A Parallel Graph Partitioning Algorithm for a Message-Passing Multiprocessor*

John R. Gilbert
Earl Zmijewski†

TR 87-803
January 1987

Department of Computer Science
Cornell University
Ithaca, NY 14853

*Publication of this report was partially supported by the National Science Foundation under grant DCR-8451385.

†AT&T Bell Laboratories Scholar.

A Parallel Graph Partitioning Algorithm for a Message-Passing Multiprocessor*

John R. Gilbert
Earl Zmijewski†

Computer Science Department
Cornell University
Ithaca, New York 14853

9 January 1987

Abstract

We develop a parallel algorithm for partitioning the vertices of a graph into $p \geq 2$ sets in such a way that few edges connect vertices in different sets. The algorithm is intended for a message-passing multiprocessor system, such as the hypercube, and is based on the Kernighan-Lin algorithm for finding small edge separators on a single processor [14]. We use this parallel partitioning algorithm to find orderings for factoring large sparse symmetric positive definite matrices. These orderings not only reduce fill, but also result in good processor utilization and low communication overhead during the factorization. We provide a complexity analysis of the algorithm, as well as some numerical results from an Intel hypercube and a hypercube simulator.

*Publication of this report was partially supported by the National Science Foundation under grant DCR-8451385.

†AT&T Bell Laboratories Scholar.

1 Introduction

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 [15], efficient sparse Gaussian elimination [13,16], and solving various graph problems [17].

With the commercial availability of parallel machines, we are faced with the problem of developing efficient parallel algorithms for finding small separators of graphs. In this paper, we develop an algorithm for this problem that is based on a simple modification of the Kernighan-Lin algorithm [14] for finding edge separators on a single processor. We have designed our algorithm for a message-passing multiprocessor. One of the main advantages of our algorithm is that it can find separators of graphs that are too large to reside in the memory available to any single processor.

Our algorithm is designed for a class of message-passing multiprocessors typified by the currently available hypercube machines of Ametek, Intel, and NCUBE. 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 [6] for a survey of network topologies.

We have used our parallel separator algorithm in solving systems of linear equations of the form

$$Ax = b,$$

where A is a large sparse symmetric positive definite matrix. We can solve for x by computing the Cholesky factor L of A (i.e., the lower triangular matrix L such that $A = LL^T$) and then solving the systems $Ly = b$ and $L^T x = y$. 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 reordered system

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

for some permutation matrix P. Since A is positive definite, no pivoting is required to maintain numerical stability, and hence, we are free to choose P to make the factorization more efficient. We have developed a parallel ordering algorithm that uses our parallel separator algorithm. The algorithm attempts to find orderings that not only reduce fill, but also decrease the total volume of message passing and result in good processor utilization during the numeric factorization. We will consider only the ordering phase of sparse parallel Cholesky factorization. George, Gilbert, Heath, Liu, Ng, and Zmijewski [8,22,23] have examined the symbolic and numeric factorization phases of this computation. George, Liu, and Ng [9], in work done independently of ours, have used the elimination tree of a matrix to assign its columns to processors in a way that both reduces the communication and results in good processor utilization during the factorization.

Our paper is organized as follows. Section 2 reviews the Kernighan-Lin algorithm [14] for finding small edge separators of graphs. Section 3 contains a parallel version of this algorithm that is appropriate for message-passing multiprocessors. Section 4 provides some implementation details of the algorithm along with a discussion of its computational and communication complexity. Section 5 shows how the algorithm can be used to reorder the nonzeros of a sparse matrix in a way that reduces fill and communication during the factorization. Here, we assume the reader is familiar with the graph theoretic model of Cholesky factorization [11]. Section 6 contains some numerical results using both a hypercube simulator written by T. H. Dunigan of the Oak Ridge National Laboratory [4] and an Intel hypercube. Section 7 shows that the Kernighan-Lin algorithm will not necessarily find minimum edge separators for grid graphs. Section 8 contains remarks and conclusions.

2 The Kernighan-Lin Algorithm

In this section, we briefly review the Kernighan-Lin algorithm [14] 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. All other edges are *internal* edges. 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

$$g = D_a + D_b - 2c_{ab} \tag{1}$$

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.

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

$$D'_y = D_y + 2c_{yb} - 2c_{ya}, \quad y \in B - \{b\}. \tag{3}$$

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 and 3. 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 hopefully 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. In both implementations, they found that the algorithm almost always converges in 2 to 4 iterations

and that the probability of a single iteration finding an optimal solution is approximately $2^{-n/30}$, where n is 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 p sets, where p is a power of 2.

3 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 we are primarily interested in large sparse graphs, we assume that G is stored as a collection of adjacency lists. A processor is assigned variable $v \in V$ 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 $l_1 \in P_1$ and $l_2 \in P_2$, to be the *leader* of that part. If $s \in P_i$ then we will say that the leader of s is l_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 and 3. From these equations, we see that they need the adjacency lists of both selected vertices. The leaders request this information

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 $l_1 \in P_1$ and $l_2 \in P_2$, 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 l_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.

Algorithm 1: A parallel Kernighan-Lin algorithm.

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. The entire procedure is outlined in Algorithm 1.

To reduce the number of messages passed between 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.

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 7 and 8.

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.

4 An Implementation and Complexity Analysis

To analyze the computational and communication complexity of the parallel Kernighan and 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 Algorithm 1 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 Algorithm 1. 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 [14] 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 Algorithm 1, 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 [2,3] 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.

5 A Parallel Ordering Algorithm

As noted in Section 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. 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 below, 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.

George, Liu, and Ng [9] independently made similar observations and implemented an algorithm that sequentially orders the columns of a matrix on the host of a hypercube and then uses the elimination tree to assign the columns to processors. In what follows, we will use both narrow and wide vertex separators to order the columns of A. In a different setting, Liu [19] suggested both of these orderings and analyzed the parallelism that results during the outer product Cholesky factorization of grid graphs. We will discuss Cholesky factorization in terms of graphs [21] and will compute factorizations by columns. For a review of parallel sparse Cholesky factorization by columns see Gilbert and Zmijewski [22,23] and George, Heath, Liu, and Ng [8].

We can represent the structure of A by a graph $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ and $(v_i, v_j) \in E$ if and only if $a_{ij} \neq 0$. An *elimination order* of G is an ordering of the vertices of G which we will write as a one-to-one function $\alpha : V \rightarrow 1, \ldots, n$. Finding an elimination order on G corresponds to finding a permutation matrix P for A. That is, column (row) i of A is column (row) $\alpha(i)$ of PAP^T. The *filled graph* G^*_α of G with respect to α is the graph with the same vertices as G, whose edges are all those edges (u, w) such that 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 $1 < i < k$. If L is

the Cholesky factor of PAP^T, then $L + L^T$ is the adjacency matrix of G^*_α. Thus, finding P to reduce the amount of fill in L corresponds to finding an α to reduce the number of edges in G^*_α.

Nested dissection is an ordering heuristic that both reduces fill [11] and allows for parallelism [18,20]. Nested dissection begins by finding a set of vertices S contained in G whose removal would disconnect G into at least two components $C_1, \ldots, C_k$. The set S is a *vertex separator* of G. 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 eliminate vertices in different C_i in parallel [20]. 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 [12,16].

We use our parallel edge separator algorithm to find nested dissection orderings. We assume the columns of A (i.e., the adjacency lists of G) have been distributed among the p processors of a message-passing multiprocessor in some roughly even manner and that the processors are numbered from 0 to $p - 1$. We further assume that the processor assigned column i of A is responsible for computing column $\alpha(i)$ of L, the Cholesky factor of PAP^T.

First, the processors run the parallel edge separator algorithm on G. 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 eliminate the vertices in $V - V_1$. However, as the vertices in $V - V_1$ are eliminated, the corresponding columns of L 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 vertices that are not in the separator are eliminated.

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 eliminating 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 vertex in $V_1 \cup V_2$ is eliminated. 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 6 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 [11] 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 edge 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.

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, the vertices in V_1 are *wrapped* onto the processors in P_1. That is, if $V_1 = \{v_1, \ldots, v_k\}$ and $P_1 = \{p_0, \ldots, p_{l-1}\}$, then vertex v_i is reassigned to processor $(i-1) \bmod l$. This does not change the edge separator between P_1 and P_2. The vertices in V_2 are wrapped similarly onto the processors in P_2. In the case of a narrow separator, V_1 is wrapped onto all the processors in P_1 and P_2. Since vertex separators correspond to dense submatrices of L, and hence are more time consuming to eliminate, redistributing them 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.

Note that using either narrow or wide separators, at most $p/2^k$ processors need

Problem	Equations	Nonzeros	Density (%)
1	265	1753	2.50
2	406	2716	1.65
3	577	3889	1.17
4	778	5272	0.87
5	1009	6865	0.67
6	869	7285	0.96
7	918	7384	0.88
8	1005	8621	0.85
9	1007	8575	0.88
10	1242	10426	0.68

Table 1: Test problems.

to communicate in order to eliminate the vertices in a level-k vertex separator. On a hypercube, this implies that the columns of L corresponding to level-k vertex separators can be computed in a dimension-k 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.

6 Numerical Results

We have implemented the wide and narrow ordering algorithms of Section 5 that use the parallel Kernighan-Lin algorithm. We have added this code to Gilbert and Zmijewski's parallel symbolic factorization code [22], and George, Heath, Liu, and Ng's parallel numeric factorization and parallel triangular system solver codes [8,9]. The resulting collection of routines performs all phases of sparse Cholesky factorization in parallel. The code is written in Fortran and runs on both the Cornell Theory Center's Intel hypercube under Xenix with the beta version 3.0 of the node operating system and, using the Oak Ridge National Laboratories' hypercube simulator [4], on a Vax 780 under Berkeley Unix. We have used the simulator to generate communication statistics and the Intel hypercube to measure running times.

We have compared three algorithms for ordering the columns of a matrix

Problem	Seq-wrap	Narrow	Wide
1	1.18	4.49	4.30
2	1.98	5.23	5.13
3	2.94	7.26	7.21
4	4.18	7.01	6.93
5	5.72	7.61	7.50
6	5.06	9.19	9.35
7	5.80	17.04	14.96
8	7.44	22.58	22.23
9	5.88	9.44	9.33
10	8.02	15.30	15.29

Table 2: Ordering time (seconds).

and assigning them to processors: the narrow and wide algorithms of Section 3, 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 1. The first five problems are derived from L-shaped triangular meshes and are described by George and Liu [10]; the second five represent various physical structures and are described by Everstine [5]. In running our experiments, we used all 16 processors of the Cornell Theory Center's Intel hypercube.

Table 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 amoung 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

Problem	Seq-wrap	Narrow	Wide
1	2447 (2.14)	1739 (2.00)	1424 (1.81)
2	3745 (2.13)	2452 (1.96)	2101 (1.79)
3	5574 (2.11)	3259 (1.86)	2553 (1.71)
4	7706 (2.12)	4510 (1.95)	3606 (1.77)
5	10031 (2.12)	5568 (1.95)	4423 (1.76)
6	8161 (2.12)	3860 (1.91)	3267 (1.75)
7	8849 (2.13)	4992 (1.96)	4205 (1.83)
8	10123 (2.13)	5714 (1.98)	4980 (1.82)
9	10438 (2.13)	4843 (1.83)	3318 (1.60)
10	12897 (2.13)	7540 (2.00)	6312 (1.82)

Table 3: Message traffic during numeric factorization.

of problems 7, 8, and 10 were very poor. Due to message-passing delays, narrow and wide both require more time then seq-wrap in all cases. 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. 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 [8,9] (in an experimental version from summer 1986) to compute the Cholesky factor. For each problem, Table 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 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 all of the messages passed were smaller than 1024 bytes, we have listed only the total number of messages. As expected, the wide approach results in both the lowest total message

Problem	Seq-wrap	Narrow	Wide
1	32610	34548	60062
2	69466	71048	114658
3	125986	129580	221506
4	201594	228852	373867
5	312595	336249	579599
6	158444	160609	270999
7	239274	283613	651428
8	458580	501475	992610
9	258793	315552	608565
10	558595	781317	1437048

Table 4: Flops during numeric factorization.

Problem	Seq-wrap	Narrow	Wide
1	2.81	2.56	2.51
2	5.82	4.04	4.06
3	6.92	5.31	6.09
4	9.89	7.50	8.48
5	14.08	11.20	13.14
6	9.39	6.63	7.47
7	11.30	9.44	14.60
8	15.59	19.39	28.61
9	12.41	8.92	13.71
10	20.17	20.88	28.10

Table 5: Numeric factorization time (seconds).

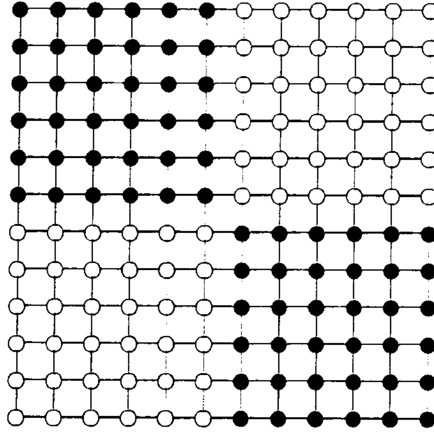

Figure 1: A partitioning of a 12×12 grid graph.

traffic and lowest average distance travelled per message.

Table 4 lists the total number of flops the processors perform during the numeric factorization. For many 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 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 42% to 72%.

Table 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. Thus, for some of these problems, the decrease in communication more than compensates for the increase in fill. We plan on conducting further tests with larger matrices.

7 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 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 1. 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 1, 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 shows the partition after swapping the first 30 pairs of vertices. The black vertices in the lower half of Figure 2 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 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 1, 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.

8 Conclusion

Lipton, Rose, and Tarjan [16] 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. There-

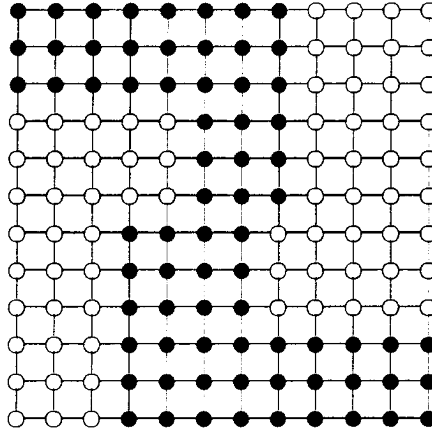

Figure 2: The partitioning after swapping the first 30 pairs of vertices.

fore, 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 7.

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.

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. Fox and Otto [7] describe a different approach to automatic partitioning and use it to solve various numerical problems on the Caltech hypercube.

Acknowledgements

We thank Laurie Hulbert for reading earlier drafts of this paper and for making suggestions that made it more readable. We also thank Alan George, Mike Heath, Joseph Liu, and Esmond Ng for allowing us to use their sparse parallel Cholesky factorization codes, and Mike Heath and Esmond Ng for providing us with the Oak Ridge National Laboratory hypercube simulator. Finally, we express our appreciation to Cornell University's Theory Center for the use of their Intel iPSC hypercube computer.

References

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

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

[3] 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.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] Alan George, Joseph Liu, and Esmond Ng. Communication reduction in parallel sparse Cholesky factorization on a hypercube. In *Proceedings of the Second Conference on Hypercube Multiprocessors*, SIAM Press, 1987. (to appear).

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

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

[12] John R. Gilbert and Robert Endre Tarjan. The analysis of a nested dissection algorithm. To appear in *Numerische Mathematik.*

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

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

[15] 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.

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

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

[18] Joseph W. H. Liu. *Computational models and task scheduling for parallel sparse Cholesky factorization.* Technical Report CS–85-01, York University, 1985. To appear in *Parallel Computing.*

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

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

[21] 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.

[22] Earl Zmijewski. *Sparse Cholesky Factorization on a Multiprocessor.* Ph.D. thesis, Cornell University, 1987. (in preparation).

[23] 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.