# HIGHLY PARALLEL SPARSE CHOLESKY FACTORIZATION

JOHN R. GILBERT* AND ROBERT SCHREIBER†

**Abstract.** We develop and compare several fine-grained parallel algorithms to compute the Cholesky factorization of a sparse matrix. Our experimental implementations are on the Connection Machine, a distributed-memory SIMD machine whose programming model conceptually supplies one processor per data element. In contrast to special-purpose algorithms in which the matrix structure conforms to the connection structure of the machine, our focus is on matrices with arbitrary sparsity structure.

The most promising alternative is a supernodal, multifrontal algorithm whose inner loop performs several dense factorizations simultaneously on a two-dimensional grid of processors. A fine-grained parallel, dense factorization algorithm is used as the key subroutine. The sparse code attains execution rates comparable to those of the dense subroutine. Although at present architectural limitations prevent the dense factorization from realizing its potential efficiency, we conclude that a regular data parallel architecture can be used efficiently to solve arbitrarily structured sparse problems.

We also present a performance model and use it to analyze our algorithms. We find that asymptotic analysis combined with experimental measurement of parameters is accurate enough to be useful in choosing among alternative algorithms for a complicated problem.

**Key words.** sparse matrix algorithms, Cholesky factorization, supernodal factorization, multifrontal factorization, systems of linear equations, parallel computing, data parallel algorithms, chordal graphs, clique trees, Connection Machine, performance analysis.

05C50, 15A23, 65F05, 65F50, 68M20.

## 1. Introduction.

### 1.1. Data parallelism.
Highly parallel, local memory computer architectures promise to achieve high performance inexpensively by assembling a large amount of simple hardware in a way that scales without bottlenecks. A data parallel programming model simplifies the programming of local memory parallel architectures by associating a processor with every data element in a computation (at least conceptually), thus hiding from the programmer the details of distribution of data and work to the processors.

Some major challenges come along with these promises. Communication is expensive relative to computation, so an algorithm must minimize communication. Locality is important in communication, so it pays to substitute communication with nearby processors for more general patterns where possible. The sequential programmer tunes the inner loop of an algorithm for high performance, but simple data parallel algorithms tend to have "everything in the inner loop" because a sequential loop over the data is typically replaced by a parallel operation. (From this point of view, the advantage of the more complicated of our two algorithms, Grid Cholesky, is that it removes the general-pattern communication from the inner loop.)

Algorithms for data parallel architectures must make different trade-offs than sequential algorithms: They must exploit regularity in the data. For efficiency on SIMD machines, they must also be highly regular in the time dimension. In some

* Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304. 

† Research Institute for Advanced Computer Science, MS T045-1, NASA Ames Research Center, Moffett Field, CA 94035. This author's work was supported by the Numerical Aerodynamic Simulation Systems Division of NASA and by DARPA via Cooperative Agreement NCC 2-387 between NASA and the Universities Space Research Association.

cases entirely new approaches may be appropriate; examples of experiments with such approaches include particle-in-box flow simulation, knowledge base maintenance [5], and the entire field of neural computation [20]. On the other hand, the same kind of regularity in a problem or an algorithm can often be exploited in a wide range of architectures; therefore, many ideas from sequential computation turn out to be surprisingly applicable in the highly parallel domain. For example, block-oriented matrix operations are useful in sequential machines with hierarchical storage and conventional vector supercomputers [3]; we shall see that they are also crucial to efficient data parallel matrix algorithms.

**1.2. Goals of this study.** Data parallel algorithms are natural for computations on matrices that are dense or have regular nonzero structures arising from, for example, regular finite difference discretizations. The main goal of this research is to determine whether data parallelism is useful in dealing with irregular, arbitrarily structured problems. Specifically, we consider computing the Cholesky factorization of an arbitrary sparse, symmetric, positive definite matrix. We will make no assumptions about the nonzero structure of the matrix besides symmetry. We will present evidence that arbitrary sparse problems can be solved nearly as efficiently as dense problems by carefully exploiting regularities in the nonzero structure of the triangular factor that come from the clique structure of its chordal graph.

A second goal is to perform a case study in analysis of parallel algorithms. The analysis of sequential algorithms and data structures is a mature and useful science that has contributed to sparse matrix computation for many years. By contrast, the study of complexity of parallel algorithms is in its infancy, and it remains to be seen how useful parallel complexity theory will be in designing efficient algorithms for real parallel machines. We will argue by example that, at least within a particular class of parallel architectures, asymptotic analysis combined with experimental measurement of parameters is accurate enough to be useful in choosing among alternative algorithms for a single fairly complicated problem.

**1.3. Outline.** The structure of the remainder of the paper is as follows. In Section 2 we review the definitions we need from numerical linear algebra and graph theory, sketch the architecture of the Connection Machine, and present a timing model for a generalized data parallel computer that abstracts that architecture.

In Section 3 we present the first of two parallel algorithms for sparse Cholesky factorization. The algorithm, which we call Router Cholesky, is based on a theoretically efficient algorithm in the PRAM model of parallel computation. We analyze the algorithm and point out two reasons that it fails to be practical, one having to do with communication and one with processor utilization.

In Section 4 we present a second algorithm, which we call Grid Cholesky. Grid Cholesky is a data parallel implementation of a supernodal, multifrontal method that draws on the ideas of Duff and Reid [7] and Ashcraft et al. [1]. It improves on Router Cholesky by using a two-dimensional grid of processors to operate on dense submatrices, thus replacing most of the slow generally-routed communication of Router Cholesky with faster grid communication. It also solves the processor utilization problem by assigning different data elements to the working processors at different stages of the computation. We present an analysis and experimental results for a pilot implementation of Grid Cholesky on the Connection Machine.

The pilot implementation of Grid Cholesky is approximately as efficient as a dense Cholesky factorization algorithm, but is still slow compared to the theoretical peak performance of the machine. Several steps necessary to improve the absolute efficiency of the algorithm, most of which concern efficient Cholesky factorization of dense matrices, are described. Finally we draw some conclusions and discuss avenues for further research.

**2. Definitions.** The first two subsections below summarize the linear algebra and graph theory needed to study sparse Cholesky factorization. Most of this material is covered in more detail by George and Liu [13, 23, 24]. The remainder of the section outlines the data parallel programming model and its implementation on the Connection Machine, and describes our parametric model of a data parallel computer.

**2.1. Linear algebra.** Let $A = (a_{ij})$ be an $n \times n$ real, symmetric, positive definite sparse matrix. There is a unique $n \times n$ lower triangular matrix $L = (l_{ij})$ with positive diagonal such that

$$A = LL^T .$$

This is the Cholesky factorization of $A$. We seek to compute $L$; with it we may solve the linear system $Ax = b$ by solving $Ly = b$ and $L^T x = y$. We will discuss algorithms for computing $L$ below. In general, $L$ is less sparse than $A$. The nonzero positions of $L$ that were zero in $A$ are called *fill* or *fill-in*. For any matrix $X$, we write $\eta(X)$ to denote the number of nonzero elements of $X$.

The rows and columns of $A$ may be symmetrically reordered so that the system solved is

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

where $P$ is a permutation matrix. We assume that such a reordering, chosen to reduce $\eta(L)$ and the number of operations required to compute $L$, has been done. We further assume that the structure of $L$ has been determined by a symbolic factoring process. We ignore these preliminary computations in this study because the cost of actually computing $L$ typically dominates. (In many cases, several identically structured matrices may be factored using the same ordering and symbolic factorization.) A study of the implementation of appropriate reordering and symbolic factorization procedures on data parallel architectures is in preparation [18].

If the matrix $A$ is such that its Cholesky factor $L$ has no more nonzeros than the lower triangle of $A$, i.e. there is no fill, then $A$ is a *perfect elimination matrix*. If $PAP^T$ is a perfect elimination matrix for some permutation matrix $P$, we call the ordering corresponding to $P$ a *perfect elimination ordering* of $A$.

Let $R$ and $S$ be subsets of $\{1, \ldots, n\}$. Then $A(R, S)$ is the $|R| \times |S|$ matrix whose elements are $a_{r,s}$ for $r \in R$ and $s \in S$. (For any set $S$, we write $|S|$ to denote its cardinality.)

**2.2. Graph theory.**

**2.2.1. Vertex elimination.** We associate two ordered, undirected graphs with the sparse, symmetric matrix $A$. First, $G(A)$, the graph of $A$, is the graph with vertices $\{1, 2, \ldots, n\}$ and edges

$$E(A) = \{(i, j) \mid a_{ij} \neq 0\} .$$

(Note that $E(A)$ is a set of unordered pairs.) Next, we define the *filled graph*, $G^*(A)$, with vertices $\{1, 2, \ldots, n\}$ and edges

$$E^*(A) = \{(i, j) \mid l_{ij} \neq 0\} ,$$

so that $G^*(A)$ is $G(L+L^T)$. The edges in $G^*(A)$ that are not edges of $G(A)$ are called *fill edges*. The output of a symbolic factorization of $A$ is a representation of $G^*(A)$.

For every fill edge $(i,j)$ in $E^*(A)$ there is a path in $G(A)$ from vertex $i$ to vertex $j$ whose vertices all have numbers lower than both $i$ and $j$; moreover, for every such path in $G(A)$ there is an edge in $G^*(A)$ [28]. Consider renumbering the vertices of $G^*(A)$. With another numbering, this last property may or may not hold. If it does, then the new ordering is a perfect elimination ordering of $G^*(A)$; Liu [24] calls it an *equivalent reordering* of $G^*(A)$.

**2.2.2. Chordal graphs.** Every cycle of more than three vertices in $G^*(A)$ has an edge between two nonconsecutive vertices (a *chord*). Such a graph is said to be *chordal*. Not only is $G^*(A)$ chordal for every $A$, but every chordal graph is the filled graph of some matrix [27].

Let $G = G(V,E)$ be any undirected graph. A *clique* is a subset $X$ of $V$ such that for all $u, v \in X$, $(u,v) \in E$. A clique is *maximal* if it is not a proper subset of any other clique. For any $v \in V$, the *neighborhood* of $v$, written $\mathrm{adj}(v)$, is the set $\{u \in V \mid (u,v) \in E\}$. The *monotone neighborhood* of $v$, written $\mathrm{madj}(v)$, is the smaller set $\{u \in V \mid u > v\,, (u,v) \in E\}$. We extend adj and madj to sets of vertices in the usual way.

A vertex $v$ is *simplicial* if $\mathrm{adj}(v)$ is a clique. Two vertices, $u$ and $v$, are *indistinguishable* if $\{u\} \cup \mathrm{adj}(u) = \{v\} \cup \mathrm{adj}(v)$. Two vertices are *independent* if there is no edge between them. A set of vertices is independent if every pair of vertices in it is independent; two sets $A$ and $B$ are independent if no vertex of $A$ is adjacent to a vertex of $B$.

It is immediate that any two simplicial vertices are either independent or indistinguishable. A set of indistinguishable simplicial vertices thus forms a clique, though not in general a maximal clique. The equivalence relation of indistinguishability partitions the simplicial vertices into pairwise independent cliques. We call these the *simplicial cliques* of the graph.

**2.2.3. Elimination trees.** A fundamental tool in studying sparse Gaussian elimination is the *elimination tree*. This structure was defined by Schreiber [30]; Liu [24] gives a survey of its many uses. Let $A$ have the Cholesky factor $L$. The elimination tree $T(A)$ is a rooted spanning forest of $G^*(A)$ defined as follows. If vertex $u$ has a higher-numbered neighbor $v$, then the parent $p(u)$ of $u$ in $T(A)$ is the smallest such neighbor; otherwise $u$ is a root. In other words, the first off-diagonal nonzero element in the $u^{\text{th}}$ column of $L$ is in row $p(u)$. It is easy to show that $T(A)$ is a forest consisting of one tree for each connected component of $G(A)$. For simplicity we shall assume in what follows that $A$ is irreducible, so that vertex $n$ is the only root, though our algorithms do not assume this.

There is a monotone increasing path in $T(A)$ from every vertex to the root. If $(u,v)$ is an edge of $G^*(A)$ with $u < v$ (that is, if $l_{vu} \neq 0$) then $v$ is on this unique path from $u$ to the root. This means that when $T(A)$ is considered as a spanning tree of $G^*(A)$, there are no "cross edges" joining vertices in different subtrees. It implies that, if we think of the vertices of $T(A)$ as columns of $A$ or $L$, any given column of $L$ depends only on columns that are its descendants in the tree.

**2.2.4. Clique trees.** The elimination tree describes a Cholesky factorization in terms of operations on single columns. A description in terms of operations on full blocks can yield algorithms with better locality of reference, which is an advantage either on a machine with a memory hierarchy (registers, cache, main memory, disk) or on a distributed-memory parallel machine. The Connection Machine falls into both of these categories.

Describing symmetric factorization in terms of operations on full submatrices is the key idea in both Duff and Reid's "multifrontal" algorithm [7] and the "supernodal" algorithm of Ashcraft, Grimes, Lewis, Peyton, and Simon [1], which can be traced back to the so-called element model of factorization [15, 33]. A full submatrix of $L$ is a clique in the chordal graph $G^*(A)$. The clique structure of chordal graphs has been explored extensively in the combinatorial literature; representations of chordal graphs as trees of cliques date back at least to 1972 [10] and continue to be used [16, 25, 26].

A *clique tree* for matrix $A$ is a tree whose nodes are sets that partition the vertices of $G^*(A)$ into cliques, in such a way that all the vertices of a node $N$ are indistinguishable simplicial vertices in the graph that results by deleting from $G^*(A)$ all vertices in nodes that are proper descendants of $N$. An equivalent definition is to think of starting with an elimination tree and collapsing vertices that are indistinguishable from their parents. (This definition differs slightly from that of Peyton [26], whose tree nodes are overlapping maximal cliques of $G^*(A)$.)

A clique which is a node in a clique tree is also called a *supernode*. If all vertices of $G^*(A)$ in proper descendants of some supernode are deleted, the supernode becomes a simplicial clique in the resulting graph. The clique tree is sometimes called a *supernode tree* or *supernodal elimination tree* [2]. A matrix may have many different clique trees—indeed, the elimination tree itself is one. Our numerical factorization algorithm Grid Cholesky can actually use any clique tree; the symbolic factorization we describe in Section 4.1 uses a block Jess and Kees algorithm to compute a shortest possible clique tree.

### 2.3. The Connection Machine.

**2.3.1. Architecture.** The Connection Machine (model CM-2) is a local memory, SIMD parallel computer. The description we present here corresponds to the machine architecture presented by the assembly language instruction set Paris [34]. We programmed the CM in *lisp, which is compiled into Paris.

A full-sized CM has $2^{16} = 65{,}536$ processors, each of which can directly access 65,536 bits of memory. (Since this work was done, larger memories have become available.) The processors are connected by a communication network called the *router*, which is configured by a combination of microcode and hardware to be a 16-dimensional hypercube.

The essential feature of the CM programming model is the *parallel variable* or pvar. A pvar is an array of data in which every processor stores and manipulates one element. The size of a pvar may be a multiple of the number of physical machine processors, $p$. If there are $V$ times as many elements in the pvar $X$ as there are processors, then (through microcode) each physical processor simulates $V$ virtual processors; thus the programmer's view remains "one processor per datum." The ratio $V$ is called the *virtual processor (VP) ratio*. The CM requires that $V$ must be a power of two. Thus we will find useful the notation $\lceil x \rceil$, meaning the smallest power of two not smaller than the real number $x$.

The *geometry* of each set of virtual processors (and its associated pvars) is also determined by the programmer, who may choose to view it as any multidimensional array with dimensions that are powers of two. The VP sets and their pvars are embedded in the machine (using Gray codes) so that neighboring virtual processors are simulated by the same or neighboring physical processors.

The Paris instruction set corresponds fairly well to the abstract model of data parallel programming that the CM attempts to present to the programmer, but it does not correspond closely to the actual hardware of the CM. Largely for this reason, it is hard to get high performance when programming in Paris or in a language that is compiled into Paris [31]. We shall go into this point in detail later. There are other ways to view and to program the hardware of the CM-2 that can provide better performance. These are just now becoming available to users, but were not when this work was done.

**2.3.2. Connection Machine programming.** Parallel computation on the CM is expressed through elementwise binary operations on pairs of pvars that reside in the same VP set—that is, have the same VP ratio and layout on the machine. (Optionally, one may specify a boolean mask that selects only certain virtual processors to be active.) These operations take time proportional to $V$, since the actual processors must loop over their simulated virtual processors. This remains true even when the set of selected processors is very sparse.

Interprocessor communication is expressed and accomplished in three ways, which we discuss in order of increasing generality but decreasing speed.

Communication with virtual processors at nearest neighbor grid cells is most efficient. A pvar may be shifted along any of its axes using this type of communication. The shift may be circular or end-off at the programmer's discretion.

A second communication primitive, *scan*, allows broadcast of data. For example, if $x$ is a one-dimensional pvar with the value [1, 2, 3, 4, 5, 6, 7, 8] then a scan of $x$ yields [1, 1, 1, 1, 1, 1, 1, 1]. Scans are implemented using the hypercube connections. The time for a scan of length $n$ is linear in $\log n$. Scans can also be used to broadcast along either rows or columns of a two-dimensional array. Scans that perform parallel prefix arithmetic operations are also available, but we do not use them.

Scans of subarrays are possible. In a *segmented scan*, the programmer specifies a boolean pvar, the *segment pvar*, congruent to $x$. The segments of $x$ between adjacent T values in the segment pvar are scanned independently. Thus, for example, if we use the segment pvar [T F F F T F F T] and $x$ is as above, then a segmented scan returns [1, 1, 1, 1, 5, 5, 5, 8].

The third and most general form of communication allows a virtual processor to access data in the memory of any other virtual processor. These operations go by several different names even within the CM environment; we shall refer to them in terms already familiar in sparse matrix computation: *gather* and *scatter*.

A gather allows processors to read data in the memory of other processors. The CM term for a gather is `pref!!` (for the *lisp programmer) or `get` (for the Paris programmer). In a gather, three pvars are involved: the source, the destination, and the address. The address of the processor whose memory is to be read is taken from the integer address pvar. Suppose the source is the one-dimensional pvar $[15, 14, 13, \ldots, 2, 1, 0]$ and the address pvar is $[0, 1, 2, 0, 1, 2, \ldots, 0, 1, 2, 0]$. Then the data stored in the destination is $[15, 14, 13, 15, 14, 13, \ldots, 15, 14, 13, 15]$. The Fortran-90 or Matlab statement that accomplishes this is is `dest = source(address)`; it performs the assignment $dest(i) \leftarrow source(address(i))$ for $1 \leq i \leq \text{length}(dest)$.

A scatter allows processors to write data to the memory of other processors. The CM term for a scatter is `*pset` (for the *lisp programmer) or `send` (for the Paris programmer). Again the three pvars are a source, a destination, and an integer address. The Fortran-90 or Matlab version is is `dest(address) = source`, and the effect is $dest(address(i)) \leftarrow source(i)$ for $1 \leq i \leq \text{length}(source)$.

Table 1
*Parameters of CM model*

| Connection Machine Parametric Model | | |
|---|---|---|
| Parameter | Description | Measured CM-2 value |
| $V$ | Virtual processor ratio | |
| $\mu$ | memory reference time | $4.8 \cdot V$ $\mu$sec |
| $\phi$ | Multiply or add time $\div \mu$ | 7 |
| $\sigma$ | Scan time $\div \phi$ | $6.2 + 1.2 \log_2$ scan-distance |
| $\nu$ | News time $\div \phi$ | 3 |
| $\rho$ | Route time $\div \phi$ | scatter (no collisions): 64<br>scatter-add (4 collisions): 110<br>scatter-add (100 collisions): 200<br>gather (many collisions): 430 |

In a scatter, when the address pvar has duplicate values, data from several source processors are sent to the same destination processor. (When this happens we say that a *collision* has occurred.) The programmer can select one of several different ways to combine colliding values by specifying a *combining operator*. For example, if the source and address are as above and the destination initially has the value $[1, 1, \ldots, 1]$ then after a scatter-with-add, the destination has the value $[45, 40, 35, 1, 1, 1, \ldots, 1]$. The sum of elements $source(j)$ such that $address(j) = k$ is stored in $dest(k)$ if there are any such elements; otherwise $dest(k)$ is unchanged. Other combining operators include product, maximum, minimum, and "error". We have found combining scatter to be a powerful aid to data parallel programming.

**2.3.3. Measured CM performance.** We will develop a model of performance on data parallel architectures and use it to analyze performance of several algorithms for sparse Cholesky factorization. The essential machine characteristics in the model are described by five parameters:

| | |
|---|---|
| $\mu$ | The memory reference time for a 32 bit word |
| $\phi$ | The 32 bit floating-point multiply time, in units of $\mu$ |
| $\nu$ | The 32 bit grid communication time, in units of $\phi$ |
| $\sigma$ | The 32 bit scan time, in units of $\phi$ |
| $\rho$ | The 32 bit router time, in units of $\phi$ |

(Floating-point addition takes the same time as multiplication, $\phi\mu$.) In our model, execution time scales linearly with VP ratio, which is essentially correct for the Connection Machine. Therefore $\mu$ is proportional to VP ratio, and the other parameters are independent of VP ratio. In Table 1, we give measured values for these parameters obtained by experiment on the CM-2. We observe that router times range over a factor of four depending on the number of collisions; it is possible to design pathological routing patterns that perform even worse than this. For any given pattern, gather usually takes just twice as long as scatter, presumably because it is implemented by sending a request and then sending a reply. In our approximate analyses, therefore, we generally choose a value of $\rho$ for scatter corresponding to the number of collisions observed, and model gather as taking $2\rho$ floating-point times.

**3. Router Cholesky.** Our first parallel Cholesky factorization algorithm is a parallel implementation of a standard column-oriented sparse Cholesky; it is based closely on Gilbert and Hafsteinsson's theoretically efficient algorithm [17] for the PRAM model of computation. Its communication requirements are too unstructured for it to be very efficient on a fine-grained multiprocessor like the CM, but we implemented and analyzed it to use as a basis for comparison and to help tune our performance model of the CM.

**3.1. The Router Cholesky algorithm.** Router Cholesky uses the elimination tree $T(A)$ to organize its computation. For the present, assume that both the tree and the symbolic factorization $G^*(A)$ are available. (In our experiments we computed the symbolic factorization sequentially; Gilbert and Hafsteinsson [17] describe a parallel algorithm.) Each vertex of the tree corresponds to a column of the matrix.

Following George et al. [12], we express a sequential column-oriented Cholesky factorization in terms of two primitive operations, *cdiv* and *cmod*:

**procedure** *Sequential Cholesky* (**matrix** $A$);
  **for** $j \leftarrow 1$ **to** $n$ **do**
    **for** each edge $(i, j)$ of $G^*(A)$ **with** $i < j$ **do**
      *cmod* $(j, i)$ **od**;
    *cdiv* $(j)$ **od**
**end** *Sequential Cholesky*;

Routine *cdiv* $(j)$ divides the subdiagonal elements of column $j$ by the square root of the diagonal element in that column, and routine *cmod* $(j, i)$ modifies column $j$ by subtracting a multiple of column $i$. This is called a *left-looking* algorithm because column $j$ accumulates all necessary updates *cmod* $(j, i)$ from columns to its left just before the *cdiv* $(j)$ that completes its computation. By contrast, a *right-looking* algorithm would perform all the updates *cmod* $(j, i)$ using column $i$ immediately after the *cdiv* $(i)$.

Now consider the elimination tree $T(A)$. A given column (vertex) is modified only by columns (vertices) that are its descendants in the tree [30]. Therefore a parallel left-looking algorithm can compute all the leaf vertex columns at once.

**procedure** *Router Cholesky* (**matrix** $A$);
  **for** $h \leftarrow 0$ **to** $height(n)$ **do**
    **for** each edge $(i, j)$ with $height(i) < height(j) = h$ **pardo**
      *cmod* $(j, i)$ **od**;
    **for** each vertex $j$ with $height(j) = h$ **pardo**
      *cdiv* $(j)$ **od**
    **od**
**end** *Router Cholesky*;

Here $height(j)$ is the length of the longest path in $T(A)$ from vertex $j$ to a leaf. Thus the leaves have height 0, vertices whose children are all leaves have height 1, and so forth. The outer loop of this algorithm works sequentially from the leaves of the elimination tree up to the root. At each step, an entire level's worth of *cmod*'s and *cdiv*'s are done.

A processor is assigned to every nonzero of the triangular factor (or, equivalently, to every edge and vertex of the filled graph $G^*$). Suppose processor $P_{ij}$ is assigned to the nonzero that is initially $a_{ij}$ and will eventually become $l_{ij}$. (If $l_{ij}$ is a fill, then $a_{ij}$ is initially zero; recall that we assume that the symbolic factorization is already done, so we know which $l_{ij}$ will be nonzero.) In the parallel *cdiv* $(j)$, processor $P_{jj}$ computes $l_{jj}$ as the square root of its element, and sends $l_{jj}$ to processors $P_{ij}$ for $i > j$, which then divide their own nonzeros by $l_{jj}$. In the parallel *cmod* $(j, i)$, processor $P_{ji}$ sends the multiplier $l_{ji}$ to the processors $P_{ki}$ with $k > j$. Each such $P_{ki}$ then computes the update $l_{ki}l_{ji}$ locally and sends it to $P_{kj}$ to be subtracted from $l_{kj}$.

We call this a *left-initiated* algorithm because the multiplications in *cmod* $(j, i)$ are performed by the processors in column $i$ who then, on their own initiative, send these updates to a processor in column $j$. Each column $i$ is involved in at most one *cmod* at a time because every column modifying $j$ is a descendant of $j$ in $T(A)$, and the subtrees rooted at vertices of any given height are disjoint. Therefore each processor participates in at most one *cmod* or *cdiv* at each parallel step. If we ignore the time taken by communication (including the time to combine updates to a single $P_{kj}$ that may come from different $P_{ki_1}$, $P_{ki_2}$, ...), then each parallel step takes a constant amount of time and the parallel algorithm runs in time proportional to the height of the elimination tree $T(A)$.

**3.2. CM implementation of Router Cholesky.** To implement Router Cholesky on the CM we must specify how to assign data to processors, and then describe how to do the communication in *cdiv* and *cmod*.

We use one (virtual) processor for each nonzero in the triangular factor $L$. We lay out the nonzeros in a one-dimensional array in column major order, as in many standard sequential sparse matrix algorithms [13]. This makes operations within a single column efficient because they can use the CM scan instructions. Each column is represented by a processor for its diagonal element followed by a processor for each sub-diagonal nonzero. The symmetric upper triangle is not stored. We can also think of this processor assignment as a processor for each vertex $j$ of the filled graph, followed by a processor for each edge $(i, j)$ with $i > j$.

Router Cholesky, like many data parallel algorithms, is profligate of parallel variable storage. Each processor contains some working storage and the following pvars:

| | |
|---|---|
| `l` | Element of factor matrix $L$, initially $A$. |
| `i` | Row number of this element. |
| `j` | Column number of this element. |
| `j_ht` | $height(j)$ in $T(A)$. |
| `i_ht` | $height(i)$ in $T(A)$. |
| `diagonal_p` | Boolean: Is this a diagonal element? |
| `e_parent` | In processor $P_{ij}$, a pointer to $P_{i,p(j)}$. |
| `next_update` | Pointer to next element this one may update. |

(Recall that $p(j) > j$ is the elimination tree parent of vertex $j < n$.)

At each stage of the sequential outer loop, each processor uses `i_ht` and `j_ht` to decide whether it participates in a *cdiv* or *cmod*. By comparing the local processor's `i_ht` or `j_ht` to the current value of the outer loop index, a processor can determine if its element is in a column that is involved in a *cdiv* or a *cmod*.

The *cdiv* uses a scan operation to copy the diagonal element to the rest of the active column.

The *cmod* uses a similar scan to copy the multiplier $l_{ji}$ down to the rest of column $i$. The actual update is done by a scatter-with-add, which uses the router to send the update to its destination.

To figure out where to send the update, each element maintains a pointer called `next_update` to a later element in its row. The nonzero positions in each row are a connected subgraph of the elimination tree, and are linked together in this tree structure by the `e_parent` pointers. Each nonzero updates only elements in columns that are its ancestors in the elimination tree. At each stage, `next_update` is moved one step up the tree using the `e_parent` pointers.

**3.3. Router Cholesky performance: Theory.** Each stage of Router Cholesky does a constant number of router operations, scans, and arithmetic operations. The number of stages is $h + 1$, where $h$ is the height of the elimination tree. In terms of the parameters of the machine model in Section 2.3.2, its running time is

$$(c_1\rho + c_2\sigma + c_3)\phi\mu h.$$

Recall that the memory reference time $\mu$ is proportional to the virtual processor ratio, which in this case is $\lceil \eta(L)/p \rceil$. The $c_i$ are constants.

The most time-consuming step is incrementing the `next-update` pointer; the router is used twice in this operation. The dominant term is the router term $c_1\rho\phi\mu h$. Notice that we do not explicitly count time for combining updates to the same element from different sources, since this is handled within the router and is thus included in $\rho$.

To get a feeling for this analysis consider the model problem, a 5-point finite difference mesh in two dimensions ordered by nested dissection [11]. If the mesh has $k$ points on a side, then the graph is a $k$ by $k$ square grid, and we have $n = k^2$, $h = O(k)$, and $\eta(L) = O(k^2 \log k)$. The number of arithmetic operations in the Cholesky factorization is $O(k^3)$, in either the sequential or parallel algorithms. Router Cholesky's running time is $O(\rho k^3 \log k/p)$. If we define performance as the ratio of the number of operations in the sequential algorithm to parallel time, we find that the performance is $O(p/\log k)$ (taking $\rho$ to be a constant independent of $p$ or $k$; this is approximately correct for the Connection Machine, although theoretically $\rho$ should grow at least with $\log p$). This analysis points out two weak points of Router Cholesky. First, the performance on the model problem drops with increasing problem size. (This depends on the problem, of course; for a three-dimensional model problem a similar analysis shows that performance is $O(p)$ regardless of problem size.) More seriously, the constant in the leading term of the complexity is proportional to the router time $\rho$, because every step uses general communication.

This analysis can be extended to any two-dimensional finite element mesh with bounded node degree, ordered by nested dissection [17]. The asymptotic analysis is the same but the values of the constants will be different.

**3.4. Router Cholesky performance: Experiments.** In order to validate the timing model and analysis, we experimented with Router Cholesky on a variety of sparse matrices. We present one example here in detail. The original matrix is $2500 \times 2500$ with 7400 nonzeros (counting symmetric entries only once), representing a $50 \times 50$ five-point square mesh. It is preordered by Sparspak's automatic nested dissection heuristic [13], which gives orderings very similar to the ideal nested dissection ordering used in the analysis of the model problem above. The Cholesky factor has $\eta(L) = 48,608$ nonzeros, an elimination tree of height $h = 144$, and takes 1,734,724 arithmetic operations to compute.

We ran this problem on CM-2's at the Xerox Palo Alto Research Center and the NASA Ames Research Center. The results quoted here are from $p = 8{,}192$ processors, with floating point coprocessors, of the machine at NASA. The VP ratio was therefore $V = \lceil \eta(L)/p \rceil = 8$. (Rounding up to a power of two has considerable cost here, since we use only 48,608 of the 65,536 virtual processors.) We observed a running time of 53 seconds, of which about 41 seconds was due to gathers and scatters. Substituting into the analysis above (using $\rho = 200$ since there were in general many collisions), we would predict router time $c_1\rho\phi\mu h = 39$ seconds and other time $(c_2\sigma + c_3)\phi\mu h = 1.5$ seconds. This is not a bad fit for router time; it is not clear why the remaining time is such a poor fit, but the expensive square root and the data movement involved in the pointer updates contribute to it, and it seems that I/O may have affected the measured 53 seconds.

The observation, in any case, is that router time completely dominates.

**3.5. Remarks on Router Cholesky.** Router Cholesky is too slow as it stands to be a cost-effective way to factor sparse matrices. Each stage does two gathers and a scatter with exactly the same communication pattern. More careful use of the router could probably speed it up by a factor of two to five. However, this would not be enough to make it practical; something more like a hundredfold improvement in router speed would be needed.

The one advantage of Router Cholesky is the extreme simplicity of its code. It is no more complicated than the numeric factorization routine of a conventional sequential sparse Cholesky package [13]. It is interesting to note that column-oriented sparse Cholesky codes on MIMD message-passing multiprocessors [12, 14, 35] are more complex. They exploit MIMD capability to implement dynamic scheduling of the *cmod* and *cdiv* tasks. They allow arbitrary assignment of columns to processors and therefore are required to use indirect addressing of columns. Finally, they are written with low-level communication primitives, the explicit "send" and "receive."

Router Cholesky's simplicity comes dearly. Flexibility in scheduling allows MIMD implementations to gain a modest performance advantage over any possible SIMD implementation. More important, we employ $\eta(L)$ virtual processors, regardless of the number of physical processors. It is essential that these virtual processors not all sit idle, consuming physical processor time slices, when there is nothing for them to do. As implemented by the Paris instruction set, they do sit idle.

We described Router Cholesky as a left-initiated, left-looking algorithm. In a right-initiated algorithm, processor $P_{ij}$ would perform the updates to $l_{ij}$. In a right-looking algorithm, updates would be applied as soon as the updating column of $L$ was computed instead of immediately before the updated column of $L$ was to be computed. Router Cholesky is thus one of four cousins. It is the only one of the four that maps operations to processors evenly; the other three alternatives require an inner sequential loop of some kind. All four versions require at least $h$ router operations.

**4. Grid Cholesky.** In this section we present a parallel supernodal, multifrontal Cholesky algorithm and its implementation on the CM. Multifrontal methods, introduced by Duff and Reid [7], compute a sparse Cholesky factorization by performing symmetric updates on a set of dense submatrices. We follow Liu [23] in referring to an algorithm that uses rank-1 updates "multifrontal" and the block version that uses rank-$k$ updates "supernodal multifrontal." The idea of using block methods to operate on supernodes has been used in many different sparse factorization algorithms [1, 7]. Parallel supernodal or multifrontal algorithms have been used on MIMD message-passing and shared-memory machines [2, 6, 32].

The algorithm uses a two-dimensional VP set (which we call the "playing field") to partially factor, in parallel, a number of dense principal submatrices of the partially factored matrix. By working on the playing field, we may use the fast grid and scan mechanisms for all the necessary communication during the factorization of the dense submatrices. Only when we need to move these dense submatrices back and forth to the playing field do we need to use the router. In this way we drastically reduce the use of the router: for the model problem on a $k \times k$ grid we reduce the number of uses from $h = 3k - 1$ to $2\log_2 k - 1$. The playing field can also operate at a lower VP ratio in general because it does not need to store the entire factored matrix at once.

This method, like all multifrontal methods, is in essence an "out of core" method in that the Cholesky factor is kept in a data structure that is not referred to while doing arithmetic, all of which is done on dense submatrices. The novelty here is the factorization of many of these dense problems in parallel; the simultaneous exploitation of the parallelism available *within* each of the dense factorizations; the use of a two-dimensional grid of processors for these parallel dense factorizations; the use of the machine's router for parallel transfers from the matrix storage data structure; and the use of the combining scatter operations for parallel update of the matrix storage data structure.

### 4.1. The Grid Cholesky algorithm.

**4.1.1. Block Jess and Kees reordering.** First we describe an equivalent reordering of the chordal graph $G^* = G^*(A)$ that we call the *block Jess/Kees ordering*. Block Jess/Kees is a perfect elimination ordering that has two properties that make it the best equivalent reordering for our purposes: It eliminates vertices with identical monotone neighborhoods consecutively, and it produces a clique tree of minimum height.

Our reordering eliminates all the simplicial vertices of $G^*$ simultaneously as one major step. In the process, it partitions all the vertices of $G^*$ into supernodes. Each of these supernodes is a clique in $G^*$, and is a simplicial clique when its component vertices are about to be eliminated. Each vertex is labeled with the *stage*, or major step number, at which it is eliminated. In more detail, the reordering algorithm is as follows.

> **procedure** *Reorder* **(graph** $G^*$**)**
>     $active\_stage \leftarrow -1$;
>     **while** $G^*$ is not empty **do**
>         $active\_stage \leftarrow active\_stage + 1$;
>         Number all the simplicial vertices in $G^*$, assigning
>             consecutive numbers within each supernode;
>         $stage(v) \leftarrow active\_stage$ for all simplicial vertices $v$;
>         Remove all the simplicial vertices from $G^*$ **od**;
>     $\hbar \leftarrow active\_stage$
> **end** *Reorder*

The cliques are the nodes of a clique tree whose height is $\hbar$, one less than the number of major elimination steps. The parent of a given clique is the lowest-stage clique adjacent to the given clique.

The name "block Jess/Kees" indicates a relationship with an algorithm due to Jess and Kees [21] that finds an equivalent reordering for a chordal graph so as to minimize the height of its elimination tree. The original (or "point") Jess/Kees ordering eliminates just one vertex from each simplicial clique at each major step. (This is a maximum-size independent set of simplicial vertices.) Each step of point Jess/Kees produces one level of an elimination tree, from the leaves up. The resulting elimination tree has minimum height over all perfect elimination orders on $G^*$. Our block Jess/Kees eliminates all the simplicial vertices at each major step, producing a clique tree one level at a time from the leaves up. This ordering may not minimize the height of the elimination tree. However, as Blair and Peyton [4] have shown, it does produce a clique tree of minimum height over all perfect elimination orders on $G^*$.

Every vertex is included in exactly one supernode. We number the supernodes as $\{S_1, \ldots, S_m\}$ in such a way that if $i < j$ then the vertices in $S_i$ have lower numbers than the vertices in $S_j$. The *stage* at which a supernode $S$ is eliminated is the iteration of the *while* loop at which its vertices are numbered and eliminated; thus, for all $v \in S$, $stage(v) = stage(S)$ is the height of node $S$ in the clique tree.

**4.1.2. Parallel supernodal multifrontal elimination.** Let $C$ be a supernode. It is immediate that $K = \mathrm{adj}(C) \cup C$ is a clique, and that it is maximal. Our factorization algorithm works by forming the principal submatrices of $A$ corresponding to vertices in the maximal cliques generated by supernodes. Let $\gamma = |C|$ and $\delta = |\operatorname{adj}(C)|$. Write $A(K,K)$ for the principal submatrix of order $|K| = \gamma + \delta$ consisting of elements $a_{i,j}$ with $i,j \in K$. It is natural to partition the principal submatrix $A(K,K)$ of $A$ as

$$A(K,K) = \begin{pmatrix} X & E \\ E^T & Y \end{pmatrix},$$

where $X = A(C,C)$ is $\gamma \times \gamma$, $Y = A(\operatorname{adj}(C), \operatorname{adj}(C))$ is $\delta \times \delta$, and $E$ is $\gamma \times \delta$.

In the terminology of the previous section, Grid Cholesky is a "block right-looking" algorithm. The details are as follows.

**procedure** *Grid Cholesky*(**matrix** $A$)
    **for** *active_stage* $\leftarrow 0$ **to** $\hbar$ **do**
        **for each** supernode $C$ with *stage(C)* = *active_stage* **pardo**
            Move $A(K,K)$ to the playing field,
                where $K = C \cup \operatorname{adj}(C)$;
            Set $Y$ to zero on the playing field;
            Perform $\gamma = |C|$ steps of parallel Gaussian elimination
                to compute the Cholesky factor $L$ of $X$,
                the updated submatrix $E' = L^{-1}E$,
                and the Schur complement $Y' = -E^T X^{-1} E$,
                where $X$, $E$, and $Y$ partition $A(K,K)$ as above;
            $A(C,C) \leftarrow L$;
            $A(\operatorname{adj}(C), C) \leftarrow E'^T$;
            $A(\operatorname{adj}(C), \operatorname{adj}(C)) \leftarrow A(\operatorname{adj}(C), \operatorname{adj}(C)) + Y'$ **od**
        **od**
**end** *Grid Cholesky*;

**4.2. Multiple dense partial factorization.** In order to make this approach useful, we need a fast dense matrix factorizations on two-dimensional VP sets. To that end, we discuss an implementation of $LU$ decomposition without pivoting. (We use $LU$ instead of Cholesky here because we can see no efficient way to exploit symmetry with a two-dimensional machine; moreover, $LU$ avoids the square root at each step and so is a bit faster.) We analyzed and implemented two methods: a systolic algorithm that uses only nearest neighbor communication on the grid, and a rank-1 update algorithm that uses row and column broadcast. With either of these methods, all the submatrices $A(K,K)$ corresponding to supernodes at a given stage are distributed about the two-dimensional playing field simultaneously (each as a separate "baseball diamond"), and the partial factorization is applied to all the submatrices at once.

We describe the rank-1 algorithm in terms of its effect on a single submatrix $A(K,K)$, with a supernode of size $\gamma$ and a Schur complement of size $\delta$. A single rank-1 update consists of a division to compute the reciprocal of the diagonal element, a scan down the columns to copy the pivot row to the remaining rows, a parallel multiplication to compute the multiplier for each row, another scan to copy the multiplier across each row, and finally a parallel multiply and subtract to apply the update. The number of rank-1 updates is $\gamma$, the size of the supernode.

An entire stage of partial factorizations is performed at once, using segmented scans to keep each factorization within its own "diamond." The number of steps on the playing field at stage $s$ is $\gamma_s$, the maximum value of $\gamma$ over all supernodes at stage $s$. Then a stage of rank-1 partial factorization takes time

$$(c_3\sigma + c_4)\phi\mu.$$

Here $c_3$ is $2\gamma_s$, and $c_4$ is proportional to $\gamma_s$ as well.

The relative cost of the various parts of the rank-1 update code are summarized below, for a complete factorization (that is, one in which $\delta = 0$). The bookkeeping includes nearest-neighbor communication to move three one-bit tokens that control which processors perform reciprocals, multiplications, and so on at each step.

| | |
|---|---|
| Scans (row and column broadcast): | 79.7% |
| News (moving the tokens): | 5.5% |
| Multiply (computing multipliers): | 2.7% |
| Divide (reciprocal of pivot element): | 7.1% |
| Multiply-subtract (Gaussian elimination): | 4.8% |

Unlike the rank-1 implementation, the systolic implementation never uses a scan; all communication is between grid neighbors. Thus its communication terms are proportional to $\nu$ rather than $\sigma$. This advantage is more than offset by the fact that $3\gamma_s + 2\delta_s$ sequential iterations are necessary, while the rank-1 method only needs $\gamma_s$.

**4.2.1. Remarks on dense partial factorization.** Theoretically, systolic factorization should be asymptotically more efficient as machine size and problem size grow without bound, because scans must become more expensive as the machine grows. Realistically, however, the CM happens to have $\sigma \approx 6\nu$; for a full factorization ($\delta = 0$) a sixfold decrease in communication time per step more than balances the threefold increase in number of steps, and so the systolic method *is* somewhat faster. But for a partial factorization the rank-1 algorithm is the clear winner. For example, for the two-dimensional model problem the average Schur complement size $\delta_s$ is about $4\gamma_s$, so the rank-1 code has an 11-to-1 advantage in number of steps. This more than makes up for the fact that scan is much slower than grid communication.

It is interesting to note that the only arithmetic that matters in a sequential algorithm, the multiply-subtract, accounts for only 1/21 of the total time in the rank-1 parallel algorithm. Moreover, only 1/3 of the multiply-subtract operations actually do useful work, since the active part of the matrix occupies a smaller and smaller subset of the playing field as the computation progresses. This gives the code an overall efficiency of one part in 63 for $LU$ and one part in 126 for Cholesky. We have found this to be typical in *lisp codes for matrix operations, especially with high VP ratios. The reasons are these: scan is slow relative to arithmetic; the divide and multiply operations occur on very sparse VP sets; and the VP ratio remains constant as the active part of the matrix gets smaller.

Significant performance improvements could come from several possible sources.

A more efficient implementation of virtual processors could improve performance substantially: the loop over $V$ virtual processors should be restricted to those virtual processors that are active. Sometimes a determination that this is going to happen can readily be made at compile time. As an alternative, we could have rewritten the code; the VP set could shrink as the matrix shrinks, and the divide and the multiplies could be performed in a sparser VP set.

The scans could be sped up considerably within the hypercube connection structure of the CM. Ho and Johnsson [19] have developed an ingenious algorithm that takes $O(b/d + d)$ time to broadcast $b$ bits in a $d$-dimensional hypercube, in contrast to the scan which takes $O(b + d)$. The copy scans could also be implemented so that each physical processor gets only one copy of the data rather than $V$ copies.

More efficient use of the low-level floating-point architecture of the CM-2 is possible. The Paris instruction set hides the fact that every 32 physical processors share one vector floating-point arithmetic chip. Performing 32 floating point operations implies moving 32 numbers in bit-serial fashion into a transposer chip, then moving them in parallel into the vector chip, then reversing the process to store the result. While this mode of operation conforms to the one-processor-per-data-element programming model, it wastes time and memory bandwidth when only a few processors are actually active (such as computing multipliers or diagonal reciprocals in $LU$), since 32 memory cycles are required just to access one floating-point number.

This mode also requires intermediate results to be stored back to main memory, precluding the use of block algorithms [3] that could store intermediate results in the registers in the transposer. Thus the computation rate is limited by the bandwidth between the transposer and memory (about 3.5 gigaflops for a 64K processor CM) instead of by the operation rate of the vector chip (about 27 gigaflops total).

A more efficient dense matrix factorization can be achieved by treating each 32-processor-plus-transposer-and-vector-chip unit as a single processor, and representing 32-bit floating-point numbers "slicewise," with one bit per physical processor, so that they do not need to be moved bit-serially into the arithmetic unit. (Viewed this way, we were working with just 256 real processors!) Also, if virtualization is handled efficiently, we need only keep 256 processors busy. At the time this work was done (late in 1988) the tools for programming on this level were not easily usable. While CM Fortran allows this model, it does not allow scans and scatter with combining, so we still (early 1991) cannot use it.

**4.3. CM implementation of Grid Cholesky.** Grid Cholesky uses two VP sets with different geometries: the *matrix storage* stores the nonzero elements of $A$ and $L$ (doing almost no computation), and the *playing field* is where the dense partial factorizations are done. The top-level factorization procedure is just a loop that moves the active submatrices to the playing field, factors them, and moves updates back to the main matrix.

**4.3.1. Matrix storage.** The matrix storage VP set is a one-dimensional array of virtual processors that stores all of $A$ and $L$ in essentially the same form as Router Cholesky. Each of the following pvars has one element for each nonzero in $L$.

| | |
|---|---|
| `l_value` | Elements of $L$, initially those of $A$. |
| `grid_i` | The playing field row in which this element sits. |
| `grid_j` | The playing field column in which this element sits. |
| `active_stage` | The stage at which j occurs in a supernode. |
| `updates` | Working storage for sum of incoming updates. |

We use a scatter to move the active columns from matrix storage to the playing field. The supernodes $C$ are disjoint, but their neighboring sets $\mathrm{adj}(C)$ may overlap; that is, more than one $Y_C$ may be computing updates to the same element of $L$ at the same stage. Therefore, we use scatter-with-add to move the partially factored matrix from the playing field back to matrix storage.

**4.3.2. The playing field.** The second VP set, called the playing field, is the two-dimensional grid on which the supernodes are factored. In our implementation it is large enough to hold all the principal submatrices for all maximal cliques at any stage, although it could actually use different VP ratios at different stages for more efficiency. Its size is determined as part of the symbolic factorization and reordering. The pvars used in this VP set are

`dense_a` The playing field for matrix elements.

`update_dest` The matrix storage location (processor) that holds this matrix element; an integer pvar array indexed by stage.

as well as some boolean flags used to coordinate the simultaneous partial factorization of all the maximal cliques.

The processors of the playing field compute $LU$ factorizations by simultaneously doing rank-1 updates (see Section Section 4.2) of all the dense submatrices stored there, using segmented scans to distribute the pivot rows and columns within all submatrices at the same time. The number of rank-1 update steps is the size of the largest supernode at the current stage. The submatrices may be different sizes; each matrix only does as many rank-1 updates as the size of its supernode.

In order to use this procedure we need to find a placement of all the submatrices $A(K, K)$ for all the maximal cliques $K$ at every stage. This is a two-dimensional bin packing problem. In order to minimize CM computation time, we want to pack these square arrays onto the smallest possible rectangular playing field (whose borders must be powers of two). Optimal two-dimensional bin packing is in general an NP-hard problem, though various efficient heuristics exist [9]. Our experiments use a simple "first fit by levels" heuristic. This layout is done during the sequential symbolic factorization, before the numeric factorization is begun.

**4.4. Grid Cholesky performance: Theory.** We separate the running time for Grid Cholesky into time in the matrix storage VP set and time on the playing field. The former includes all the router traffic, and essentially nothing else. (There is one addition per stage to add the accumulated updates to the matrix.) There is a fixed number of router uses per stage, so the matrix storage time is

$$T_{MS} = c_5 \rho \phi \mu_{MS} \hbar$$

for some constant $c_5$. The subscript $MS$ indicates that the value of $\mu$ is taken in the matrix storage VP set, whose VP ratio set is $V_{MS} = \lceil \eta(L)/p \rceil$. In the current implementation $c_5 = 4$, since two scatters are used to move the dense matrices to the playing field at the beginning of a stage, and then two scatters are used to move the completely computed columns and the Schur complements back to matrix storage.

We express the playing field time as a sum over levels. At each level $s$ the number of rank-1 updates is the size of the largest supernode at that level, which is $\gamma_s$. According to the analysis in Section 4.2,

$$T_{PF} = (c_6 \sigma + c_7) \sum_{s=0}^{\hbar} \gamma_s \phi \mu_s ,$$

where $c_6$ and $c_7$ are constants (in fact $c_6 = 2$), and the subscript $s$ indicates that the value of $\mu$ is taken in the playing field VP set at stage $s$. The VP ratio in this VP set could be approximately the ratio of the total size of the dense submatrices at

Table 2
*Subproblem counts and playing field size for the model problem.*

| Stage $s$ | $R(s)$ | $\gamma_s$ | $\gamma_s + \delta_s$ | $\sum(\gamma_C + \delta_C)^2$ |
|---|---|---|---|---|
| $\hbar$ | 1 | $k$ | $k$ | $k^2$ |
| $\hbar - 1$ | 2 | $k/2$ | $3k/2$ | $4.5k^2$ |
| $\hbar - 2$ | 4 | $k/2$ | $3k/2$ | $9k^2$ |
| $\hbar - 3$ | 8 | $k/4$ | $3k/2$ | $18k^2$ |
| $\hbar - 4$ | 16 | $k/4$ | $5k/4$ | $25k^2$ |
| $\hbar - 5$ | 32 | $k/8$ | $7k/8$ | $24.5k^2$ |
| $\hbar - 6$ | 64 | $k/8$ | $5k/8$ | $25k^2$ |
| $\hbar - 7$ | 128 | $k/16$ | $7k/16$ | $24.5k^2$ |
| $\hbar - 2r$ | $2^{2r}$ | $k/2^r$ | $5k/2^r$ | $25k^2$ |
| $\hbar - 2r - 1$ | $2^{2r+1}$ | $k/2^{r+1}$ | $7k/2^{r+1}$ | $24.5k^2$ |

stage $s$ to the number of processors, changing at each stage as the number and size of the maximal cliques vary. However in our implementation it is simply fixed at the maximum of this value over all stages.

Again, to get a feeling for this analysis let us consider the model problem, a five-point finite difference mesh on a $k \times k$ grid ordered by nested dissection. For this problem $n = k^2$, $\hbar = O(\log k)$, and $\eta(L) = O(k^2 \log k)$. The factorization requires $O(k^3)$ arithmetic operations. Table 2 summarizes the number and sizes of the cliques that occur at each stage. The columns in the table are as follows.

| | |
|---|---|
| $R(s)$ | Number of supernodes at stage $s$. |
| $\gamma_s$ | Size of largest supernode at stage $s$. |
| $\gamma_s + \delta_s$ | Size of largest maximal clique $C \cup \mathrm{adj}(C)$ at stage $s$. |
| $\sum(\gamma_C + \delta_C)^2$ | Total area of all dense submatrices $A(K, K)$ at stage $s$. |

The VP ratio in matrix storage for the model problem is $O(\eta(L))/p = O(k^2 \log k/p)$, so the matrix storage time is $O(k^2 \log^2 k/p)$. Our pilot implementation uses the same size playing field at every stage. According to Table 2, a playing field of size $\lceil 25k^2 \rceil$ suffices if the problems can be packed in without too much loss. The VP ratio is $O(k^2/p)$. The sum over all stages of $\gamma_s$ is $O(k)$ (in fact it is $3k + O(1)$), so the playing field time is $O(k^3/p)$. In summary, the total running time of Grid Cholesky for the model problem is

$$O\left(\frac{k^2 \log^2 k}{p}\rho + \frac{k^3}{p}\sigma\right).$$

Two things are notable about this: First, the performance, or ratio of sequential arithmetic operations to time, is $O(p)$; the $\log k$ inefficiency of Router Cholesky has vanished. This is because the playing field, where the bulk of the computation is done, has a lower VP ratio than the matrix storage structure. Second, and much more important in practice, the router speed $\rho$ appears only in the second-order term. This is because the playing field computations are done on dense matrices with more efficient grid communication. This means that the router time becomes less important as the problem size grows, whether or not the number of processors grows with the problem size.

One way of looking at this analysis is to think of increasing both problem size and machine size so that the VP ratio remains constant. Then the model problem requires $O(k)$ total parallel operations, but only $O(\log k)$ router operations. The analysis of the model problem carries through (with different constant factors) for any two-dimensional finite element problem ordered by nested dissection; a similar analysis carries through for any three-dimensional finite element problem. Thus, Grid Cholesky is a "scalable" parallel algorithm.

**4.5. Grid Cholesky performance: Experiments.** Here we present experimental results from a fairly small model problem, the matrix arising from the five-point discretization of the Laplacian on a square $63 \times 63$ mesh, ordered by nested dissection. This matrix has $n = 3{,}969$ columns and 11,781 nonzeros (counting symmetric entries only once). The Cholesky factor has $\eta(L) = 86{,}408$ nonzeros, a clique tree with $\bar{h} = 11$ stages of supernodes, and takes 3,589,202 arithmetic operations to compute.

The matrix storage VP set requires 128K VPs. The fixed-size playing field requires $256 \times 512$ VPs. The results quoted here are from 8,192 processors, with floating-point coprocessors, of the machine at NASA. Both VP sets therefore had a VP ratio of 16. (A larger problem would need a higher VP ratio in the matrix storage than in the playing field.)

We observed a running time of 6.13 seconds. Of this, 4.09 seconds was playing field time (3.12 for the scans, 0.15 for nearest-neighbor moves of one-bit tokens, and 0.82 for local computation). The other 2.04 seconds was matrix storage time, consisting mostly of the four scatters at each stage. Our analytic model predicts playing field time to be about $3k \cdot (2\sigma + 4)\phi\mu_{PF}$. This comes to 4.0 seconds, which is in good agreement with experiment. The model predicts matrix storage time of about $\bar{h} \cdot 4\rho\phi\mu_{MS}$. This comes to between 1.5 and 4.7 seconds, depending on which value we choose for $\rho$. In fact 3/4 of the router operations are scatters with no collisions, and the other 1/4 are scatter-with-add, typically with two to four collisions. The fit to the model is therefore quite close.

**4.6. Remarks on Grid Cholesky.** On this small example , Grid Cholesky is about 20 times as fast as Router Cholesky. It is, however, only running at .586 megaflops on 8K processors, which would scale to 4.68 megaflops on a full 64K machine. A larger problem would run somewhat faster, but it is clear that making Grid Cholesky attractive will require major improvements. Some of these were sketched in Section 4.2.1.

A first question is whether Grid Cholesky is a router-bound code like Router Cholesky. For the small sample problem the relative times for router and non-router computations are as follows.

| | |
|---|---|
| Moving data to the playing field: | 12% |
| Factoring on the playing field: | 67% |
| Moving Schur complements back to matrix storage: | 21% |

Evidently, the Grid Cholesky code is not router-bound for this problem. For larger (or structurally denser) problems this situation gets better still: for a machine of fixed size, the time spent using the router grows like $O(\log^2 k)$ while the time on the playing field grows like $O(k^3)$ for a $k \times k$ grid, as we showed above. If we solved the same problem on a full-sized 64K processor machine, the relative times would presumably be the same as above; but if we solved a problem 8 times as large the operation count would increase by a factor of about 22 while the number of stages, and router operations, would increase only by a factor of about 1.3.

Next, we ask whether our use of the playing field is efficient or not. The number of parallel elimination steps on the playing field is given by

$$\sum_{s=1}^{\hbar} \gamma_s ,$$

which is 177 for the example. On the playing field of 131,072 processors this allows us to do $22.8 \times 10^6$ multiply-adds or $45.6 \times 10^6$ flops. The number of flops actually done is is $3.69 \times 10^6$, so the processor utilization is 7.9%. There are several reasons for this loss of efficiency:

- The algorithm does both halves of the symmetric dense subproblems (factor of 2);
- the implementation uses the same playing field size at every level (factor of about 4/3);
- the architecture forces the dimensions of the playing field to be powers of two (factor of about 4/3);
- the playing field is not fully covered by active processors initially, and as the dense factorization progresses processors in the supernodes fall idle (factor of about 7/2).

These effects account for a factor of roughly 12.4, which is consistent with our measured result; $1/.079 = 12.6$. In other words, every step must use all 131,072 virtual processors, but on average we only have work for about 10,500 of them. Thus, the Paris virtualization method cost us a factor of roughly $(131,072/10,500) = 12.5$.

A similar analysis shows that virtualization slows the use of the router by a factor of 7.6.

We summarize the reasons that our achieved performance is so far below the CM's peak:

- Virtualization. We used $2^{17}$ virtual processors. On average, we make use of 10,500 on the playing field and 5,600 in matrix storage. In actuality, there are 256 physical (floating-point) processors in the machine we used.
- The slow router.
- Communication costs for scans on the playing field, using the built-in suboptimal algorithm.
- The SIMD restriction. This causes us to have to wait for the divides and multiplies. (Since there are very few active virtual processors during these tasks, most of this effect could also be removed by efficient virtualization).
- The Paris instruction set, which makes reuse of data in registers impossible, thus exposing the low memory bandwidth of the CM.

Table 3 gives an upper bound on the improvement that removal of each of these impediments to performance would provide. In each case, we hypothesize an "ideal" machine in which the corresponding cost is completely removed. Thus, for example, the statistics for the third row of the table are for a marvelous machine in which a router operation takes no time whatever.

Table 3
*Factors affecting efficiency of Grid Cholesky.*

| Impediment removed | Time | Mflops | Speedup |
|---|---|---|---|
| None | 6.13 | 4.8 | |
| Virtualization | 0.60 | 49. | 10. |
| Slow router | 0.33 | 89. | 1.8 |
| Slow scans | 0.065 | 450. | 5.0 |
| SIMD | 0.021 | 1,400. | 3.1 |
| Paris | 0.0042 | 7,000. | 5.0 |

Clearly, good efficiency is possible, even on an SIMD machine with a router. The chief problems are the Paris virtualization method; the lack of a fast broadcast in the grid; and the memory-to-memory instruction set. All of these are peculiarities of Paris on the CM-2; none is fundamental to SIMD, data parallel computing.

**5. Conclusions.** We have compared two highly parallel general-purpose sparse Cholesky factorization algorithms, implemented for a data parallel computer. The first, Router Cholesky, is concise and elegant and takes advantage of the parallelism present in the elimination tree, but because it pays little attention to the cost of communication it is not practical.

We therefore developed a parallel algorithm, Grid Cholesky, that does most of its work with grid communication on dense submatrices. Analysis shows that the requirement for expensive general-purpose communication grows only logarithmically with problem size; even for modest problems the code is not limited by the CM router. Experiment shows that Grid Cholesky is about 20 times as fast as Router Cholesky for a moderately small sample problem.

Our pilot implementation of Grid Cholesky is not fast enough to make the Connection Machine cost-effective for solving generally structured sparse matrix problems. We believe, however, that our experiments and analysis lead to the conclusion that a parallel supernodal, multifrontal algorithm can be made to perform efficiently on a highly parallel SIMD machine. We showed in detail that Grid Cholesky could run two to three orders of magnitude faster with improvements in the instruction set of the Connection Machine. of our pilot implementation from the 27-gigaflop theoretical peak performance of a 64K processor CM yawn somewhat less dauntingly.

Most of these improvements are below the level of the Paris virtual processor abstraction, which is to say below the level of the assembly-language architecture of the machine. Although TMC has recently released a low-level language called CMIS in which a user can program below the virtual-processor level, we believe that ultimately most of these optimizations should be applied by high-level language compilers. Future compilers for highly parallel machines, while they will support the data-parallel virtual processor abstraction at the user's level, will generate code at a level below that abstraction.

Although Grid Cholesky is more complicated than Router Cholesky, we are still able to use the data parallel programming paradigm to express it in a straightforward way. The high-level scan and scatter-with-add communication primitives substantially simplified the programming. The simplicity of our codes speaks well for this data parallel programming model.

In summary, even though our pilot implementation is not fast, we are nonetheless encouraged — about the Grid Cholesky algorithm and about the potential of data parallel programming and highly parallel architectures for solving unstructured problems. We believe that future generations of highly parallel machines may allow efficient parallel programs for complex tasks to be written nearly as easily as sequential programs. To get to that point, there will have to be improvements in compilers, instruction sets, and router technology. Virtualization will have to be implemented without sacrificing efficiency.

We mention four avenues for further research.

The first is scheduling the dense partial factorizations efficiently. The tree of supernodes identifies a precedence relationship among the various partial factorizations. Our simple approach of scheduling these one level at a time onto a fixed-size playing field is not the only possible one. There is in general no need to perform all the partial factorizations at a single level simultaneously. It should be possible to use more sophisticated heuristics to schedule these factorizations onto a playing field of varying VP ratio, or even (for the Connection Machine) onto a playing field considered as a mesh of individual vector floating-point chips. Some theoretical work has been done on scheduling "rectangular" tasks onto a square grid of processors efficiently [8].

The second avenue is improving the time spent in the matrix storage VP set. Of course, as problems get larger this time becomes a smaller fraction of the total. At present matrix storage time is not very significant even for a small problem, but it will become more so as the playing field time is improved.

Third, we mention the possibility of an out-of-main-memory version of Grid Cholesky for very large problems. Here the clique tree would be used to schedule transfers of data between the high-speed parallel disk array connected to the CM and the processors themselves.

Fourth and finally, we mention the possibility of performing the combinatorial preliminaries to the numerical factorization in parallel. Our pilot implementation uses a sequentially generated ordering, symbolic factorization, and clique tree. We are currently designing data parallel algorithms to do these three steps [18].

We conclude by extracting one last moral from Grid Cholesky. We find it interesting and encouraging that the key idea of the algorithm, namely partitioning the matrix into dense submatrices in a systematic way, has also been used to make sparse Cholesky factorization more efficient on vector supercomputers [32] and even on workstations [29]. In the former case, the dense submatrices vectorize efficiently; in the latter, the dense submatrices are carefully blocked to minimize traffic between cache memory and main memory. We expect that more experience will show that many techniques for attaining efficiency on sequential machines with hierarchical storage will turn out to be useful for highly parallel machines.

## REFERENCES

[1] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon, *Recent progress in sparse matrix methods for large linear systems*, International Journal of Supercomputer Applications, (1987), pp. 10–30.

[2] C. C. Ashcraft, *The domain/segment partition for the factorization of sparse symmetric positive definite matrices*, Tech. Report ECA–TR–148, Boeing Computer Services Engineering Computing and Analysis Division, Seattle, 1990.

[3] C. H. Bischof and J. J. Dongarra, *A project for developing a linear algebra library for high-performance computers*, Tech. Report MCS–P105–0989, Argonne National Laboratory, 1989.
[4] J. R. S. Blair and B. W. Peyton, *On finding minimum-diameter clique trees*, Tech. Report ORNL/TM–11850, Oak Ridge National Laboratory, 1991.
[5] M. Dixon and J. de Kleer, *Massively parallel assumption-based truth maintenance*, in Proceedings of the National Conference on Artificial Intelligence, 1988, pp. 199–204.
[6] I. S. Duff, *Multiprocessing a sparse matrix code on the Alliant FX/8*, Tech. Report CSS 210, Computer Science and Systems Division, AERE Harwell, 1988.
[7] I. S. Duff and J. K. Reid, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Transactions on Mathematical Software, 9 (1983), pp. 302–325.
[8] A. Feldmann, J. Sgall, and S.-H. Teng, *Dynamic scheduling on parallel machines*.
[9] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.
[10] F. Gavril, *Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph*, SIAM Journal on Computing, 1 (1972), pp. 180–187.
[11] A. George, *Nested dissection of a regular finite element mesh*, SIAM Journal on Numerical Analysis, 10 (1973), pp. 345–363.
[12] A. George, M. T. Heath, J. Liu, and E. Ng, *Sparse Cholesky factorization on a local-memory multiprocessor*, SIAM Journal on Scientific and Statistical Computing, 9 (1988), pp. 327–340.
[13] A. George and J. W. H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.
[14] A. George and E. Ng, *On the complexity of sparse QR and LU factorization of finite-element matrices*, SIAM Journal on Scientific and Statistical Computing, 9 (1988), pp. 849–861.
[15] J. A. George and D. McIntyre, *On the application of the minimum degree algorithm to finite element systems*, SIAM Journal on Numerical Analysis, 15 (1978), pp. 90–112.
[16] J. R. Gilbert, *Some nested dissection order is nearly optimal*, Information Processing Letters, 26 (1988), pp. 325–328.
[17] J. R. Gilbert and H. Hafsteinsson, *Parallel solution of sparse linear systems*, in SWAT 88: Proceedings of the First Scandinavian Workshop on Algorithm Theory, Springer-Verlag Lecture Notes in Computer Science 318, 1988, pp. 145–153.
[18] J. R. Gilbert, C. Lewis, and R. Schreiber, *Parallel preordering for sparse matrix factorization*. In preparation.
[19] C.-T. Ho and S. L. Johnsson, *Spanning balanced trees in Boolean cubes*, SIAM Journal on Scientific and Statistical Computing, 10 (1989), pp. 607–630.
[20] J. J. Hopfield, *Neural networks and physical systems with emergent collective computational abilities*, Proceedings of the National Academy of Science, 79 (1982), pp. 2554–2558.
[21] J. A. G. Jess and H. G. M. Kees, *A data structure for parallel L/U decomposition*, IEEE Transactions on Computers, C-31 (1982), pp. 231–239.
[22] S. G. Kratzer, *Massively parallel sparse matrix computations*, Tech. Report SRC–TR–90–008, Supercomputer Research Center, 1990.
[23] J. W. H. Liu, *The multifrontal method for sparse matrix solution: Theory and practice*, Tech. Report CS–90–04, York University Computer Science Department, 1990.
[24] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM Journal on Matrix Analysis and Applications, 11 (1990), pp. 134–172.
[25] J. Naor, M. Naor, and A. J. Schäffer, *Fast parallel algorithms for chordal graphs*, SIAM Journal on Computing, 18 (1989), pp. 327–349.
[26] B. W. Peyton, *Some Applications of Clique Trees to the Solution of Sparse Linear Systems*, PhD thesis, Clemson University, 1986.
[27] D. J. Rose, *A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations*, in Graph Theory and Computing, R. C. Read, ed., 1972, pp. 183–217.
[28] D. J. Rose, R. E. Tarjan, and G. S. Lueker, *Algorithmic aspects of vertex elimination on graphs*, SIAM Journal on Computing, 5 (1976), pp. 266–283.
[29] E. Rothberg and A. Gupta, *Fast sparse matrix factorization on modern workstations*, Tech. Report STAN–CS–89–1286, Stanford University, 1989.
[30] R. Schreiber, *A new implementation of sparse Gaussian elimination*, ACM Transactions on Mathematical Software, 8 (1982), pp. 256–276.
[31] R. Schreiber, *An assessment of the connection machine*, in Scientific Applications of the Connection Machine, H. Simon, ed., World Scientific, Singapore, 1991.
[32] H. Simon, P. Vu, and C. Yang, *Performance of a supernodal general sparse solver on the Cray*

*Y-MP*, Tech. Report SCA–TR–117, Boeing Computer Services, 1989.

[33] B. Speelpenning, *The generalized element method*, Tech. Report UIUCDCS–R–78–946, University of Illinois, 1978.

[34] Thinking Machines Corporation, Cambridge, Massachusetts, *Paris reference manual, version 5.0*.

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