypercube Multiprocessors*, SIAM Press, 1987.

[10] T. H. Dunigan. *A message-passing multiprocessor simulator*. Technical Report ORNL/TM–9966, Oak Ridge National Laboratory, 1986.

[11] Anders Gunnar Edenbrandt. *Combinatorial Problems in Matrix Computation*. Ph.D. thesis, Cornell University, 1985.

[12] G. C. Everstine. A comparison of three resequencing algortihms for the reduction of matrix profile and wave front. *International Journal for Numerical Methods in Engineering*, 14:837–853, 1979.

[13] Tse-yun Feng. A survey of interconnection networks. *IEEE Computer*, 12:12–27, 1981.

[14] Geoffrey C. Fox and Steve W. Otto. *Concurrent computation and the theory of complex systems*. Technical Report CALT–68–1343, California Institute of Technology, 1986.

[15] George A. Geist and Michael T. Heath. Matrix factorization on a hypercube multiprocessor. In *Proceedings of the Conference on Hypercube Multiprocessors*, SIAM Press, 1986.

[16] George A. Geist and Michael T. Heath. *Parallel Cholesky factorization on a hypercube multiprocessor*. Technical Report ORNL–6190, Oak Ridge National Laboratory, 1985.

[17] Alan George, Michael T. Heath, and Joseph Liu. Parallel Cholesky factorization on a shared-memory multiprocessor. *Linear Algebra and its Applications*, 77:165–187, 1986.

[18] Alan George, Michael T. Heath, Joseph Liu, and Esmond Ng. *Sparse Cholesky factorization on a local-memory multiprocessor*. Technical Report ORNL/TM–9962, Oak Ridge National Laboratory, 1986. To appear in *SIAM Journal on Scientific and Statistical Computing*.

[19] Alan George, Michael T. Heath, Joseph Liu, and Esmond Ng. Symbolic Cholesky factorization on a local-memory multiprocessor. To appear in *Parallel Computing*, 1987.

[20] Alan George, Joseph Liu, and Esmond Ng. Communication reduction in parallel sparse Cholesky factorization on a hypercube. In Michael T. Heath, editor, *Hypercube Multiprocessors*, pages 576–586, SIAM Press, 1987.

[21] Alan George and Joseph W. H. Liu. An automatic nested dissection algorithm for irregular finite element problems. *SIAM Journal on Numerical Analysis*, 15:1053–1069, 1978.

[22] Alan George and Joseph W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, 1981.

[23] Alan George, Joseph W. H. Liu, and Esmond Ng. Communication results for parallel sparse Cholesky factorization on a hypercube. Submitted to *Parallel Computing*, 1987.

[24] John R. Gilbert and Hjálmtýr Hafsteinsson. *A parallel algorithm for finding fill in a sparse symmetric matrix.* Technical Report 86–789, Cornell University, 1986.

[25] John R. Gilbert and Robert Endre Tarjan. The analysis of a nested dissection algorithm. *Numerische Mathematik*, 50:377–404, 1987.

[26] John R. Gilbert and Earl Zmijewski. *A parallel graph partitioning algorithm for a message-passing multiprocessor.* Technical Report 87–803, Cornell University, 1987. To appear in *Proceedings of the International Conference on Supercomputing*, Springer-Verlag.

[27] John Russell Gilbert. *Graph Separator Theorems and Sparse Gaussian Elimination.* Ph.D. thesis, Stanford University, 1980.

[28] Frank Harary. *Graph Theory.* Addison-Wesley Publishing Company, 1969.

[29] Michael T. Heath. *Parallel Cholesky factorization in message-passing multiprocessor environments.* Technical Report ORNL–6150, Oak Ridge National Laboratory, 1985.

[30] Michael T. Heath and Charles H. Romine. *Parallel solution of triangular systems on distributed-memory multiprocessors.* Technical Report ORNL/TM–10384, Oak Ridge National Laboratory, 1987.

[31] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49:291–307, 1970.

[32] S. Kirkpatrick. Optimization by simulated annealing: quantitative studies. *Journal of Statistical Physics*, 34:975–986, 1984.

[33] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[34] Charles E. Leiserson. Area-efficient graph layouts (for VLSI). In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, pages 270–281, 1980.

[35] Guangye Li and Thomas F. Coleman. *A new method for solving triangular systems on distributed memory message-passing multiprocessors.* Technical Report 87–812, Cornell University, 1987.

[36] Guangye Li and Thomas F. Coleman. *A parallel triangular solver for a hypercube multiprocessor.* Technical Report 86–787, Cornell University, 1986.

[37] Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16:346–358, 1979.

[38] Richard J. Lipton and Robert Endre Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9:615–627, 1980.

[39] Joseph W. H. Liu. A compact row storage scheme for Cholesky factors using elimination trees. *ACM Transactions on Mathematical Software*, 12:127–148, 1986.

[40] Joseph W. H. Liu. Computational models and task scheduling for parallel sparse Cholesky factorization. *Parallel Computing*, 3:327–342, 1986.

[41] Joseph W. H. Liu. *Equivalent sparse matrix reordering by elimination tree rotations.* Technical Report CS–86–12, York University, 1986. To appear in *SIAM Journal on Scientific and Statistical Computing.*

[42] Joseph W. H. Liu. *Reordering sparse matrices for parallel elimination.* Technical Report CS–87–01, York University, 1987. Submitted to *Parallel Computing.*

[43] Joseph W. H. Liu. *The solution of mesh equations on a parallel computer.* Technical Report, University of Waterloo, 1974.

[44] Joseph W. H. Liu and Andranik Mirzaian. *A linear reordering algorithm for parallel pivoting of chordal graphs.* Technical Report CS–87–02, York University, 1987. Submitted to *SIAM Journal on Discrete Mathematics.*

[45] Cleve Moler. Matrix computations on an Intel hypercube. *Conference on Hypercube Multiprocessors, 1986*, To appear in Michael T. Heath, editor, *Hypercube Multiprocessors*, SIAM Press, 1987.

[46] S. Parter. The use of linear graphs in gauss elimination. *SIAM Review*, 3:119–130, 1961.

[47] Frans J. Peters. MIMD machines and sparse linear equations. In Gerard L. Reijns and Michael H. Barton, editors, *Highly Parallel Computers: Proceedings of the IFIP WG 10.3 Working Conference on Highly Parallel Computers for Numerical and Signal Processing Applications, 1986*, pages 201–210, North-Holland, published 1987.

[48] Frans J. Peters. Parallel pivoting algorithms for sparse symmetric matrices. *Parallel Computing*, 1:99–110, 1984.

[49] Charles H. Romine and James M. Ortega. *Parallel solution of triangular systems of equations.* Technical Report RM–86–05, University of Virginia, 1986.

[50] Donald J. Rose, Robert Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5:266–283, 1976.

[51] Youcef Saad and Martin H. Schultz. *Data communication in hypercubes.* Technical Report YALEU/DCS/RR–428, Yale University, 1985.

[52] Youcef Saad and Martin H. Schultz. *Data communication in parallel architectures.* Technical Report YALEU/DCS/RR–461, Yale University, 1986.

[53] Youcef Saad and Martin H. Schultz. *Topological properties of hypercubes.* Technical Report YALEU/DCS/RR–389, Yale University, 1985.

[54] Robert Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software*, 8:256–276, 1982.

[55] Charles L. Seitz. The cosmic cube. *Communications of the ACM*, 28:22–33, 1985.

[56] Paul Wiley. A parallel architecture comes of age at last. *IEEE Spectrum*, 46–50, June 1987.

[57] Earl Zmijewski and John R. Gilbert. *A parallel algorithm for large sparse symbolic and numeric Cholesky factorization on a multiprocessor.* Technical Report 86–733, Cornell University, 1986.

[58] Earl Zmijewski and John R. Gilbert. A parallel algorithm for sparse symbolic Cholesky factorization on a multiprocessor. To appear in *Parallel Computing.*