Research Institute for Advanced Computer Science
NASA Ames Research Center

Highly Parallel Sparse Cholesky Factorization

John R. Gilbert Robert Schreiber

RIACS Technical Report 90.39 August, 1990

Submitted: SIAM Journal on Scientific and Statistical Computing

Highly Parallel Sparse Cholesky Factorization

John R. Gilbert Robert Schreiber

The Research Institute of Advanced Computer Science is operated by Universities Space Research Association, The American City Building, Suite 311, Columbia, MD 244, (301)730-2656

Work reported herein was supported by the NAS Systems Division of NASA and DARPA via Cooperative Agreement NCC 2-387 between NASA and the University Space Research Association (USRA). Work was performed at the Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center, Moffett Field, CA 94035.

Highly Parallel Sparse Cholesky Factorization

John R. Gilbert* Robert Schreiber†

March 8, 1990

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. Our most promising algorithm is one whose inner loop performs several dense factorizations simultaneously on a two-dimensional grid of processors. Virtually any massively parallel dense factorization algorithm can be 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.

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

†Research Institute for Advanced Computer Science, MS 230-5, NASA Ames Research Center, Moffett Field, CA 94035. This author's work was supported by the NAS Systems Division and DARPA via Cooperative Agreement NCC 2-387 between NASA and the University Space Research Association (USRA).

Keywords: sparse matrix algorithms, Cholesky factorization, systems of linear equations, parallel computing, data parallel algorithms, chordal graphs, Connection Machine, performance analysis.

AMS(MOS) subject classifications: 05C50, 15A23, 65F05, 65F50, 68M20.

1 Introduction

1.1 Data parallelism

Highly parallel computer architectures promise to achieve high performance inexpensively by assembling a large amount of simple hardware in a way that scales without bottlenecks. By associating a processor with every data element in a computation (at least conceptually), they present a programming model that is relatively simple compared to distributed architectures with medium-grain parallelism.

Some major challenges come along with these promises. Communication is expensive relative to computation, so an algorithm must minimize communication, and substitute simple communication patterns for complex ones where possible. The sequential programmer tunes the inner loop of an algorithm for high performance, but 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. For example, the inner two levels of looping in dense Cholesky factorization are performed in parallel, so the square root at each diagonal element is an "inner loop" computation that could dominate the entire running time of the algorithm. Efficient processor utilization is a challenge for the same reason: When an operation is applied to only a few data elements, the processors associated with the rest of the data sit idle.

Algorithms for data parallel architectures must make different trade-offs than sequential algorithms: They must exploit regularity in the data, but to be efficient they must also be highly regular in the time dimension. In some cases entirely new approaches may be appropriate for highly parallel algorithms; examples of experiments with such approaches include particle-in-box flow simulation, knowledge base maintenance [2], and the entire field of neural computation [9]. On the other hand, the same kind of regular-

ity 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 [1]; we shall see that they are also crucial to efficient data parallel matrix algorithms.

1.2 Goals of this study

Data parallel algorithms are attractive 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. Section 2 reviews the definitions we need from numerical linear algebra and graph theory, sketches the architecture of the Connection Machine, and presents 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. 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. In Section 5 we outline several steps necessary to improve the absoluteefficiency of the algorithm, most of which concern efficient Cholesky factorization of dense matrices. Finally we draw some conclusions and discuss avenues of further research.

2 Required definitions

For any real number x, we write $\lceil x \rceil$ to denote the smallest power of two not smaller than x. For any set S, we write $|S|$ to denote its cardinality. For any matrix X, we write $\eta(X)$ to denote the number of nonzero elements of X.

2.1 Linear algebra

Let A be an $n \times n$ real, symmetric, positive definite sparse matrix. There is a unique $n \times n$ lower triangular matrix L 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 nonzeros of L that were zero in A are called *fill* or *fill-in*.

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.) Nevertheless we plan to study the implementation of appropriate reordering and symbolic factorization procedures on data parallel architectures as well.

If the matrix A is such that its Cholesky factor L has no more nonzeros than 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 $A_{r,s}, r \in R, s \in S$.

2.2 Graph theory

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

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

(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

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

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 $\langle i, j\rangle$ 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 $G^*(A)$ [15]. 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 $G^*(A)$.

Every cycle of more than three vertices in $G^*(A)$ has an edge between two nonconsecutive vertices (a *chord*) [14]. A graph with this property is said to be *chordal*.

Let $G = G(V, E)$ be any undirected graph. A *clique* is a subset X of V such that for all $u, v \in V$, $\langle u, v\rangle \in E$. 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 adj(v), is the set $\{u \in V \mid \langle u, v\rangle \in E\}$. The *monotone neighborhood* of v, written madj(v), is the smaller set $\{u \in V \mid u > v, \langle u, v\rangle \in E\}$. We also use the usual extensions of adj and madj to sets of vertices.

A vertex v is *simplicial* if adj(v) is a clique. Two vertices, u and v, are *indistinguishable* if $\{u\} \cup \text{adj}(u) = \{v\} \cup \text{adj}(v)$. 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 adjancent to a vertex of B.

The proof of the following is immediate.

Proposition 1 *Two simplicial vertices are either independent or indistinguishable.*

A set of indistinguishable simplicial vertices forms a clique, though not in general a maximal clique. The proposition implies that the equivalence relation of indistinguishability partitions the simplicial vertices into pairwise independent cliques. We call these the *simplicial cliques* of the graph.

2.2.1 Elimination trees

A fundamental tool in studying sparse Gaussian elimination is the *elimination tree.* Schreiber [17] defined this structure, and Liu [12] 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 offdiagonal nonzero element in the u^{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 $\langle u, v\rangle$ 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.

2.3 The Connection Machine

The Connection Machine (model CM-2) is an SIMD parallel computer. A full-sized CM has $2^{16} = 65{,}536$ processors, each of which could directly access 65,536 bits of memory when this work was done. 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. 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. At the time of our work on the CM, v had to be a power of two.

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 an array of arbitrary rank with dimensions that are powers of two. The VP sets and their pvars are embedded in the machine using Gray codes that guarantee that neighboring virtual processors are stored and simulated by the same or neighboring physical processors.

Parallel computation is expressed through elementwise binary operations on pairs of pvars that have the same geometry and reside in the same VP set. Such operations take time proportional to v, for the actual processors must loop over their simulated virtual processors.

2.3.1 Connection Machine programming

The language of our pilot implementations is *lisp, which we have found to be a convenient means of expressing data parallel algorithms; we will therefore use some of the conventions and nomenclature of that language in our descriptions. We assume that the reader knows the rudiments of sequential lisp. A *lisp convention is that parallel variables and functions are given names ending in `!!` (suggesting two parallel lines).

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 (called NEWS grid communication, although the VP set may be of arbitrary rank) is done by the *lisp function **news!!**. For example, if `x!!` is a pvar defined on a two-dimensional VP set, then

```
(news!! x!! -1 0)
```

is `x!!` shifted a distance -1 in the first coordinate and not shifted in the second. The shift may be circular or end-off at the programmer's discretion.

The second mechanism is **scan!!**, or parallel prefix, which is familiar as the scan pseudo-operator of APL. For example, if `x!!` is a one-dimensional pvar with the value [1, 2, 3, 4, 5, 6, 7, 8] then

```
(scan!! x!! '+!!)
```

has the value [1, 3, 6, 10, 15, 21, 28, 36]; in general, `result(0) = x!!(0)`, and, for $i > 0$, `result(i) = result(i-1) OP x!!(i)` where `OP` is the combining operator specified by the programmer – addition in this case. Scans are implemented using the hypercube connections. At a virtual processor ratio of 1, the time for a scan of length n is in theory proportional to $\log n$, though as implemented on the CM it is most accurately modelled as being constant.

Scans can use other associative binary operators in place of `+!!`. We will only use scans with the left projection operator `'copy!!`; the effect of a so-called copy-scan is to copy `x!!(1)` to all elements of the result. This is the most efficient way to broadcast in the CM. In a two-dimensional VP geometry it can be used to broadcast along either rows or columns of a two-dimensional array.

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 `seg!!` with the value [T F F F T F F T] and `x!!` is as above, then

```
(scan!! x!! '+!! :segment-pvar seg!!)
```

returns [1, 3, 6, 10, 5, 11, 18, 8].

The third and most general form of communication, which allows a processor to access data in the memory of any other virtual processor, is done with the function `pref!!` and the form `*pset`. The address of the processor whose memory is to be read or written is taken from an integer pvar called the *address pvar*. Function `pref!!` is a parallel read: suppose the pvar `x!!` is one-dimensional with the 16 elements [15, 14, 13, ..., 2, 1, 0]. Let `p!!` be the integer address pvar. Suppose it has the value [0, 1, 2, 0, 1, 2, ..., 0, 1, 2, 0]. Then the result returned by

```
(pref!! x!! p!!)
```

is [15, 14, 13, 15, 14, 13, ..., 15, 14, 13, 15]; i.e. `result(i) = x!!(p!!(i))`.

Function ***pset** is a parallel write. Suppose that `p!!` and `x!!` are both [15, 14, 13, ..., 0]. Then

```
(*pset x!! y!! p!!)
```

has the side effect of storing [0, 1, 2, ..., 15] in `y!!`, i.e. `y!!(p!!(i)) = x!!(i)`.

When the address pvar has duplicate values, data from several processors is sent to the same destination processor. The value actually stored is some combination of the values received. The way that they are combined is specified by giving a combining operator such as **:add** in the ***pset** form. For example, if `p!!` is [0, 1, 2, 0, 1, 2, ..., 0, 1, 2, 0] and `y!!` is initially [1, 1, ..., 1] then

```
(*pset :add x!! y!! p!!)
```

has the side effect of changing `y!!` to [45, 40, 35, 1, 1, ..., 1]. The sum of elements `x!!(j)` such that `p!!(j) = k` is stored in `y!!(k)` if there are any such elements; otherwise `y!!(k)` is unchanged. Other combining operators (**:max**, **:min**, **:product**, etc.) are available.

2.3.2 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:

μ	The memory reference time for a 32 bit word
ϕ	The 32 bit floating point time, in units of μ
ν	The 32 bit **news!!** time, in units of ϕ
σ	The 32 bit **scan!!** time, in units of ϕ
ρ	The 32 bit router time, in units of ϕ

Our model is that time scales linearly with VP ratio, which is essentially correct for the Connection Machine. Therefore μ is proportional to VP

Connection Machine Parametric Model		
Parameter	Description	Measured CM2 value
v	Virtual processor ratio	
μ	32-bit memory reference time	4.77 $\cdot v\mu$sec
ϕ	Floating-point operation time $\div \mu$	7
σ	Scan time $\div \phi$	16 – 20
ν	News time $\div \phi$	2
ρ	Route time $\div \phi$	pset with no collisions: 64 pset with add ($\approx$ 4 collisions): 108 pset with add ($\approx$ 100 collisions): 203 pref (many collisions): 430

Table 1: Parameters of CM model

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. The range of scan times reflects the fact that the time actually depends somewhat on the number of underlying hypercube dimensions in the direction of the scan, which depends on the aspect ratio of the VP set. 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 much worse than this. For any given pattern, `pref!!` usually takes just twice as long as `*pset`, 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 ρ for `*pset` corresponding to the number of collisions observed, and model `pref!!` as taking 2ρ floating-point times.

3 Router Cholesky

Our first parallel Cholesky factorization algorithm is based closely on that of Gilbert and Hafsteinsson [7], which is a theoretically efficient algorithm in

the PRAM model of computation. Its communication requirements are too unstructured for it to be very efficient on a message-passing 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 [7] describe a parallel algorithm.) Each vertex of the tree corresponds to a column of the matrix.

A sequential column-oriented Cholesky factorization algorithm is as follows.

```
procedure Sequential-Cholesky(matrix A);
    for j ← 1 to n do
        for each edge ⟨i, j⟩ of G+(A) with i < j do
            cmod (i, j) od;
        cdiv (j) od
end Sequential-Cholesky;
```

Here routine *cdiv* (j) divides the subdiagonal elements of column j by the square root of the diagonal element in that column, and routine *cmod* (i, j) modifies column j by subtracting a multiple of column i. This is a *left-looking* algorithm, so-called because column j accumulates all necessary updates *cmod* (i, j) 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* (i, j) using column i immediately after the *cdiv* (i).

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

```
procedure Router-Cholesky(matrix A);
```

```
    for h ← 0 to height(n) do
        for each edge ⟨i, j⟩ with height(i) < height(j) = h pardo
            cmod (i, j) 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, the 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 is done.

A processor is assigned to every nonzero of the triangular factor (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* (i, j), 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* (i, j) 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, which 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 $\langle i, j \rangle$ with $i > j$. Incidentally, this one-dimensional column-major arrangement is a common storage layout for sequential sparse matrix algorithms.

We are profligate of parallel variable storage in Router Cholesky. Each processor contains 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*. Macros `in-active-column-p!!`, `in-active-row-p!!`, `in-done-column-p!!`, and `in-done-row-p!!` just compare the local processor's `i-ht!!` or `j-ht!!` to `active-height`, the current value of the outer loop index.

The *cdiv* uses a **scan** operation to copy the diagonal element to the rest of the active column. The following is a slightly simplified version of

`cdiv-active-columns`, which does all the *cdiv*'s at a particular height. The boolean `diagonal-p!!` separates the copy-scan into columns.

```
(*defun cdiv-active-columns (active-height)
     (*when (in-active-column-p!!)
          (*set l!!
               (/!! l!!
                    (scan!! (sqrt!! l!!) 'copy!!
                                 :segment-pvar diagonal-p!!)
               ))))
```

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 `pset :add`, which uses the router to send the update to its destination. The `:add` option means that multiple updates to the same element will be added together as they collide in the router.

To figure out where to send the update, each element maintains a pointer `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. To figure out where to send the update, each element maintains a pointer `next-update!!` to a later element in its row. At each stage, `next-update!!` is moved one step up the tree using the `e-parent!!` pointers. A simplified version of `cmod-active-columns` follows. We omit the details of the segmented scan that copies the multiplier down its column.

```
(*defun cmod-active-columns (active-height update-edge!!)
   (*let ((updates!! (!! 0.0))) ; accumulator for updates.

      ;; proc <ki> sends an update
      ;; (l<ki> * l<ji>) to element l<kj>
      (*when (in-done-column-p!!)
         ;; scan the multiplier down from the
         ;; unique element in an active row
         (*let ((l-j-i!! (scan-down-l-j-i!!)))
            ;; if a nonzero multiplier arrived then update
            (*when (/=!! l-j-i!! (!! 0.0))
```

```
      (*pset :add (*!! 1!! l-j-i!!)
                  updates!! update-edge!!)))
    ;; but in any event, update-edge must be updated
    (*when (in-active-column-p!! update-edge!!)
      (*set update-edge!!
            (pref!! e-parent!! update-edge!!))))

;; proc <kj> subtracts accumulated updates from l<kj>
(*when (in-active-column-p!!)
  (*set l!! (-!! l!! updates!!))
)))
```

3.3 Router Cholesky performance: Theory

Each stage of Router Cholesky does a constant number of router calls, 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 2.3.1, then, its running time is

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

Recall that the memory reference time μ 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 of the entire algorithm is incrementing the `update-edge!!` pointer. The router is used once (by a `pref!!`) in `(in-active-column-p!! update-edge!!)` to determine whether to do the increment, and again by the `pref!!` that does it. Counting the `*pset`, then, c_1 is about 5. Then c_2 is about 2 and c_3, which accounts for all the local computation, is about 4 (there is one square root, one divide, one multiplication and one subtraction). 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 ρ.

To get a feeling for this analysis, consider a model problem which is a 5-point finite difference mesh in two dimensions, ordered by nested dissection [4]. 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 ρ to be a constant independent of p or k; this is approximately correct for the Connection Machine although theoretically ρ 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 ρ, 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. 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 2500×2500 with 7400 nonzeros (counting symmetric entries only once), representing a 50×50 five-point square mesh. It is preordered by Sparspak's automatic nested dissection heuristic, which gives orderings very similar to the ideal nested dissection ordering used in the analysis of the model problem above. The Cholesky factor has $\eta(L) = 48608$ nonzeros, an elimination tree of height $h = 144$, and takes 1734724 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 8192 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 48608 of the 65536 virtual processors.) We observed a running time of 53 seconds, of which about 41 seconds was due to `pref!!` and `*pset`. 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 contributes 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 Router Cholesky.

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 `pref!!`'s and a `*pset` 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 [6], which compares very favorably to the complexity of a column-oriented sparse Cholesky code on a MIMD message-passing multiprocessor [5, 19]. This speaks well for the data parallel programming model of the Connection Machine, and suggests that with improvements in router technology future generations of data parallel machines may allow efficient parallel programs for complex tasks to be written nearly as easily as sequential programs.

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 multifrontal Cholesky algorithm and its implementation on the CM. 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 news and copy scan mechanisms for all the necessary communication during the factorization of the dense submatrices. Only when we need to move these dense submatrices to the playing field and remove Schur complements from it 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.

4.1 The Grid Cholesky algorithm

4.1.1 A block Jess and Kees reordering

Consider the chordal graph $G = G^*(A)$. The ordering $\{1, 2, \ldots, n\}$ is a perfect elimination ordering of G. We first present a method for reordering the vertices of G in such a way that we introduce no additional fill, producing a new perfect elimination ordering of G. The new ordering has two desirable properties: It eliminates vertices with identical monotone neighborhoods consecutively, and it greedily minimizes the height of a *clique tree* that we define below.

The objective of this strategy is twofold. First, in performing the factorization of A we may work with dense submatrices that correspond to sets of vertices of G having the same monotone neighborhood; we factor this submatrix on a two-dimensional grid of virtual processors using fast communication mechanisms. Second, we show that several such sets may be eliminated in parallel. Our reordering minimizes the number of parallel major steps (consisting of parallel elimination of independent sets of vertices of the same structure) in the same way that the Jess/Kees reordering procedure [10] minimizes the number of parallel steps over all perfect elimination orderings of G.

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

procedure Reorder(graph $G^*(A)$)
  $G \leftarrow G^*(A)$;
  active_stage $\leftarrow -1$;
  while G is not empty **do**
    active_stage $\leftarrow$ *active_stage* $+ 1$;
    Number all the simplicial vertices in G, with those in a given
      simplicial clique numbered consecutively;
    stage$(v) \leftarrow$ *active_stage* for all such vertices v;
    Remove all the simplicial vertices from G **od**;
  $\hbar \leftarrow$ *active_stage*
end *Reorder*

We contrast this with the Jess and Kees method for reordering to minimize elimination tree height. At one step we eliminate all the simplicial vertices (that is, all the vertices in every simplicial clique); at one step they eliminate a maximum-size independent set of simplicial vertices (that is, one vertex from each simplicial clique). Thus this might be called a "block Jess and Kees" ordering.

These cliques can also be arranged into a 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. We call this tree the *clique tree* of A. A related but not identical clique tree was used by Lewis, Peyton, and Pothen in their efficient implementation of the point Jess and Kees algorithm [11].

We shall refer to all the nodes of the clique tree as the "simplicial cliques" of A, although strictly speaking a clique is not simplicial until its children have been eliminated. Every vertex is included in exactly one simplicial clique. Suppose the simplicial cliques $\{S_1, \ldots, S_m\}$ are numbered 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 simplicial clique S is eliminated is the iteration

of the *while* loop at which its vertices are numbered and eliminated; thus, for all $v \in S$, $stage(v) = stage(S)$.

4.1.2 Multifrontal elimination

Let C be a simplicial clique. It is straightforward to show that $K = \mathrm{adj}(C) \cup C$ is also 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 simplicial cliques in this way.

Let $\gamma_C = |C|$ and $\sigma_C = |\mathrm{adj}(C)|$. Write $A(K,K)$ for the principal submatrix of order $|K| = \gamma_C + \sigma_C$ consisting of elements $A_{i,j}$ with $i,j \in K$. It is natural to partition $A(K,K)$ as

$$A(K,K) = \begin{pmatrix} X_C & E_C \\ E_C^T & Y_C \end{pmatrix},$$

where $X_C = A(C,C)$ is $\gamma_C \times \gamma_C$, $E_C = A(C, \mathrm{adj}(C))$ is $\gamma_C \times \sigma_C$, and $Y_C = A(\mathrm{adj}(C), \mathrm{adj}(C))$ is $\sigma_C \times \sigma_C$.

The Grid Cholesky algorithm is as follows:

procedure *Grid-Cholesky*(**matrix** A)
  for *active_stage* $\leftarrow$ 0 **to** $\hbar$ **do**
    forall simplicial cliques C such that *stage(C)* = *active_stage* **pardo**
      Move $A(K,K)$ to the playing field,
        where $K = C \cup \mathrm{adj}(C)$;
      Set Y_C to zero on the playing field;
      Perform γ_C steps of parallel Gaussian elimination without pivoting
        to compute the Cholesky factor L_C of X_C,
        the updated submatrix $E'_C = L_C^{-1} E_C$,
        and the Schur complement $Y'_C = -E_C^T X_C^{-1} E_C$;
      $A(C,C) \leftarrow L_C$;
      $A(\mathrm{adj}(C), C) \leftarrow E_C^{T\prime}$;
      $A(\mathrm{adj}(C), \mathrm{adj}(C)) \leftarrow A(\mathrm{adj}(C), \mathrm{adj}(C)) + Y'_C$ **od**
    od
end *Grid-Cholesky*;

4.2 Multiple dense partial factorization

In order to make this approach useful, we need to be able to perform dense matrix factorizations fast on two-dimensional VP sets. To that end, we discuss an implementation of LU decomposition without pivoting. (We can see no efficient way to exploit symmetry with a two-dimensional machine; moreover, compared with LL^T factorization, the LU factorization substitutes a reciprocal for a reciprocal square root 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 by copy scan. With either of these methods, all the submatrices $A(K, K)$ corresponding to simplicial cliques 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 algorithms in terms of their effect on a single submatrix $A(K, K)$, with a simplicial clique of size γ_C and a Schur complement of size σ_C.

4.2.1 Systolic factorization

Our systolic algorithm is based on the wavefront algorithm of O'Leary and Stewart [13]. It differs in that we compute an LU factorization instead of an LL^T factorization in order to avoid the diagonal square root, and of course we compute a partial factorization and a Schur complement instead of a complete factorization.

The communication is entirely nearest-neighbor. The wavefront moves across the matrix in steps. The number of steps is $3\gamma_C + 2\sigma_C$: the first wavefront must travel an l_1 distance of $2\gamma_C + 2\sigma_C$ to reach the lower right corner of the matrix, and in all γ_C wavefronts must reach that corner at a rate of one per step. A step consists of two NEWS operations (one in each dimension), a multiplication (as some processors compute elements of L), and a multiply-subtract (as some processors perform the unit steps of Gaussian elimination). Also, γ_C of the steps include a division to compute the inverse of a diagonal element.

If γ_0 and σ_0 are the sizes of the largest simplicial clique and Schur com-

plement in a given stage, then the running time of the stage is approximately

$$c_1 \nu \phi \mu + c_2 \phi \mu,$$

where c_1 is about $2(3\gamma_0 + 2\sigma_0)$ and c_2 accounts for the arithmetic mentioned above as well as some bookkeeping.

Here are the experimentally observed relative times taken by the various operations when factoring a single $n \times n$ dense matrix (so that $\gamma_0 = n$ and $\sigma_0 = 0$).

News (to move matrix elements systolically):	30.4%
Determining context and other bookeeping:	43.5%
Multiply (computing multipliers):	7.1%
Divide (reciprocal of pivot element):	9.1%
Multiply-subtract (Gaussian elimination):	10.1%

4.2.2 Factorization by rank-1 update

The second dense partial factorization algorithm works by applying rank-1 updates. 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 γ_C, the size of the simplicial clique. Again we compute an LU factorization instead of a Cholesky factorization to substitute a reciprocal for a square root in the inner loop of the algorithm. (At the end of each stage we convert the LU factorization to a Cholesky factorization by taking square roots of all the diagonal elements simultaneously, scanning them down their columns, and dividing by them, which takes negligible time.)

In terms of the parameters above, a stage of rank-1 partial factorization

takes time

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

Here c_3 is about $2\gamma_0$, and constant c_4 is at most about $c_2/3$ (or smaller if $\sigma_0 > 0$).

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 there is no Schur complement). The bookkeeping includes nearest-neighbor **news** operations to move three one-bit tokens that control which processors perform reciprocals, multiplications, and so on at each step.

Copy 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%

4.2.3 Remarks on dense partial factorization

The choice between systolic and rank-1 factorization depends on the architecture. 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 $\sigma \approx 3\nu$, so for a full factorization a threefold decrease in communication time per step just balances the threefold increase in number of steps. For a partial factorization the rank-1 algorithm is the clear winner because its time does not grow with the size of the Schur complement. For example, for the two-dimensional model problem the average Schur complement size σ_C is about $4\gamma_C$, 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 three to four times slower than **news!!**.

It is interesting to note that the only arithmetic that matters in a sequential algorithm, the multiply-subtract, accounts for only 1/20 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 sixty for LU, or half that 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.

More efficient use of virtual processors could improve performance by a small factor, perhaps four. The VP set could shrink as the matrix shrinks, and the multiplies could be performed in a sparser VP set.

However significant improvements in performance must come from other sources. At least two such sources exist.

First, the scans could be sped up considerably within the hypercube connection structure of the CM. Ho and Johnsson [8] 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 *lisp **scan** which takes $O(b + d)$. It is rumored to be available in a forthcoming release of the CM Fortran library.

Second, more efficient use of the low-level floating-point architecture of the CM-2 is possible. The performance model of Section 2.3 does not take into account 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 a lot of time when only a few processors are actually active, such as when computing multipliers or diagonal reciprocals. This mode also requires intermediate results to be stored back to main memory, precluding the use of block algorithms that could store intermediate results in the registers in the transposer chip. This causes the computation rate to be limited by the bandwidth between the transposer chip and the processor memories instead of by the operation rate

of the vector chip.

A more efficient dense matrix factorization can be achieved by thinking of each 32-processor-plus-transposer-and-vector-chip unit as a single processor, and representing 32-bit floating-point numbers "sideways", with one bit per physical processor, so that they do not need to be moved bit-serially into the arithmetic unit. At the time this work was done the tools for programming on this level were not easily usable. Recently, Thinking Machines has made available a library of dense matrix routines that use this approach; we are currently considering how best to incorporate it into our code. (It is also rumored that CM Fortran will eventually adopt this model.)

4.3 CM implementation of Grid Cholesky

We use two VP sets for Grid Cholesky: `matrix-storage` stores the nonzero elements of A and L (doing almost no computation), and `factoring-grid` implements the playing field 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. Here is a slightly simplified version of the code.

```
(*defun grid-factor () "Factor matrix a to produce l"
      (*set l-value!! a-value!!)
      (dotimes (stage max-stage)
            (move-to-factor-grid stage)
            (factor-on-grid stage)
            (update-from-factor-grid stage)
      ))
```

We present simplified versions of the three main subroutines in the Appendix.

4.3.1 Matrix storage

The VP set `matrix-storage` is a one-dimensional array of virtual processors that stores all of A and L in a form similar to the standard column-oriented sparse storage scheme used for example in Sparspak [6], and in

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 simplicial clique.

`updates!!` Working storage for sum of incoming updates.

Routine `move-to-factor-grid` uses `*pset` to move the active columns from `matrix-storage` to the playing field. The simplicial cliques C are disjoint, but their neighboring sets 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, routine `update-from-factor-grid` uses `*pset :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 `factoring-grid`, is the two-dimensional playing field on which the simplicial cliques 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 subroutine `factor-on-grid` carries out the factorizations on the playing field. It performs partial LU factorization by simultaneously doing

rank-1 updates of all the dense submatrices on the playing field, as described in Section 4.2.2. The number of rank-1 update steps is the size of the largest simplicial clique at the current stage. The submatrices may be different sizes; each matrix only does as many rank-1 updates as the size of its simplicial clique. We omit the complete code for this subroutine because the bookkeeping operations to do all the factorizations at once render it opaque. Instead, the Appendix contains a simplified code that computes a Schur complement in a single dense matrix by rank-1 updates.

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 comptation 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 [3]. 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 are a fixed number of router calls per stage, so the matrix storage time is

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

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

We express the playing field time as a sum over levels. At each level the number of rank-1 updates is the size of the largest simplicial clique at that

stage s	$R(s)$	$\max \gamma_C$	$\max(\gamma_C + \sigma_C)$	$\sum(\gamma_C + \sigma_C)^2$
$\hbar$	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}	$k/2^{r+1}$	$7k/2^{r+1}$	$24.5k^2$

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

level. According to the analysis in Section 4.2.2, then,

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

where c_6 and c_7 are constants (in fact $c_6 = 2$), and the subscript s indicates that the value of μ 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 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 5-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 simplicial cliques at stage s.
$\max \gamma_C$	Size of largest simplicial clique at stage s.
$\max(\gamma_C + \sigma_C)$	Size of largest maximal clique $C \cup \text{adj}(C)$ at stage s.
$\sum(\gamma_C + \sigma_C)^2$	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 about $25k^2$ suffices if the problems can be packed in without too much loss. Rounding this up to a power of 2, we actually need to use a $4k \times 8k$ playing field. The VP ratio is $O(k^2/p)$. The sum over all stages of $\max \gamma_C$ 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 ρ appears only in the second-order term. This is because the playing-field computations are done on dense matrices with more efficient grid communication.

One way of looking at this difference 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 $O(\log k)$ routers. This means that the router time becomes less important as the problem size grows. 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.

4.5 Grid Cholesky performance: Experiments

Here we present experimental results from a relatively small model problem, the matrix arising from the 5-point discretization of the Laplacian on a square 63×63 mesh, ordered by nested dissection. This matrix has $n = 3969$ columns and 11781 nonzeros (counting symmetric entries only once). The Cholesky factor has $\eta(L) = 85416$ nonzeros, a clique tree with $\hbar = 11$ stages of simplicial cliques, and takes 3658949 arithmetic operations to compute.

The VP set **matrix-storage** requires 128K VPs. The fixed-size playing field requires 256×512 VPs (which is quite inefficient for the last few stages, where we see from Table 2 that the total size of the dense subproblems is much smaller). We performed our experiments on CM-2's at the Xerox Palo Alto Research Center and the NASA Ames Research Center. The results quoted here are from 8192 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 copy 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 ***psets** at each stage. Our analytic model predicts playing field time to be just 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 $\hbar \cdot 4\rho\phi\mu_{MS}$. This comes to between 1.5 and 4.7 seconds, depending on which value we choose for ρ. In fact 3/4 of the routers are ***pset** with no collisions, and the other 1/4 are ***pset :add** typically with two to four collisions. The fit to the model is therefore quite close.

4.6 Remarks on Grid Cholesky

The 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.

Move-to-factor-grid:	12%
Factor-on-grid:	67%
Update-from-factor-grid:	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 calls, would only increase 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

$$\sum_{s=1}^{h} \max_{stage(C)=s} \gamma_C,$$

which for the model problem is $3k$. On a playing field of $32k^2$ processors (with dimensions rounded up to a power of 2), this allows us to do $96k^3$ flops. The number of "useful flops" (that is, flops in the sequential Cholesky factorization) is $(829/84)k^3$ plus lower order terms. This is an efficiency of about $829/(84 \times 96) = 10.3\%$. 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 5/4); each rank-1 update consists of a divide, multiply, and multiply-add, the first two of which occur in only a small number of processors (factor of about 5/2); and as the dense factorization progresses processors in the simplicial cliques fall idle (factor of about 3/2).

It is also interesting to note that computing many Schur complements in parallel is actually more efficient on a mesh of processors than computing a single dense factorization. The reason is that computing Schur complements keeps more of the processors busy all the time, while the processors involved in a factorization fall idle as their elements are computed. A careful analysis of the model problem indicates that if the VP ratio of the playing field could be varied at each stage so that the dense submatrices corresponding to maximal cliques just fit on it, then useful flop rate for the playing field part of the model problem would be about 192% of that for a dense factorization. This illustrates the importance of regularity in the time dimension for data parallel algorithms.

On this small example, Grid Cholesky is about 20 times as fast as Router Cholesky. It is, however, only running at .597 megaflops on 8K processors, which would scale to 4.77 megaflops on a full 64K machine. A larger problem would run somewhat faster, but it is clear that making Grid Cholesky attractive will require improvements in the dense partial factorizations along the lines described in Section 4.2.3.

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 for the Connection Machine.

We therefore developed a parallel supernodal algorithm, Grid Cholesky, that does most of its work with efficient communication on dense submatrices. Analysis shows that the requirement for expensive general-purpose communication grows only logarithmically with problem size, and experiment shows that Grid Cholesky is about 20 times as fast as Router Cholesky for a moderately small sample problem. 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.

As it stands, our pilot implementation of Grid Cholesky is not fast enough to make the Connection Machine a cost-effective alternative to minisupercomputers for solving generally structured sparse matrix problems. We believe, however, that these experiments and analysis lead to the conclusion that a parallel supernodal/multifrontal algorithm can be made to perform efficiently on a data parallel machine. This is because, first, the performance of our pilot implementation is limited basically by the performance of its dense matrix kernel; and, second, the path to improving that kernel is fairly clear.

Let us expand on the latter point. Potential sources of increased efficiency in the dense factorization include: representing playing field data sideways (factor of perhaps 2); taking advantage of transposer chip registers (factor of perhaps 5); improved algorithms for row and column broadcasts

(factor of perhaps 3); changing the VP mapping at each stage to use the playing field more fully (factor of perhaps 3). These directions make the factor of 5000 between the performance of our pilot implementation and the 27-gigaflop theoretical peak performance of a 64K processor CM yawn somewhat less impressively.

We note that most of these improvements are below the level of the virtual processor abstraction, which is to say below the level of the assembly-language level architecture of the machine. Though TMC has recently made available a low-level language called CMIS in which a user can program below the virtual-processor level, we believe that ultimately most of these dense matrix optimizations should be applied by high-level language compilers. In other words, we believe that future high-level language compilers for data parallel machines, while they may support the virtual processor abstraction at the user's level, will generate code at a level below that abstraction. The CM Fortran compiler is moving in that direction, which interestingly enough makes Fortran in some ways the highest-level of the languages available for the Connection Machine.

In summary, even though our pilot implementation is not extremely fast, we are nonetheless very encouraged both about the Grid Cholesky algorithm, and about the potential of data parallel architectures for solving unstructured problems.

We mention four good avenues for further research.

The first is scheduling the dense partial factorizations efficiently. The tree of simplicial cliques 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.

The second avenue is improving the matrix storage VP set time. Of course, as problems get larger this time takes 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.

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 [18] and even on workstations [16]. 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 used to implement sequential algorithms efficiently on sequential machines with hierarchical storage will turn out to be useful for data parallel machines.

Appendix: Details of Grid Cholesky

Here we give a somewhat more detailed view of the `*lisp` implementation of Grid Cholesky, as described in Section 4.

The Cholesky factor L is held in a one-dimensional VP set, which is called `matrix-storage`. The pvars used are

`a-value!!`	Elements of A.
`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 simplicial clique.

updates!! Working storage for sum of incoming updates.

The playing field is a two-dimensional VP set called **factoring-grid** that is large enough to hold all the principal submatrices for all maximal cliques at any stage. Its size is determined as part of the symbolic factorization and reordering. (As described in the main paper, an optimization would be to reconfigure this VP set at each stage of the factorization to fit the submatrices at that stage.) 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.

in-clique-p!! A boolean pvar array that is true when this location holds an element whose column is a member of a simplicial clique at this stage; indexed by stage.

in-schur-p!! A boolean pvar array that is true when this location is in a Schur complement; indexed by stage.

Here is the ***lisp** code. As in Section 3, the code has been simplified somewhat in the interest of clarity.

```
(*defun grid-factor () "Factor matrix a to produce l"
      (*set l-value!! a-value!!)
      (dotimes (stage max-stage)
            (move-to-factor-grid stage)
            (factor-on-grid stage)
            (update-from-factor-grid stage)
      ))
```

The two functions **move-to-factor-grid** and **update-from-factor-grid** do pretty much what their names say they do. Here is **move-to-factor-grid**:

```
(*defun move-to-factor-grid (stage)  "Move columns active at
                                      this stage to the playing field"
```

```
        (*with-vp-set factoring-grid
          (*set dense-a!! (!! 0.0)))
        (*with-vp-set matrix-storage
          (*when (=!! active-stage!! (!! stage))

            ;; Move the lower triangle of each active
            ;; simplicial clique to the playing field.
            (*pset :no-collisions
                   l-value!!
                   dense-a!!
                   grid-i!! grid-j!!
                   :vp-set factoring-grid)
      ;; Move same stuff to the upper triangle
      ;; in the playing field.
           (*pset :no-collisions
                  l-value!!
                  dense-a!!
                  grid-j!! grid-i!!
                  :vp-set factoring-grid))))
```

and here is update-from-factor-grid:

```
(*defun update-from-factor-grid (stage)  "Move the updates
                                back from the playing field."
     (*with-vp-set matrix-storage
       (*set updates!! (!! 0.0)))

     (*with-vp-set factoring-grid

       ;; First store back the completely computed
       ;; columns of the simplicial cliques.
       (*when (aref in-clique-p!! (!!stage))
            (*pset :no-collisions
                   dense-a!!
                   l-value!!
                   (aref!! update-dest!! (!! stage))
                   :vp-set matrix-storage))
```

```
  ;; Next accumulate the updates from the newly
  ;; computed Schur complements.
  (*when in-schur-p!!
       (*pset :add
              dense-a!!
              updates!!
              (aref!! update-dest!! (!! stage))
              :vp-set matrix-storage)))

;; Finally, add the updates to the original matrix values
(*with-vp-set matrix storage
     (*set l-value!! (+!! l-value!! updates!!))))
```

The procedure `factor-on-grid` carries out the factorizations on the playing field. Instead of showing all the details of `factor-on-grid` (which is rather opaque), we present the following code, which is a version that omits the bookkeeping necessary to solve many problems at once in parallel. The following code uses the algorithm of `factor-on-grid` to compute a single Schur complement in a matrix `a!!` on a two-dimensional VP set:

```
(*defun rank-1-dense-factor (a!! n gamma)
            "Scur complement of a dense matrix"

     ;; a!! is the matrix.
     ;; On output, it holds the partial LU decomposition.
     ;; n is the order of a!!.  The VP set is n by n
     ;; gamma is the order of the (1,1)
     ;; block of a!! to be eliminated

          (*set row!! (self-address-grid!! 0))
          (*set col!! (self-address-grid!! 1))

     ;; done-token is true in the pivot row
     ;; mult-token is true in the pivot column
     ;; div-token is true at the pivot element
     ;; update-token is true in a(k+1:n, k+1:n)
```

```
    (*set done-token!! (=!! row!! (!! 0)))
    (*set mult-token!! (=!! col!! (!! 0)))
    (*set div-token!! (and!! done-token!! mult-token!!))
    (*when div-token!!
      (*set done-token!! nil!!)
      (*set mult-token!! nil!!))
    (*set update-token!!
         (and!! (>!! row!! (!! 0))
                (<=!! row!! (!! (- n 1)))
                (>!! col!! (!! 0))
                (<=!! col!! (!! (- n 1)))))
    (*when (or!! div-token!! mult-token!! done-token!!)
      (*set update-token!! nil!!))

;; Gaussian elimination
    (dotimes (k gamma)
      (*when div-token!!
        (*set a!! (/!! (!! 1.0) a!!)))
;; Broadcast the pivot row
      (*set col-belt!!
        (scan!! a!! 'copy!! :dimension 0
          :segment-pvar (or!! div-token!! done-token!!)))
;; Compute the multipliers
      (*when mult-token!!
        (*set a!! (*!! col-belt!! a!!)))
;; Broadcast the multipliers
      (*set row-belt!!
           (scan!! a!! 'copy!! :dimension 1
                :segment-pvar mult-token!!))
;; Rank-one update to the submatrix
      (*when update-token!!
        (*decf a!!
              (*!! row-belt!! col-belt!!)))
;; move the tokens using news!!
      (*set done-token!! (news!! done-token!! -1 0))
      (*set mult-token!! (news!! mult-token!! 0 -1))
      (*when (or!! done-token!! mult-token!!)
        (*set update-token!! nil!!))
```

```
(*set div-token!!
      (and!! done-token!! mult-token!!))
(*when div-token!!
  (*set done-token!! nil!!)
  (*set mult-token!! nil!!))
))
```

The code uses four boolean pvars (the "tokens") to determine context. `div-token` is true in $VP(k,k)$ at step k and triggers taking the reciprocal of the pivot element $a_{k,k}^{(k-1)}$. `done-token` is true in $VP(k,*)$ at step k and triggers the broadcast of the pivot row to all other rows. `mult-token` is true in $VP(*,k)$ at step k and triggers the computation of multipliers $a_{i,k}^{(k)} = a_{i,k}^{(k-1)} / a_{k,k}^{(k-1)}$. Finally, `update-token` is true in all virtual processors $VP(i,j)$ with $k < i,j \leq n$ and triggers the elimination operation (the `*decf` form in the code).

References

[1] Christian H. Bischof and Jack J. Dongarra. A project for developing a linear algebra library for high-performance computers. Technical Report MCS–P105–0989, Argonne National Laboratory, 1989.

[2] M. Dixon and J. de Kleer. Massively parallel assumption-based truth maintenance. In *Proceedings of the National Conference on Artificial Intelligence*, pages 199–204, 1988.

[3] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[4] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10:345–363, 1973.

[5] Alan George, Michael T. Heath, Joseph Liu, and Esmond Ng. Sparse Cholesky factorization on a local-memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9:327–340, 1988.

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

[7] John R. Gilbert and Hjálmtýr Hafsteinsson. Parallel solution of sparse linear systems. In *SWAT 88: Proceedings of the First Scandinavian Workshop on Algorithm Theory*, pages 145–153. Springer-Verlag Lecture Notes in Computer Science 318, 1988.

[8] Ching-Tien Ho and S. Lennart Johnsson. Spanning balanced trees in Boolean cubes. *SIAM Journal on Scientific and Statistical Computing*, 10:607–630, 1989.

[9] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Science*, 79:2554–2558, 1982.

[10] Jochen A. G. Jess and H. G. M. Kees. A data structure for parallel L/U decomposition. *IEEE Transactions on Computers*, C-31:231–239, 1982.

[11] John G. Lewis, Barry W. Peyton, and Alex Pothen. A fast algorithm for reordering sparse matrices for parallel factorization. *SIAM Journal on Scientific and Statistical Computing*, 10:1146–1173, 1989.

[12] Joseph W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[13] Dianne P. O'Leary and G. W. Stewart. Data-flow algorithms for parallel matrix computations. *Communications of the ACM*, 28:840–853, 1985.

[14] Donald J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In Ronald C. Read, editor, *Graph Theory and Computing*, pages 183–217, 1972.

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

[16] Edward Rothberg and Anoop Gupta. Fast sparse matrix factorization on modern workstations. Technical Report STAN–CS–89–1286, Stanford University, 1989.

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

[18] Horst Simon, Phuong Vu, and Chao Yang. Performance of a supernodal general sparse solver on the Cray Y-MP. Technical Report SCA–TR–117, Boeing Computer Services, 1989.

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