Efficient sparse Cholesky factorization on a parallel SIMD computer

Fredrik Manne* Hjálmtýr Hafsteinsson†

Abstract

We investigate the effect of load balancing when performing Cholesky factorization on a SIMD computer. In particular we describe a supernodal algorithm for performing sparse Cholesky factorization. The way the matrix is mapped onto the processors has significant effect on its efficiency. We show that this assignment problem can be modeled as a graph coloring problem in a weighted graph. By a simple greedy algorithm, we obtain substantial speed-up compared to previously suggested data mapping schemes. Experimental runs have been made on a 16K processor MasPar MP-2 parallel computer using symmetric test matrices with irregular sparsity structure. On these problems our implementation achieves performance rates of well above 200 Mflops in double precision arithmetic.

1 Introduction

The solution of sparse linear systems has long been important for many applications in science and engineering. As these applications are increasingly being moved to parallel computers it becomes crucial to find efficient algorithms to solve such linear systems. In many cases the matrices involved are symmetric positive definite so they can be factored by Cholesky factorization and the linear system solved by forward and back substitution. Iterative methods are also available for solving linear systems, but direct methods might be preferred for reasons of stability and accuracy.

*Department of Informatics, University of Bergen, Bergen High Tech Center, N-5020 Bergen, Norway (Fredrik.Manne@ii.uib.no).

†Department of Computer Science, University of Iceland, Reykjavík, Iceland (hh@rhi.hi.is). This work was performed while the author was visiting the Department of Informatics, University of Bergen.

Most of the recent work on parallel implementations of sparse Cholesky factorization has been for vector supercomputers and MIMD multiprocessors [16]. High performance has been achieved on the Cray Y/MP [27] and the Cray-2 [6]. Implementations on MIMD computers are mostly for hypercube architectures, such as the Intel iPSC [11][25][29].

Until recently the conventional wisdom has been that parallel computers based on the SIMD model are better suited for iterative rather than for direct algorithms for solving sparse linear systems. Gilbert and Schreiber [15] attempted to refute that belief by implementing a supernodal, multifrontal algorithm to compute the Cholesky factorization on the Connection Machine CM-2. Despite a disappointing Mflop rate they managed to show the feasibility of direct sparse methods on SIMD computers.

Kratzer [18] demonstrated a method for efficient sparse LU factorization on the MasPar MP-1 computer by mapping the data onto the processors in a special way. His implementation achieves throughput of up to 11.3 Mflops on a 4K-processor MP-1 using single precision data.

In this paper we implement sparse Cholesky factorization on a 16K processor MasPar MP-2 parallel computer. Our algorithm is similar to Kratzer's for LU factorization, but we introduce a new way of assigning the data to the processors. Direct comparison with the work by Kratzer is difficult because of the differences in computers, but an implementation of Kratzer's mapping scheme on our computer shows that our method gives an improved Mflop rate of about 20%.

We have done experiments on irregular sparse test matrices and have achieved performance rates of up to 219 Mflops in double precision and 287 Mflops in single precision. Even though the double precision peak rate of the 16K processor MP-2 is 2300 Mflops, we cannot realistically expect to get even close to that number, especially with irregular sparse data.

The paper is organized as follows. We begin in Section 2 with a brief description of the MasPar MP-2, the parallel computer which we use. Then we provide some background definitions for sparse Cholesky factorization in Section 3. In Section 4 we describe the data structures and algorithms in our implementation of the Cholesky factorization. Section 5 discusses how different ways of distributing the data onto the processors affect the performance of our factorization algorithm. We give numerical results for some practical test matrices in Section 6 and present our conclusions and observations in the final section of the paper.

2 The MP-2 machine

The MasPar MP-2 system is a massively parallel SIMD computer. It is an upgrade of the older MP-1 system [5], incorporating more powerful processor elements while using the same communication subsystem. The MP-2 consists of two parts: a high performance work station, which acts as a front-end for the system, and a data parallel unit (DPU). The DPU contains between 1024 (1K) and 16384 (16K) processor elements. They are arranged in a 2-dimensional, toroidal-wrapped grid called the processor array. The DPU also contains an array control unit (ACU), which provides an interface between the front-end and the processor elements.

All the processor elements receive the same instruction from the ACU at the same time and execute it on their local data. However, individual processor elements can disable themselves based on logical expressions and they can also use indirect references when referring to local data.

The MP-2 provides two types of communication between the processor elements called *Xnet* and *Router*. Xnet communication is the faster, but more restricted procedure. It follows the grid lines of the processor array. Processor elements can send data any distance to the north, south, west, and east, as well as to the northwest, northeast, southwest, and southeast. The grid lines wrap around, so each processor element always has a neighbor in each of these eight directions.

There are three types of Xnets: basic Xnet, XnetP(ipe), and XnetC(opy).

The basic Xnet takes time according to the formula

$$<\!startup\!> + <\!\#bits\!> * <\!dist\!> .$$

A typical value for 64-bit operands is $6 + 66*<\!dist\!>$ clock cycles, where $<\!dist\!>$ is at most 128. On the MP-2 a clock cycle is 80 ns. The XnetP is faster over long distances taking time

$$<\!startup\!> + <\!\#bits\!> + <\!dist\!>,$$

with a typical value being $91+<\!dist\!>$ clock cycles. The XnetC is similar, but leaves a copy of the transmitted value on each intermediate processor. However, XnetP and XnetC require that every intermediate processors between the sending and receiving processor be inactive.

XnetC provides an efficient way for broadcasting a value along a row or a column of the processor array. Sending a 64-bit value to the other 127 processors in the same row/column of a 16K machine (with a 128 by 128 processor array) takes only about three times as much time as a basic Xnet to a nearest neighbor.

Router communication allows each processor to send data to any other processor in the processor array. This makes it more flexible than the Xnet, but slower. The time for a Router communication varies with the amount of collisions, but averages out to about 6200 clock cycles for 64-bit operands.

In the programs reported on in this paper we use XnetC almost exclusively for communication between processor elements. This gives higher speed, but requires that data be distributed to the processors in a special way in order to take advantage of the XnetC. This is discussed further in Section 5.

Each processor element is a 32-bit load/store arithmetic processor with 40 32-bit registers and 64Kb of RAM. There is no floating point hardware and all floating point operations are thus implemented in software. If we define the average time of a floating point operation (flop) as $\alpha = \frac{1}{2}(\mathit{Mult} + \mathit{Add})$, the peak speed of a single processor element is 0.1412 Mflops for 64-bit arithmetic. A 16K processor machine would thus have the peak performance of 2314 Mflops.

Comparing the speed of arithmetic to communication on the MP-2, we obtain the ratio

$$\frac{Xnet[1]}{\alpha} = 0.8.$$

Thus floating point arithmetic on a 64-bit value is actually *more* expensive (by 20% than sending that value to the nearest neighbor in the processor array. It should also be noted that copying a 64-bit value to all the other processors in a column or a row of the PE array, using XnetC, costs only 2.5 times more than a single 64-bit floating point operation on the MP-2.

3 Linear algebra background

Let A be an $n \times n$ symmetric positive definite matrix. One way of solving the linear system $Ax = b$ consists of finding the unique lower triangular matrix L, such that $A = LL^T$. This is called Cholesky factorization of A. The linear system $Ax = b$ can then be solved by solving two triangular systems, $Ly = b$ and $L^T x = y$.

When A is sparse there are some additional steps in the Cholesky factorization. In general the Cholesky factor L will not be as sparse as A since the factorization usually introduces some nonzeros into L. These nonzeros are called *fill-in*. By a symmetric reordering of the columns and rows of A we can often reduce the fill-in. Finding the ordering that gives the least fill-in is NP-hard [28]. Two frequently used heuristics are

```
for j = 1 to n do
    for i < j where l_ji ≠ 0 do
        cmod(j, i);
    cdiv(j);
end-do
```

Figure 1: Sparse column-Cholesky

```
for i = 1 to n do
    cdiv(i);
    for j > i where l_ji ≠ 0 do
        cmod(j, i);
end-do
```

Figure 2: Sparse submatrix-Cholesky

Minimum Degree [14] and Nested Dissection [12].

After a fill-reducing ordering has been found, the next step in a sparse Cholesky factorization is the symbolic factorization. Its purpose is to find the nonzero structure of L so that memory can be reserved for the nonzeros before their numerical values are computed. In parallel Cholesky factorization it is equally important to know the structure of L to allocate work evenly among the processors.

The last step is the numerical factorization, in which the values of the nonzeros of L are calculated. This is usually the most time-consuming part of the Cholesky factorization.

It is often convenient to view the numeric factorization as a combination of the operations *cmod*(j,i) and *cdiv*(i). The operation *cmod*(j,i), working only on elements below the diagonal, subtracts a multiple of column i from column j, where the multiplier is l_{ji}, and the operation *cdiv*(i) divides the column i by the square root of its diagonal element. Using these vector operations we can define two variants of Cholesky factorization, *column-Cholesky* shown in Figure 1 and *submatrix-Cholesky* shown in Figure 2.

The column-Cholesky variant, which is the method most often used in sequential factorization codes, is also called a *fan-in* method [16]. Column j remains unchanged throughout the execution, until at some point in the algorithm it is modified by all the previous columns that have a nonzero in row j.

In submatrix-Cholesky, on the other hand, a column j is modified by column k immediately after k has been computed. Column k is sent to all the columns where it is needed and these columns are then modified. Thus this method is often called *fan-out* or *outer product*. The parallel numeric factorization algorithm in this paper is a variant of submatrix-Cholesky.

We now turn to some graph theoretic definitions that are useful in sparse Cholesky factorization.

The graph $G(A)$ of the symmetric matrix A is the graph with vertices $1, 2, \ldots, n$ and edges $\{(i,j) \mid a_{ij} \neq 0\}$. The *filled graph* $G^*(A)$ has vertices $1, 2, \ldots, n$ and edges $\{(i,j) \mid l_{ij} \neq 0\}$. Thus $G^*(A)$ represents the nonzero structure of the matrix $L + L^T$. The main purpose of the symbolic factorization step in sparse Cholesky factorization is to compute the graph $G^*(A)$ from $G(A)$.

An important structure in sparse Cholesky factorization is the *elimination tree* [21]. The elimination tree $T(A)$ is defined as follows. If vertex v has higher numbered neighbors in $G^*(A)$, then the parent of v in $T(A)$ is the lowest numbered such neighbor $p(v)$; otherwise v is a root. In matrix terminology the parent of column v is the row number of the first off-diagonal nonzero that appears in column v of the factor matrix L. Note that $T(A)$ is a subgraph of $G^*(A)$, the edges of $G^*(A)$ that are not in $T(A)$ are called *nontree edges*. If A is irreducible, which means that $G(A)$ is one connected component, then $T(A)$ is a single tree. From now on we will assume that A is irreducible.

The elimination tree $T(A)$ represents the dependencies between the columns of A during Cholesky factorization. The columns corresponding to leaves of $T(A)$ do not have to be modified by any other columns. Therefore we could perform the cdiv-operation on all the leaf columns simultaneously. Also note that two columns that are unrelated in the elimination tree do not depend on each other and could be computed in parallel, if all their descendants have been eliminated. Thus the height of $T(A)$ is closely related to the parallelism inherent in the Cholesky factorization of A. The lower the tree, the more columns can be eliminated in parallel.

In sparse Cholesky factorization we can often take advantage of the fact that some adjacent columns in the factor matrix L may have the same nonzero structure. A set of such columns is called a *supernode* [1]. Finding and utilizing supernodes in L can give benefits both in computation speed and storage space. In the elimination tree, a supernode appears as a path of nodes in the tree that have the same nontree edges to their common ancestors in the tree. The vertices of a supernode will form a clique in the filled graph G^*.

4 The factorization algorithm

In this section we describe our algorithm for performing sparse Cholesky factorization on the MP-2 computer. As mentioned in Section 3 it is similar to an algorithm for sparse LU-factorization by Kratzer [18]. We will assume that preordering and symbolic factorization have already been performed.

We will now show how the elements of the factor matrix L are stored on the processor array. We assume that preordering and symbolic factorization have already performed, so we have determined the positions of the nonzeros of L. The initial values of the nonzeros of L are 0 for the fill-in elements and the values of the corresponding elements in A for the others. The columns and rows are mapped onto the processor array according to a mapping function $M : [1..N] \rightarrow [1..P]$, where N is the dimension of the matrix A and P is the dimension of the processor array. If $M(i) = k$ then matrix column i is mapped to processor column k. Similarly, row i is mapped to processor row k. Thus element $a_{i,j}$ will be mapped to processor $(M(i), M(j))$. With this layout each processor in the same column of the processor array will have the same columns from the matrix mapped to it and each processor row will similarly have the same matrix rows mapped to it. The processors on the diagonal of the processor array will receive the same columns as rows, so each diagonal matrix element will therefore be mapped to a diagonal processor.

Each processor has an array *column_name* containing the indices of the matrix columns that are mapped to this processor. This array is ordered by increasing value.

The nonzeros allocated to each processor are stored in column-major order in a one-dimensional array. For each nonzero its floating point value and its local row index is stored. The local row indices are calculated relative to the number of rows mapped to each processor. In order to facilitate look-up into the array of nonzero elements, each processor has a vector giving the starting point of each column. The local column index of an element is given implicitly as the column it belongs to.

Since we assume that the matrix is positive definite each diagonal matrix element is nonzero. This means that the first element of each column on the diagonal processors is a diagonal matrix element. For each of these diagonal elements the local row column indices will be identical.

We now proceed to describe the algorithm. It is a parallel version of submatrix-Cholesky factorization. The main difference is that we perform the cdiv operations after all the cmod operations have been done. This way we avoid having one processor performing a cdiv operation while the rest of the processors are idle. Because of this we have to scale column i by l_{ji}/l_{ii} before subtracting it from column j when doing cmod(j, i).

The algorithm operates by performing $N-1$ outer products in a sequential order. The first step, when performing the ith outer product, is to distribute the element $l_{i,i}$ to all the processors. The off-diagonal elements from column i are then copied across the processor rows. This means that each nonzero entry $l_{j,i}$ ($j > i$) is copied to each processor in row $M(j)$, along with its local row index j'. Each processor that receives $l_{j,i}$ stores the floating point value in an array *guest* in position j'. In order to determine whether element j' in *guest* belongs to column i a separate integer array *time_stamp* is set to i in position j'.

If $l_{j,i}$ is nonzero, cmod(j, i) must be performed as a part of the ith outer product. Column j is mapped to processor column $M(j)$. Thus the arrival of $l_{j,i}$ and j' on the diagonal processor $(M(j), M(j))$ indicates that the local column j' should be modified by column i. Remember that on the diagonal processor, j' is also the local index of column j'. The processor $(M(j), M(j))$ therefore stores the value j' in a list *update*. When column i has been copied all across the processor array the list *update* will, on each diagonal processor, contain the local indices of the columns on which a cmod operation should be performed. Each diagonal processor now initiates the necessary cmod operations by first dividing each received element $l_{j,i}$ by $l_{i,i}$. The resulting floating point value, together with the local column index j' for column j, is copied to each processor in processor column $M(j)$. Each processor then looks through its elements in column j' and updates the ones for which *time_stamp*, indexed by the local row index, is equal to i.

In order to avoid having to determine which processors have received an element after something has been sent, each processor sets to 0 the variable in which it is going to receive the local index. The data structure is such that column and row zero contain no elements and are never used.

The complete algorithm, without the cdiv operation, is given in Figure 3. In the algorithm the index of each element is the local index. Capital variables indicate that they are global and have the same value for each processor. On the MasPar each processor is identified by its coordinates in the processor array, denoted by the variables ix and iy. In the algorithm we use this feature to determine if a processor is on the diagonal or not. The **all** statement makes all the processors active, and $\mathbf{copyS}[P].y = x$ is the XnetC, which copies the value in variable x into the variable y on the next P processors to the south. Since the grid wraps around we use this statement to broadcast the value of x on one processor, into y on all the processors in the same processor column.

The algorithm in Figure 3 actually only computes something resembling an LDL^T-factorization. To get the Cholesky factorization we have to divide each column with the square-root of its diagonal element, i.e., a cdiv operation. The reason for doing the cdivs last is that it allows us to perform simultaneously one cdiv operation on every processor column. Since the square-root operation is expensive, we want to maximize the number of processors doing a square-root at the same time. To do this we distribute diagonal elements down each processor column, perform the square-root, and then let each processor send its result down the processor column in an ordered fashion. This lets us trade a square-root operation for an XnetC operation and some loop overhead. Since a double precision square-root takes approximately 10 times longer to execute on the MasPar than an XnetC, this arrangement saves execution time. The code for the cdiv operations is given in Figure 4. In the algorithm the variable nc contains the number of columns allocated to each processor, and the $\mathbf{sendS}[i].y = x$ is the XnetP, which sends the value of variable x to the variable y on the ith processor to the south.

We take advantage of supernodes in order to reduce overhead in the algorithm. The supernodes of the matrix are found in a preordering step. Storage and scheduling of the supernodes is done on the ACU. While processing the first column in a supernode each processor stores the index of the column it is updating and also the index of the elements that are updated. Thus when processing subsequent columns in the same supernode each processor knows which columns and elements to update. This way only the floating point values have to be distributed. However, when column i is being distributed across the processor array we actually also distribute the local row index. We found this to be faster than storing it in a table on each processor.

5 Load Balancing

In this section we will show how the problem of finding a good mapping M for the algorithm presented in Section 4 can be viewed as a graph coloring problem on a weighted graph. Based on this observation we develop an algorithm that constructs a mapping that should make the Cholesky factorization more efficient. First we discuss the effect of the mapping on the performance of the Cholesky factorization.

```
i := 1;
for I := 1 to N − 1 do
    k := 0;
    r := 0;
    if column_name[i] = I
        if (ix = iy)
            copyS[P].di := l_{i,i};
        copyE[P].di := di;
        for each non-diagonal element l_{t,i} in column i do
            copyE[P].fv := l_{t,i};
            copyE[P].k := t;
            all
                time_stamp[k] := I;
                guest[k] := fv;
                if (ix = iy) and (k ≠ 0)
                    r := r + 1;
                    update[r] := k;
                end-if
                k := 0;
            end-all
        end-do
        i := i + 1;
    end-if

    k := 0;
    for j := 1 to r do
        fv := guest[update[j]] / di;
        copyS[P].fv := fv;
        copyS[P].k := update[j];
        all
            for each element l_{s,k} in column k do
                if time_stamp[s] = I
                    l_{s,k} := l_{s,k} − fv ∗ guest[s];
            k := 0;
        end-all
    end-do
end-do
```

Figure 3: Parallel Cholesky factorization

```
for j := 0 to ⌊nc/P⌋ do
    if ix = iy
       for i := 1 to min{P − 1, nc − 1 − P ∗ j} do
          sendS[i].fv := l_{1,i+1+j∗P};
       fv := l_{1,1+j∗P};
    end-if

    fv := √fv;

    for i := 1 to min{P, nc − P ∗ j} do
       if ((ix + i − 1 mod P) = iy)
          copyS[P].fw := fv;
       for each element l_{s,i+j∗P} in column i + j ∗ P do
          l_{s,i+j∗P} := l_{s,i+j∗P}/fw;
    end-do
end-do
```

Figure 4: The cdiv operation

5.1 Effect of mapping

We start by considering the cmod operations since our experiments showed them to be the most time-consuming. Consider the ith outer product. Let $S = \{r_1, r_2, ..., r_k\}$ be the row indices of the nonzero elements below the diagonal in column i. Then in stage i we must perform cmod(r_j, i) for each $1 \leq j \leq k$. Thus if each column in S is mapped to a separate processor column each of these cmods can be performed in parallel in one step. If two or more columns from S are mapped to the same processor column we will have to perform their corresponding cmods sequentially. Thus when performing the ith outer product we have to perform as many cmod operations as the maximum number of columns from S that are mapped to the same processor column.

Note that the number of copy operations needed to distribute column i across the processor array is equal to the maximum number of cmod operations that are performed in the ith outer product. This implies that the maximum number of elements that any processor stores from column i is equal to the number of cmod operations performed. Thus a mapping that severely limits the number of cmod operations each processor participates in will reduce both the number of steps needed to spread column i and the amount of storage needed to hold column i.

We now look at how the mapping effects the multiplications within each cmod operation of the ith outer product. Assume that $|S| > 2$ and let $1 \leq b < c < d \leq k$. It then follows that $l_{r_c,r_b} \neq 0$ and $l_{r_d,r_b} \neq 0$ because of fill. Thus when performing cmod(r_b, i) both l_{r_c,r_b} and l_{r_d,r_b} must be modified. The element l_{r_c,r_b} is mapped to processor $(M(r_c), M(r_b))$, and l_{r_d,r_b} is mapped to processor $(M(r_d), M(r_b))$. Thus if $M(r_c) = M(r_d)$ these two elements are mapped to the same processor. This would then imply that for cmod(r_b, i) we have to perform two updates (each consisting of one multiplication and one subtraction) on processor $(M(r_c), M(r_b))$. It follows that when performing cmod(r_b, i) we have to perform as many updates on processor column $M(r_b)$ as the maximum number of elements in S, each greater than r_b, that are mapped to the same processor column.

We note that even if each column in S is mapped to a separate processor column it is not true in general that one can do all the multiplications and subtractions in one step. The reason for this is that if i is the first column of a supernode each processor searches through the local column that is being updated and performs an update (multiplication and subtraction) whenever it finds an element where *time_stamp* is set to i. Thus due to the SIMD nature of the MP-2 we might have to perform more multiplication steps than one. This, however, is true only for the first column in a supernode. For subsequent columns the position of the elements being updated is known and if each column in S is mapped to a separate processor column, the updates are performed in one step.

Finally, we note that the execution time of the cdiv operations depends upon the number of columns that are mapped to the same processor column and how well the nonzeros of each column are spread along its processor column. Since the total time of the Cholesky factorization is completely dominated by the outer products it is not as important to spread the matrix columns evenly on the processor columns in order to reduce the time spent on the cdivs.

From the above discussion we propose the following measure of the quality of a mapping: The total number of times the processors need to perform a cmod operation should be as low as possible.

As we have discussed this will also reduce the number of multiplications, speed up the spreading of each column, and reduce the maximal number of nonzeros assigned to each processor from each matrix column.

5.2 Theory

We now try to find mappings giving few cmod operations on each processor column. Consider again the ith outer product. It will update only columns that are both ancestors of i in the elimination tree and adjacent to i in the filled graph. Thus two columns that are unrelated in the elimination tree will never be updated by the same outer product. This was observed by Kratzer [18] who proposed that column i should be mapped to processor column $M(i) = (L(i) \bmod P)$, where $L(i)$ is the length of the path in the elimination tree from i to the root of the tree. The value $L(i)$ is the *level* of i in the elimination tree. Thus we see that two columns i and j, where $(L(i) \bmod P) = (L(j) \bmod P)$ will be mapped to the same processor column. If $l_{i,k}$ and $l_{j,k}$ are nonzeros, then in the kth outer product, processor column $(L(i) \bmod P)$ will have to perform two cmods.

We now proceed to set up a graph-theoretical framework that allows us to look at this problem of minimizing the number of cmods from a different view point.

Let $G(A)$ be the adjacency graph of a matrix A. The filled graph $G^*(A)$ can be constructed by playing the *elimination game* on $G(A)$. The game consists of eliminating the vertices of $G(A)$, one after the other, and at the same time adding fill edges to $G^*(A)$. When vertex i is eliminated it is removed from consideration and its higher numbered neighbors are made into a clique by adding edges.

We now have a succession of possibly overlapping cliques, $F_1, \ldots, F_{N-1}$, in $G^*(A)$, where the clique F_i contains the higher numbered neighbors of vertex i. These cliques are also called *fronts* and play an important role in the multifrontal method [8][22].

The correspondence to the matrix point of view is that if $\{r_1, \ldots, r_k\}$ are vertices in clique F_i then the nonzeros in column i of L will be in rows $\{r_1, \ldots, r_k\}$. The mapping of the matrix columns to the processors can be viewed as a coloring of the vertices of $G^*(A)$. Assuming we have P different colors, then the maximum number of vertices with the same color in a clique F_i is the same as the number of cmods that have to be done by a processor column in stage i of the factorization.

The mapping problem can now be reformulated. We want to color the vertices of $G^*(A)$ using P colors in a way that minimizes the sum of the maximum number of vertices with the same color in each frontal clique F_i. More formally:

Given $G^*(A)$, with frontal cliques $F_1, \ldots, F_{N-1}$, and a positive number P. Find a coloring $M : \{1, \ldots, N\} \to \{1, \ldots, P\}$ such that

$$\sum_{i=1}^{N-1} C_i$$

is minimum, where C_i is the maximum number of vertices with the same color in clique F_i.

If we can solve this problem exactly then we will have a mapping that gives the minimum number of cmod operations for the Cholesky factorization algorithm presented in Section 4. The mapping is not guaranteed to be optimal with respect to the time that the factorization takes, but since the cmods take such a big portion of the time we are likely to be close to that optimum.

Assume that we have clique $F_i = \{j_1, j_2, j_3, j_4\}$. If the vertices j_1 and j_2 are colored with the same color p, then there is no extra cost in assigning the other two vertices j_3 and j_4 the same color q, $p \neq q$. This aspect of the problem complicates the task of designing efficient algorithms that generate good solutions. Therefore we have reformulated the problem in order to make it easier to design good approximation algorithms. However we want those algorithms to give solutions that also are good for the original mapping problem.

In the new problem we introduce weights on the edges of $G^*(A)$, giving the weighted graph $G^*_w(A)$. The weight w on each edge (j, k) is defined as follows:

$$w(j,k) = |\{i : (j,k) \in F_i\}|$$

The value $w(j,k)$ is the number of frontal cliques that j and k are both members of. Equivalently, $w(j,k)$ denotes how many outer products update both columns j and k. If $w(j,k)$ is large then it will be difficult to assign j and k colors without conflicting with the colors of other nodes. If $M(j) = M(k)$ then there are $w(j,k)$ frontal cliques that contain at least two nodes with the same color. Thus, if $w(j,k)$ is large the likelyhood of $M(j) = M(k)$ giving a contribution to some C_i increases. This leads to the following weighted graph coloring problem:

Given G_w^* and a positive number P. Find a coloring $M : \{1, \dots, N\} \rightarrow \{1, \dots, P\}$ such that

$$\sum_{M(j)=M(k)} w(j,k)$$

is minimum.

The above problem can be seen to be equivalent to a problem called *Multiway Partition* [19]. This problem is concerned with breaking a weighted graph up into a fixed number of parts, minimizing the weight of the external edges lying between the parts. The equivalence comes from the fact that the Multiway Partition is maximizing the weight of the internal edges, i.e., edges between nodes with the same color, and by negating the weights we get the above coloring problem. The Multiway Partition problem for general graphs is NP-hard, even when restricted to only two parts [10]. Since $G_w^*(A)$ is chordal it might be possible to solve the graph coloring problem in polynomial time on this graph, although we know of no such method.

There are a number of heuristic algorithms for the Multiway Partition problem for general graphs [3][9][17]. Most of them are based on repeated application of a heuristic for bipartition. They are quite complex and take limited advantage of special structure in the the input graph.

We implemented one approximation algorithm for the weighted coloring problem on general graphs [26], but it did not give good solutions for our input graphs, in spite of the relatively high time complexity of $O(N* \mid E \mid *P)$, where E is the edge set of $G_w^*(A)$.

5.3 The Algorithm

We are now ready to describe our approximation algorithm for the weighted coloring problem.

A feature of G_w^* is that edges between higher-numbered vertices in general have more weight than edges between lower-numbered ones. It is therefore important to get a good mapping of the higher-numbered vertices. Based on this observation we propose the following algorithm for computing M: Set $M(N) = 1$. Then for each $j = N - 1$ down to 1, choose $M(j)$ so that the quantity

$$\sum_{M(j)=M(k), k>j} w(j,k)$$

```
M[N] := 1;
for i := N − 1 downto 1 do
    assigned := 0;
    for each edge (i, j) where j > i do
        if last[M[j]] ≠ i then
          cost[M[j]] := w[i, j];
          last[M[j]] := i;
          assigned := assigned + 1;
        else
          cost[M[j]] := cost[M[j]] + w[i, j];
    end-do
    if assigned = P
      find k such that cost[k] is minimum
    else
      find k such that last[k] ≠ i;
    M[i] := k;
end-do
```

Figure 5: Calculating the mapping function M

is minimized. With this algorithm $M(j)$ can be calculated in time linear in the number of higher numbered vertices incident to i. Thus given G_w^*, each value of M can be computed in time linear in the number of edges in G_w^* (or nonzeros in L). The complete algorithm is given in Figure 5. The array *cost* is used to accumulate the cost of coloring the current vertex with a given color. The main purpose of the array *last* is to avoid having to clear the *cost*-array in each iteration of the outer loop. It records which colors currently have positive cost, associated with them in the current iteration. Thus if $last[r] = i$ then color r has already had cost assigned to it in this iteration, so we can add to that value, otherwise the value in $cost[r]$ is from a previous iteration and should be overwritten.

When we determine k in the algorithm in Figure 5 we start the search from position $L(i) \bmod P$, where $L(i)$ is the level of column i in the elimination tree. This is done in order to get an even spread of the columns that can be colored at no cost.

In our experiments this approximation algorithm for the weighted coloring problem gave quite good solutions to the original problem of minimizing the number of cmod operations. We infer this from the fact that the solutions were never more than 15%

```
for i := 1 to U do
    for j := sup[i] + 1 to sup[i + 1] − 1 do
        for each edge (j, k) where j < k do
            w[k, j] := w[k, j] + j − sup[i];

    j := sup[i + 1] − 1;
    for each edge (j, k) where j < k do
        for each edge (j, l) where k < l do
            w[l, k] := w[l, k] + sup[i + 1] − sup[i];
end-do
```

Figure 6: Calculating the weights

from a lower bound that we believe to be somewhat lower than the optimal solution. We did try some other simple heuristics, without managing to improve upon the one we present here. It should be noted though, that in our algorithm we take advantage of special properties of the input graph, so even if it seems to give us good solutions we do not expect the algorithm to do as well on general graphs.

So far we have not discussed how the weight of each edge can be calculated efficiently. We do this in a way quite similar to the symbolic factorization. We consider the nodes from 1 through N. When considering node i we add a weight of one to each edge going between higher numbered neighbors of i in G^*_w. Thus we increase the weight by one on all those edges that the symbolic factorization would have added to G if they were not already present. Just like in symbolic factorization we may take advantage of the supernodes when calculating the weights of the edges in G^*_w. Consider supernode J, consisting of columns i through j. We first look at a node k such that $i < k \leq j$. Then each edge going from vertex k to higher-numbered vertices will receive an extra weight of $k - i$ from the other nodes in J. Now looking at nodes that are outside the supernode J, we know that each edge that will be updated by vertex j will also be updated by the other vertices in supernode J. Thus it is sufficient to increase the weight of the edges that would have been updated by j by a factor of $j - i + 1$. The complete algorithm is shown in Figure 6. The indices giving the beginning and end of each supernode are stored in the array *sup*. There are U supernodes altogether.

Since the calculation of weights and symbolic factorization are similar kinds of operations they could be merged into one operation giving low overhead compared to the symbolic factorization alone. In our present implementation however, we do these operations separately. This is because we use routines from Sparspak [13] for the symbolic factorization.

Finally we suggest another way of characterizing the weights of the edges of G^*_w. This method is based on *row subtrees* [21]. The ith row subtree of L is a subtree of the elimination tree for L rooted at node i, that has been pruned of all nodes that are not adjacent to i in G^*. The weight $w(j,k)$ can now be interpreted as the number of vertices in the intersection of the row subtrees of j and k. This representation can potentially lead to a method of calculating the weights without explicitly computing the symbolic factorization.

6 Numerical Results

We have implemented the numeric factorization algorithm from Section 4 on the MasPar MP-2 computer. The program was written in the programming language MPL, which is based on ANSI C, extended to support data parallel programming. It was run on a 16K processor MP-2 with 64Kb of processor element memory and also for comparison on a 4K (64-by-64 PE array) partition of the same machine.

The test matrices fall into two groups. One contains BCS structural engineering matrices from the Harwell-Boeing collection [7]. They were ordered using the minimum-degree algorithm from Sparspak [13]. Symbolic factorization was performed sequentially with a routine from Sparspak.

The other type of test matrices represent two-dimensional $k \times k$ nine-point grids and three-dimensional $k \times k \times k$ twenty-seven point grids. They were preordered by Sparspak's automatic nested dissection heuristic and the symbolic factorization computed sequentially. After the symbolic factorization of each matrix we computed its partition into supernodes.

We show some of the characteristics of the test matrices in Table 1. The dimension of each matrix is given as well as the number of nonzeros in its Cholesky factor and the height of the elimination tree. The column labeled *Work* gives the number of floating point operations involved in factoring the matrix in Mflops. Columns *NZ64* and *NZ128* denote the maximum number of nonzeros per processor in the 4K and 16K processor implementation respectively, if the nonzeros would be distributed as evenly as possible. These numbers are really $\lceil$<#nonzeros> /4K$\rceil$ and $\lceil$<#nonzeros> /16K$\rceil$.

In addition Table 1 gives lower bounds on the total number of cmod-operations

Matrix	Dim	Nonzeros	Etree	Work	NZ64	NZ128	LB64	LB128
bcsstk23	3134	437,177	825	126.10	107	27	8603	5396
bcsstk24	3562	298,426	579	39.02	73	19	6310	4329
bcsstk25	15439	1,581,668	2392	358.94	387	97	33717	22747
bcsstk30	28924	4,615,864	3179	1,412.45	1127	282	84866	51389
bcsstk31	35588	6,039,557	2887	3,272.38	1475	369	112490	69256
bcsstk33	8738	2,667,875	2055	1,341.63	652	163	45718	25108
100-2D	10000	321,681	378	20.51	79	20	11455	10278
16-3D	4096	586,524	902	134.95	144	36	11153	6900
21-3D	9261	1,931,839	1595	735.32	472	118	34631	20175
25-3D	15625	4,066,777	2301	2,116.80	993	249	71076	40124
30-3D	27000	8,816,024	3364	6,228.71	2153	539	150647	83033
32-3D	32768	11,567,458	3835	9,171.01	2825	707	196386	107916
36-3D	46656	19,023,715	4885	18,671.27	4645	1162	319678	173842

Table 1: Characteristics of the test matrices

required for factoring the matrix. These numbers were found by calculating for each column the number of nonzeros below the diagonal divided by the number of processor columns (64 or 128) and taking the ceiling of the result. This gives the minimum number of cmods needed when performing the outer product with this column, if all the processor columns are working simultaneously. Adding these numbers for all columns of the matrix gives the values denoted by *LB64* and *LB128*. The number of cmods required is the main distinguishing factor between the different mapping schemes discussed in Section 5.

Table 2 shows the results of factoring the test matrices on a 128-by-128 PE array (16K processors) in single and double precision. We tried three different assignments of the nonzeros to the processors. The method called *Min_Cost* is our new method described in Section 5, *Levels* refers to the scheme of Kratzer [18], and C&S is short for the "Cut & Stack" method where the matrix is cut up into pieces that are P-by-P and stacked on the processor array in the obvious way (i.e. $M(i) = i \bmod P$). This last method is a simple approach that works well for dense problems [4], and we include it here to show how it can be improved upon for sparse factorization.

Column *Nz* gives the maximum number of nonzeros on any processor for each layout scheme and each matrix. Column *Cl* contains the maximum number of matrix columns (and rows) assigned to any one processor. We also give the number of cmods required for each mapping method. This number should be compared to the lower bound for 16K processors in column *LB128* of Table 1. Note that the estimated lower bound

Matrix	Layout	Nz	Cl	Cmods	Ut	ST	SF	DT	DF
bcsstk23	Min_Cost	66	43	6066	29.0	1.465	86.09	1.867	67.53
	Levels	71	46	7007	25.4	1.566	76.13	2.112	59.71
	C&Sk	56	25	7717	21.2	1.856	67.96	2.376	53.06
bcsstk24	Min_Cost	47	47	4679	19.4	0.999	39.07	1.242	31.42
	Levels	55	42	5653	14.8	1.174	33.23	1.482	26.33
	C&S	49	28	7184	9.5	1.514	25.76	1.930	20.22
bcsstk25	Min_Cost	234	173	25073	23.8	5.906	60.77	7.438	48.26
	Levels	255	184	32580	16.7	7.647	46.94	9.708	36.97
	C&S	226	121	36467	13.3	8.775	40.91	11.202	32.04
bcsstk30	Min_Cost	428	249	58230	34.6	12.975	108.86	16.754	84.31
	Levels	471	264	73498	26.5	16.113	87.66	20.922	67.51
	C&S	515	226	84276	21.0	18.879	74.82	24.678	57.23
bcsstk31	Min_Cost	593	330	76716	45.6	19.566	167.25	25.520	128.23
	Levels	677	352	94560	37.3	23.649	138.38	30.876	105.98
	C&S	651	279	105450	32.2	26.533	123.33	34.749	94.17
bcsstk33	Min_Cost	218	82	28486	47.1	6.953	192.95	9.276	144.64
	Levels	226	78	34416	38.1	8.388	159.94	11.151	120.31
	C&S	240	69	36040	34.8	8.948	149.94	11.931	112.45
100-2D	Min_Cost	128	128	10396	5.8	2.409	8.52	2.914	7.04
	Levels	135	131	11652	4.6	2.680	7.65	3.268	6.28
	C&S	107	79	15175	2.8	3.473	5.91	4.313	4.76
16-3D	Min_Cost	67	46	7876	26.0	1.790	75.38	2.283	59.13
	Levels	79	46	9862	20.0	2.223	60.72	2.850	47.35
	C&S	72	32	11652	14.6	2.716	49.68	3.516	38.39
21-3D	Min_Cost	180	89	23232	36.0	5.643	130.30	7.357	99.95
	Levels	188	93	28495	29.4	6.876	106.94	8.969	81.99
	C&S	198	73	32276	23.3	8.060	91.23	10.587	69.45
25-3D	Min_Cost	350	145	46327	43.5	11.960	177.00	15.793	134.03
	Levels	395	160	56946	34.9	14.806	142.97	19.529	108.39
	C&S	385	123	62861	28.8	16.986	124.62	22.521	93.99
30-3D	Min_Cost	714	245	94197	50.5	26.910	231.46	35.987	173.08
	Levels	732	240	114324	42.0	32.608	191.02	43.482	143.25
	C&S	781	211	124879	36.2	36.607	170.15	48.996	127.13
32-3D	Min_Cost	928	299	122746	53.0	36.380	252.09	48.832	187.81
	Levels	909	280	146589	45.3	43.189	212.35	57.776	158.73
	C&S	972	256	161579	37.8	49.105	184.92	66.674	137.55
36-3D	Min_Cost	1399	398	194258	57.7	65.021	287.16	85.331	218.81
	Levels	1478	411	235768	48.1	76.523	244.00	103.276	180.79
	C&S	1548	365	255711	42.1	85.327	218.82	115.516	161.63

Table 2: Results for 16K processor MasPar MP-2

is probably less than the optimal value and should only be used as an indication of how close to the optimal the values for different mapping methods are. The column labeled *Ut* gives the average percentage of processors taking part in each relevant arithmetic operation (multiplication and subtraction). This number indicates the processor utilization for each test problem. It would be possible to increase this percentage somewhat by assembling in a list all the operations needed to be done by each processor column and then performing them simultaneously, but this was found to increase the total factorization time due to overhead. The time in seconds for single precision factorization is in ST and the resulting Mflop rate in *SF*, and finally the time and Mflop rate for double precision are in columns DT and DF respectively.

The results show that the Min_Cost layout scheme consistently gives 10–30% fewer cmods than the Levels scheme. This translates into correspondingly less time required for the factorization and a higher Mflop rate. The Levels method gives in turn 10–20% fewer cmods than the simple Cut&Stack. Thus, in our experiments, the new Min_Cost scheme is up to 50% better for sparse factorization than the standard method for dense matrix computation.

The large difference between the lowest double precision throughput of 7 Mflops and the highest one of 219 Mflops is characteristic of SIMD computers. The matrix 100-2D that gets the lowest throughput is very sparse and has a relatively high dimension compared to the amount of work required to factor it. Thus a large proportion of the time is spent on overhead, scanning through the columns, and looking for computation to do. Note however, that the number of cmods given by the Min_Cost method for this matrix is only 1% from the lower bound. The matrix getting the highest throughput, 36-3D has less than 2% density in its Cholesky factor, but has enough operations to keep the processors busy doing useful work.

When considering the Mflop rate it should be noted that the factor matrices we use are slightly denser, and thus provide more work, than the same matrices in other papers [18][25]. For instance the factor matrix for bcsstk30 that Kratzer [18] uses is about 0.9% dense, but our factor matrix is 1.1% dense. The amount of work is not reported in the Kratzer's paper, but is probably proportionally less than what we have. This makes it difficult to compare the Mflop rates even for the same matrices. The reason for this difference is that the preordering algorithms available to us give more fill than the algorithms used in the other papers.

An interesting sidenote is that in order to do the cdiv operations last, our algorithm actually does more floating point operations that the number given in Table 1. However we still use the value from the table to compute the Mflop rate.

Matrix	Layout	NZ	Cl	cmods	Ut	ST	SF	DT	DF
bcsstk23	Min_Cost	154	63	9664	45.3	2.495	50.54	3.358	37.55
	Levels	175	72	10749	42.2	2.736	46.09	3.681	34.26
	C&S	167	49	11526	37.0	3.010	41.89	4.063	31.04
bcsstk24	Min_Cost	113	68	7101	35.5	1.388	28.11	1.837	21.24
	Levels	132	73	8085	30.8	1.556	25.07	2.068	18.87
	C&S	128	56	9522	23.1	1.876	20.79	2.508	15.56
bcsstk25	Min_Cost	567	276	38050	39.6	9.164	39.17	12.218	29.38
	Levels	603	290	47068	31.0	11.514	31.17	15.328	23.42
	C&S	650	242	50606	27.2	12.659	28.36	16.909	21.23
100-2D	Min_Cost	212	192	12057	16.2	2.506	8.19	3.159	6.49
	Levels	240	201	14519	12.1	2.973	6.90	3.793	5.41
	C&S	241	157	18255	7.9	3.796	5.40	4.905	4.18
16-3D	Min_Cost	196	79	12556	43.2	2.930	46.06	3.949	34.17
	Levels	199	76	14906	36.6	3.464	38.96	4.673	28.88
	C&S	219	64	16387	30.0	4.004	33.71	5.418	24.91
21-3D	Min_Cost	517	164	38710	53.7	10.740	68.47	14.750	49.85
	Levels	602	166	45400	45.8	12.701	57.90	17.374	42.32
	C&S	625	145	49010	40.0	14.135	52.02	19.396	37.91

Table 3: Results for 4K processor MasPar MP-2

On the MasPar MP-2 double precision arithmetic takes between two and three times as long as single precision. Therefore one could expect to more than double the Mflop rate in going from double precision to single precision. In our case the speedup is around 30%. We can use this difference in speedups to crudely estimate that the portion of time spent doing arithmetic out of the total computation time is about 38%.

Table 3 is similar to Table 2, except that the values are from running the program on a 64-by-64 section of the MP-2. We were not able to try out as large matrices as with the full machine. The reason for this is that each processor has to store more data since we have fewer processors. We also believe that there is some memory overhead in simulating a 64-by-64 PE array.

The data in Table 3 gives some indication of how well the algorithm scales with increased number of processors. In most cases we get between 1.5 and 2-fold speed-up. Although a 128-by-128 PE array has 4 times as many processors as a 64-by-64 one, it should be taken into account that the number of processor *columns* is only doubled, so the potential decrease in the number of simultaneous cmods is only by a factor of two. This can be seen in Table 1 comparing the values in *NZ64* and *NZ128* on one hand and the values in *LB64* and *LB128* on the other.

Finally, we note that the maximum number of nonzeros used with each of the different mapping schemes gets closer to the absolute minimum as the size of the matrices increase.

7 Conclusion

We have presented a new method of assigning the nonzeros of a sparse matrix to the processors of a SIMD computer in order to speed up Cholesky factorization of the matrix. It is assumed that the processors are arranged in a grid and communication along the grid lines is fast. Using this new mapping scheme, our implementation of sparse Cholesky factorization achieves a Mflop rate of over 200 on a powerful SIMD computer.

The main purpose of laying out the matrix onto the processors in this special way is to reduce the total number of cmod-operations that must be done. The method we present is a relatively simple greedy algorithm, which seems to get rather close to the lower bounds of the problems tested. It would be interesting to know if there exists an efficient algorithm to find an optimal layout in this respect.

Another way of getting potentially fewer cmods is to find an ordering that gives a lower elimination tree. One can expect that as the elimination tree gets lower, fewer related matrix columns will have to be assigned to the same processor column. The extreme case is when the height is less than 128 (the number of processor columns), then we would only need to do one cmod for each column. A case close to this extreme is shown in the test matrix 100-2D, which has an elimination tree of height 378, dimension 10,000, and the Min_Cost layout scheme gives only 10,396 cmods.

We have concentrated on minimizing the number of cmod operations, but have not been concerned with optimizing the computation of the cdiv operations. The reason is that the cdivs account for only around 1% of the total execution time according to our experiments. Thus there is very little to gain in optimizing them.

The optimality criteria used for mapping the matrix to the processor array is strongly influenced by the type of Cholesky factorization that we use. Other variants would probably do better with different layout schemes. We have experimented with other types of Cholesky factorization, in particular column-Cholesky (or fan-in Cholesky) where the matrix columns were ordered according to a postordering of the elimination tree and assigned to the processor columns in that order (i.e., not wrapped). In that way it was possible to modify simultaneously all the columns that were on the same level in the tree. This algorithm looked quite good on paper, but was nevertheless slower than the fan-out algorithm presented here. This result is related to speed differences between the various communication primitives provided by MasPar. On another SIMD computer the fan-in algorithm might be more competitive.

We have also tried some other mapping schemes that have proved successful in other circumstances, such as the randomized algorithm of Ogielski and Aiello [24], that they use with good results for sparse matrix-vector multiplication on a SIMD machine. For sparse Cholesky factorization it gave worse results than the simple Cut&Stack. We tried to improve the method by balancing the load on each processor based on the amount of work in each matrix column after the randomization [23]. This was slightly better, but still did not beat Cut&Stack.

While the numeric Cholesky factorization ran on a parallel computer in this implementation, all the preprocessing was done sequentially. On the larger test matrices this preprocessing, which included preordering and symbolic factorization in addition to the layout scheme, took considerably more time than the numeric factorization. However, these preliminary stages only have to be performed once for each sparse matrix structure. If we need to factor many matrices with the same structure, but different values, the time spent on preprocessing can be amortized over many numeric factorizations. This is the case in the Interior Point Method for the solution of linear programs [2][20].

All preprocessing stages might benefit from running on a parallel computer. The job of finding efficient parallel algorithms for these steps thus awaits further research.

The last step in solving a linear system via Cholesky factorization consists of solving two triangular systems. We have not tried to implement parallel algorithms for these operations. Thus we are not sure what effect the mapping scheme presented here has on this last solution step. The amount of work in the triangular solutions is much less than in the numeric factorization so it might be more difficult to get good processor utilization.

References

[1] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon, *Progress in sparse matrix methods for large linear systems on vector supercomputers*, Internat. J. Supercomp. Appl., 1 (1987), pp. 10–30.

[2] G. Astfalk, I. Lustig, R. Marsten, and D. Shanno, *The interior-point method for linear programming*, IEEE Software, (1992), pp. 61–68.

[3] E. R. Barnes, A. Vannelli, and J. Q. Walker, *A new heuristic for partitioning the nodes of a graph*, SIAM J. Discrete Math., 1 (1988), pp. 299–305.

[4] P. Bjørstad, F. Manne, T. Sørevik, and M. Vajteršic, *Efficient matrix multiplication on simd computers*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 386–401.

[5] T. Blank, *The MasPar MP-1 architecture*, in Proceedings of IEEE Compcon Spring 1990, IEEE, February 1990.

[6] A. Dave and I. Duff, *Sparse matrix calculations on the Cray-2*, Parallel Comput., 5 (1987), pp. 55–64.

[7] I. S. Duff, R. G. Grimes, and J. G. Lewis, *Users' guide for the Harwell-Boeing sparse matrix collection*, Tech. Report TR/PA/92/86, CERFACS, 1992.

[8] I. S. Duff and J. K. Reid, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Trans. Math. Software, 9 (1983), pp. 302–325.

[9] C. M. Fiduccia and R. M. Mattheyses, *A linear time heuristic for improving network partitions*, in Proceedings of the 19th Design Automation Conference, ACM/IEEE, 1982, pp. 175–181.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability*, Freeman, 1979.

[11] A. George, M. Heath, J. W. H. Liu, and E. G. Y. Ng, *Sparse Cholesky factorization on a local-memory multiprocessor*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 327–340.

[12] A. George and J. W. H. Liu, *An automatic nested dissection algorithm for irregular finite element problems*, SIAM J. Numer. Anal., 15 (1978), pp. 1053–1069.

[13] ——, *Computer Solutions of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[14] ——, *The evolution of the minimum degree ordering algorithm*, SIAM Rev., 31 (1989), pp. 1–19.

[15] J. R. Gilbert and R. Schreiber, *Highly parallel sparse Cholesky factorization*, SIAM J. Sci. Statist. Comput., 13 (1992), pp. 1151–1172.

[16] M. T. Heath, E. Ng, and B. Peyton, *Parallel algorithms for sparse linear systems*, SIAM Rev., 33 (1991), pp. 420–460.

[17] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell Systems Technical Journal, 49 (1970), pp. 291–307.

[18] S. G. Kratzer, *Sparse LU factorization on massively parallel SIMD computers*, Tech. Report SRC-TR-92-072, Supercomputing Research Center, 1992.

[19] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley, 1990.

[20] R. Levkovitz and G. Mitra, *Solution of large scale linear programs: A review of hardware, software, and algorithmic issues*, Tech. Report TR/06/92, Dept. of Mathematics and Statistics, Brunel University, 1992.

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

[22] ——, *The multifrontal method for sparse matrix solution: Theory and practice*, SIAM Rev., 34 (1992), pp. 82–109.

[23] F. Manne and T. Sørevik, *Optimal partitioning of sequences*, Tech. Report CS-92-62, University of Bergen, Norway, 1992.

[24] A. T. Ogielski and W. Aiello, *Sparse matrix computations on parallel processor arrays.* To appear in SIAM J. Sci. Statist. Comput., 1993.

[25] C. Sun, *Efficient parallel solutions of large sparse SPD systems on distributed-memory multiprocessors*, Tech. Report CTC92TR102, Cornell Theory Center, 1992.

[26] P. M. B. Vitányi, *How well can a graph be n-colored?*, Discrete Math., 34 (1981), pp. 69–80.

[27] C. Yang, *A vector/parallel implementation of the multifrontal method for sparse symmetric positive definite linear systems on the Cray Y/MP*, tech. report, Cray Research Inc., 1990.

[28] M. Yannakakis, *Computing the minimum fill-in is NP-complete*, SIAM J. Alg. Discrete Meth., 2 (1981), pp. 77–79.

[29] E. Zmijewski, *Sparse Cholesky Factorization on a Multiprocessor*, PhD thesis, Cornell University, 1987.