Assignment and Scheduling in Parallel Matrix Factorization*

Dianne P. O'Leary and G. W. Stewart
Computer Science Department
University of Maryland
College Park, Maryland 20742

Submitted by R. G. Voigt

ABSTRACT

We consider the problem of factoring a dense $n \times n$ matrix on a network consisting of P MIMD processors, with no shared memory, when the network is smaller than the number of elements in the matrix ($P < n^2$). The specific example analyzed is a computational network that arises in computing the LU, QR, or Cholesky factorizations. We prove that if the nodes of the network are evenly distributed among processors and if computations are scheduled by a round-robin or a least-recently-executed scheduling algorithm, then optimal order of speedup is achieved. However, such speedup is not necessarily achieved for other scheduling algorithms or if the computation for the nodes is inappropriately split across processors, and we give examples of these phenomena. Lower bounds on execution time for the algorithm are established for two important node-assignment strategies.

1. INTRODUCTION

Many dense-matrix calculations can be formulated efficiently as data-flow algorithms; for example, the data-flow formalism has been used to solve linear systems of equations [1, 7, 8], symmetric eigenvalue problems [2], and Liapunov equations [9]. In a data-flow algorithm, computations are partitioned into computational nodes which are vertices of a graph whose arcs represent the communication paths between nodes. A node can communicate with nodes adjacent to it in the network, sending and requesting data. A node that has requested data is not permitted to execute until all the data it has requested have arrived. Thus the computation is synchronized by the flow of data between nodes—whence the term data-flow algorithm. For a formal definition of this model of computation see [10].

*This work was supported by the Air Force Office of Scientific Research under Grant AFOSR-82-0078.

52 Vanderbilt Ave., New York, NY 10017 0024-3795/86/$3.50

One way to implement a data-flow algorithm is to assign each node in the computational network to a processor in an isomorphic network of processors. However, the size of the processor network will then restrict the size of problems that can be solved. Alternatively, we can allow assignments of multiple computational nodes to each processor, subject to the restriction that adjacent computational nodes must be assigned to adjacent processors (any processor will be considered to be adjacent to itself). In the language of operating systems, the computational nodes become tasks on the processors. The execution of the nodes must then be coordinated by an operating system resident on each processor (for a sketch of one such system see [9]).

Although the multiple assignment of nodes to processors solves the problem of oversized networks, it creates two new problems. First, there may be many ways of assigning nodes to processors, and the question arises of which are best. Second, it may happen that several nodes on a processor will become ready for execution at the same time, in which case the operating system must choose one of them for execution according to some scheduling algorithm. Again the question arises of which scheduling algorithms are best.

In this paper we consider the problems of scheduling and assignment for a computational network to factor dense matrices on a parallel computer consisting of processors with independent instruction streams and no shared memory. The network and flow of data is almost the same whether the network is used to compute the *LU* (without pivoting), the *QR*, or the Cholesky factorization of the matrix.

The problem of factoring an $n \times n$ matrix on fewer than n^2 processors has also been considered by other people. Srinivas [11] finds an optimal scheduling for fewer than n processors with shared memory. Ipsen, Saad, and Schultz [6] compute lower bounds on the time for factorization on a ring of vector processors. George, Heath, and Liu [4] analyze some algorithms for an architecture like the HEP.

In Section 2 we discuss the parallel factorization algorithm and various ways to assign nodes to processors. In Section 3 we analyze the time complexity of the algorithm assuming that the nodes are scheduled for execution under a round-robin or a least-recently-executed regime. We also give examples to show that care must be exercised in scheduling and assignment. In Section 4 lower bounds are established for the execution time for several natural assignment strategies.

2. THE PARALLEL CHOLESKY ALGORITHM: ELEMENTARY LOWER BOUNDS

Since our results apply to the parallel computation of all three factorizations mentioned above, we may confine our investigations to one of them. For

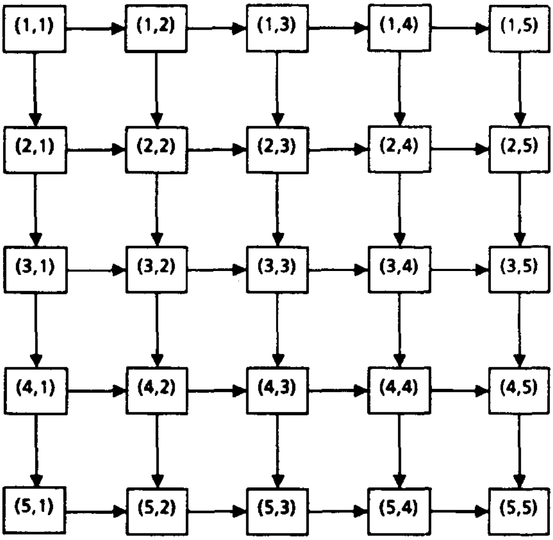


FIG. 1. Computational network for the Cholesky algorithm.

definiteness, we shall treat the parallel computation of the Cholesky factorization of a positive definite symmetric matrix. We shall assume that the reader is familiar with the usual sequential algorithm for computing this factorization.

A computational network for the algorithm is illustrated in Figure 1 for the case $n = 5$. The nodes, which are identified by ordered pairs of integers, correspond to the elements of the matrix A; e.g. the node (3,4) corresponds the element a_{34}. Note that the node (I, J) is connected only to the nodes $(I-1, J)$, $(I, J+1)$, $(I+1, J)$, and $(I, J-1)$, provided these nodes exist. Whenever it is clear from context that we are dealing with the node (I, J), we shall refer to the four surrounding nodes by the names *north*, *east*, *south*, and *west*.

Figure 2 contains a program for the node labeled (I, J) that transforms the element a_{IJ} into the corresponding element of the Cholesky factor. The program, a slight variant of the one given in [9], is written in a conventional sequential programming language, augmented by three commands. The command **send** causes the data specified by its first argument to be sent to the node specified by the second argument. For brevity, we denote multiple invocations of **send** by repeated argument lists. Messages from one particular node to another particular one are assumed to arrive in the order in which they are sent. For simplicity, we will assume that messages directed to nonexistent nodes [e.g., the message that node (1,5) sends to (1,6) when

```
init:   k := 0;
        loop
          k := k+1;
          if k=I and k=J then
sqrt:       await();
            a := sqrt(a);
            send(a:south) (a:east);
            finis;
          elsif k=J then
cdiv:       await(an:north);
            a := a/an;
            send(an:south) (a:east);
            finis;
          elsif k=I then
rdiv:       await(aw:west);
            a := a/aw;
            send(aw:east) (a:south);
            finis;
          else
elim:       await(an:north) (aw:west);
            a := a - an*aw;
            send(an:south) (aw:east);
          end if;
        end loop;
```

FIG. 2. Program for the Cholesky decomposition, node (I, J).

$n = 5]$ are simply ignored by the operating system. The **await** command causes the node to suspend execution until data from the node specified by the second argument arrive. When this occurs, the data are stored as specified by the first argument, and execution is resumed at the next statement. The **finis** command terminates the execution of the node.

Note the dummy **await** in the statement labeled sqrt. It has the effect of returning control to the operating system so that other nodes can be awakened. Since the node has not requested any data, it will be awakened when it next is examined by the scheduling algorithm. We include this statement in order to make the precedence graph, defined below, more regular.

The best way to understand the program is to trace the execution of a particular node, say (3,4). We will refer to the nonzero element in position (i, j) of either the upper triangular (if $i \leqslant j$) or lower triangular (if $j \leqslant i$) Cholesky factor by r_{ij}. Then r_{34} is given by

$$r_{34} = \frac{a_{34} - r_{31}r_{14} - r_{32}r_{24}}{r_{33}}. \tag{2.1}$$

When $k = 1$ in the program, the node (3,4) waits for two items. The one coming from the north is r_{14}, which is sent by node (1,4) to (2,4) and from there passed on to (3,4). The one from the west is r_{31}, which is passed from (3,1) via (3,2) and (3,3). On receiving these numbers, the node adjusts a_{34} by subtracting $r_{31}r_{14}$ from it, and then passes r_{14} to the south and r_{31} to the east for use by other nodes. When $k = 2$, the node behaves analogously, requesting r_{24} from the north and r_{32} from the west, subtracting their product from the current value of a, and passing on r_{24} to the south and r_{32} to the east. Finally, when $k = 3$ the node enters the section labeled rdiv, receives r_{33} from the west, divides it into the current value of a to produce r_{34}, and passes r_{33} south for use by the rest of the nodes in the third column. It passes its final value east, and then terminates.

To see how data flow from node to node, the reader may find it useful to trace the actions of the programs for the nodes (3,4) and (4,4) together.

In the spirit of the data-flow approach, the above description has been local. However, to develop scheduling strategies it is necessary to have a global view of the algorithm. This may be done by partitioning the work done by the nodes into *waves* indexed by the value of k in the program of Figure 2. If the nodes are regarded as executing in lockstep, firing only when their data are available, then the first wave moves in a diagonal front across the matrix, first touching the (1,1) node, then the (2,1) and (1,2) nodes in parallel, then the (3,1), (2,2), and (1,3) nodes—and so on. The second wave ($k = 2$) follows the first beginning with the (2,2) node. We shall have frequent occasion to refer to these waves in Section 4.

O'Leary and Stewart [10] have shown that however the nodes in a data-flow algorithm are sequenced for execution, each individual node receives input in a unique order and performs a unique series of actions, including **send**, **await**, and **finis** commands. This allows us to construct a *precedence graph* for the algorithm as follows. A vertex in the graph consists of any sequence of operations executed in a node beginning with an **await** command and ending just before an **await** or a **finis** command. (Every node program may begin with an initialization step, but this is not considered to be a vertex.) By the determinacy of the operations performed by a node, the vertices associated with any one node are linearly ordered, and we connect them with directed edges (arrows) in that order. These arrows represent *control-flow* synchronization of the algorithm (see, for example, [5, p. 29]). We also connect two vertices if the first contains a **send** command that satisfies an item in the **await** command beginning the second. These arrows represent the data-flow synchronization of the algorithm. The result is an acyclic digraph, which partially orders the vertices: the operations in a vertex can be performed only after the operations in all preceding vertices have been performed.

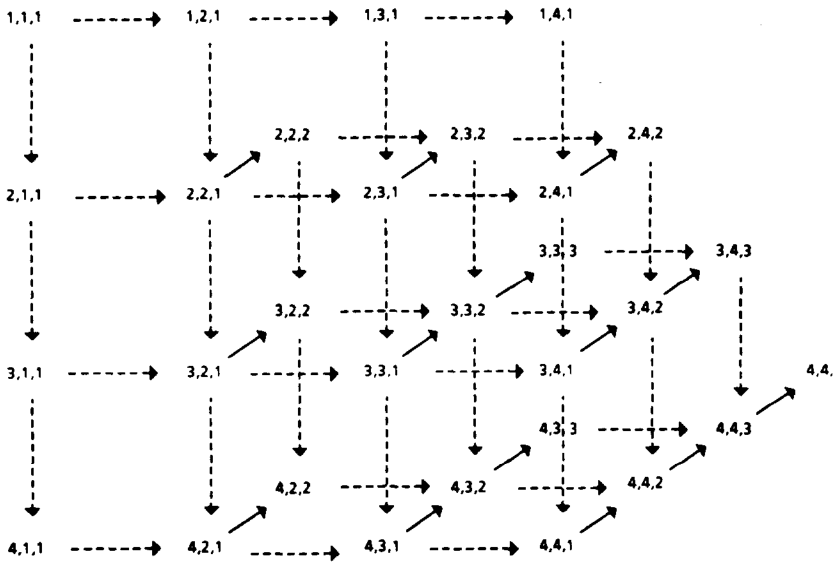


FIG. 3. Precedence graph for the Cholesky algorithm.

The precedence graph for the Cholesky algorithm is given in Figure 3.[1] A glance at the program in Figure 2 will show that each vertex for a particular computational node (i, j) must be associated with a unique value of k. Hence we label the vertices (i, j, k). The dashed arrows between nodes having the same value of k represent data-flow synchronization; the solid arrows between nodes having different values of k represent control flow.

It is important not to confuse the precedence graph with the computational network from which it derives. To keep the distinction clear, we shall consistently refer to the *nodes* of a computational network and the *vertices* of its precedence graph.

Any path in the precedence graph of a data-flow algorithm determines a lower bound on the time required to execute the algorithm. Specifically, since each vertex on the path must be executed in sequence, the algorithm cannot take less than the sum of the execution times of the vertices on any path, no

[1] Note that the precedence graph is uniquely determined by the node program in Figure 2 and a particular value of n. Other node programs which perform the same mathematical computation may lead to either the same precedence graph or a different one. The graph for the LU and QR algorithms differs slightly in that the (n, n, n) node is omitted. The QR takes rotations in the order $(1,2),(1,3),\ldots,(1, n),\ldots,(n-1, n)$.

matter how many processors are used, how the nodes are assigned to the processors, and how nodes are scheduled for execution on the processors.

For the Cholesky algorithm, the longest path is $3n-2$. Hence if $D_{\min}$ denotes the minimum time required to execute a vertex, then the time T_D required to execute the Cholesky algorithm is bounded below by $(3n-2)D_{\min}$:

$$T_D \geqslant (3n-2)D_{\min}. \tag{2.2}$$

This bound holds for any algorithm which has the precedence graph of Figure 3.

In the sequel we shall be concerned with the implementation of the algorithm on a network consisting of P processors. For this case we can derive another lower bound on the execution time. Specifically, there are $n^3/3+O(n^2)$ vertices in the precedence graph. If each vertex takes the same amount of time to execute, then even when all processors are fully utilized the algorithm cannot finish in less than $n^3/(3P)+O(n^2/P)$ times the time D required to execute a single vertex:

$$T_D \geqslant \left[\frac{n^3}{3P}+O(n^2)\right]D. \tag{2.3}$$

This bound holds for any algorithm which performs $n^3/3+O(n^2)$ computations, whether or not it has the precedence graph of Figure 3. It should be noted that this lower bound depends critically on the assumption that the vertices all require the same amount of time to execute; if some vertices execute in a very short time compared to the others, it may be possible to assign them to a single processor while the remaining processors execute the slower vertices. Fortunately, this assumption is approximately true for the Cholesky algorithm.

3. SCHEDULING ALGORITHMS

In this section we shall show that two natural scheduling algorithms for the nodes of Figure 1 are in some sense optimal under very weak assumptions about assignment. Optimality is defined by comparison with lower bound (2.3) established in Section 2, and consequently the caveat about vertices requiring equal time for their execution applies in interpreting the results of this section. In this section we also ignore transmission delays. The results apply to any arrangement of processors (square grid, hypercube, linear array, etc.) as long as adjacent nodes are assigned to adjacent processors.

The first scheduling algorithm is the *round-robin* algorithm. Here the operating system on a processor moves cyclicly through the nodes on the processor, looking for one that can execute. When it finds a node that is ready, the node is allowed to execute until it issues an **await** command, whereupon control is returned to the operating system, which goes on to the next node.

The basic result is that round-robin scheduling gives optimal order time if the nodes are more or less evenly distributed among the processors.

THEOREM 3.1. *Suppose that the vertices in the precedence graph for the Cholesky algorithm require time bounded by $D_{\max}$ to execute. Further suppose that there is a constant Q such that as $n \to \infty$ the number of nodes assigned to a processor is bounded by Qn^2/P, where P is the number of processors performing the computation. Then the time T_D required to execute the Cholesky algorithm with round-robin scheduling satisfies*

$$T_D \leqslant (3n-2)\frac{n^2}{P}QD_{\max}. \tag{3.1}$$

Proof. Let $T(i, j, k)$ be the time at which vertex (i, j, k) finishes executing, and let $D(i, j, k)$ be the time required for vertex (i, j, k) to execute. Then

$$\begin{aligned} T(i,j,k) = \max\{&T(i-1,j,k), T(i,j-1,k), T(i,j,k-1)\} \\ &+ D(i,j,k) + \delta(i,j,k), \end{aligned} \tag{3.2}$$

where $\delta(i, j, k)$ is the delay due to scheduling. Since at most $(Qn^2/P)-1$ processes can be examined before the vertex (i, j, k) is seen by the operating system,

$$\delta(i,j,k) \leqslant \left[\frac{n^2}{P}Q - 1\right]D_{\max}. \tag{3.3}$$

Hence

$$T(i,j,k) \leqslant \max\{T(i-1,j,k), T(i,j-1,k), T(i,j,k-1)\} + \frac{n^2}{P}QD_{\max}. \tag{3.4}$$

This recurrence can be solved to give

$$T(i,j,k) \leqslant (i+j+k-2)\frac{n^2}{P}QD_{\max}, \tag{3.5}$$

from which it follows that

$$T_D \equiv T(n,n,n) \leqslant (3n-2)\frac{n^2}{P}QD_{\max}. \quad \blacksquare \tag{3.6}$$

Note that Q must be greater than or equal to one. When it is near one, the nodes are spread out evenly among the processors, and the bound becomes effectively $(3n-2)(n^2/P)D_{\max}$. Comparing this value with the bound (2.3), we see that if all the vertices of the precedence graph require the same amount of time to execute, then balanced-assignment round-robin scheduling is less than optimal by no more than a factor of about nine.

Another method of scheduling is to give priority to nodes that have been waiting a long time. In this *least-recently-executed* algorithm, the operating system chooses from among the nodes that are ready to execute the one that has the earliest pending **await** command. Since a node ceases executing when it issues an **await** command and does not resume execution until the command is no longer pending, this algorithm chooses the node that has been waiting longest for a chance to execute. It turns out that Theorem 3.1 remains valid for the least-recently-executed scheduling algorithm. (Funderlic and Geist [3] give some simulations of least-recently-executed scheduling for the Cholesky algorithm on a torus of processors.)

THEOREM 3.2. *In Theorem* 3.1, *the bound* (3.1) *continues to hold when round-robin scheduling is replaced by least-recently-executed scheduling.*

Proof. Verify that (3.2) and (3.3) continue to hold. ■

It should not be thought that node balancing is enough to make any scheduling algorithm work well, as the following example shows.

EXAMPLE 3.3. Consider the following implementation of the parallel Cholesky algorithm on a $p \times p$ grid of processors (so that $P = p^2$). Suppose that for a positive integer m, we have $n = pm$. Partition the matrix A into $m \times m$ submatrices, and assign the nodes associated with each submatrix to the corresponding processor on the grid.

The scheduling algorithm is the following. The operating system on a processor will refuse to schedule a node lying on the last row or column of the submatrix on the processor until all the nodes not on the last row or column have executed their **finis** commands. In other words, the nodes on the last row or column of a submatrix are executed only when the processor has no prospect of doing anything else.

Let $t(i, j)$ denote the time that the (i, j)th processor first begins executing nodes on the last row and column of its submatrix. Clearly, the (i, j)th processor cannot begin executing *any* nodes before $t(i-1, j)$ or $t(i, j-1)$, whichever is larger. Moreover, before it can begin executing nodes on the last row or column, the $(m-1)^2$ other nodes must process all the data being sent to them from the north and west, something that requires time bounded below by $\min\{i-1, j-1\}(m-1)^3 D_{\min}$, where as usual $D_{\min}$ denotes the least time required by a vertex in the precedence graph to execute.

It follows that

$$t(i, j) \geqslant \max\{t(i-1, j), t(i, j-1)\} + \min\{i-1, j-1\}(m-1)^3 D_{\min}. \tag{3.7}$$

Taking $t(i, 1) = t(1, j) = 0$, we get from (3.7)

$$t(n, n) \geqslant (p-1)^2(m-1)^3 D_{\min} = \frac{n^3}{p} D_{\min} + O(n^2). \tag{3.8}$$

Since $p = \sqrt{P}$, this strategy produces a time that is far from optimal. Note that even if nodes in the last row and column of a submatrix are allowed to execute whenever all other nodes on the processor are in a wait state, (3.8) still gives a worst-case bound for the scheduling strategy.

The nice results of Theorems 3.1 and 3.2 are in part due to the fact that we assign nodes of the computational network, not vertices of the precedence graph. This forces a systematic assignment of the latter that works well when the former are balanced. Since vertices represent a finer granularity of computation than nodes, we can in principle speed up the computations by assigning at this level. But a simple load-balancing strategy will not necessarily work, as is shown in the following theorem.

THEOREM 3.4. *There is a way to evenly distribute the vertices of the precedence graph for the $n \times n$ Cholesky algorithm on n processors so that, even with instantaneous communication between all processors, the computation time is $O(n^3)$.*

Proof. Consider the precedence graph G of the computation given in Figure 3, and write down the $n^3/3+O(n^2)$ vertices of the graph in order of their distance from the initial vertex, $(1,1,1)$ (i.e., in order of level sets of the graph). Assign (roughly) $n^2/3$ vertices to each processor, without partitioning level sets among processors. We show that this assignment produces time $O(n^3)$.

Denote the level sets by

$$L_m=\{(i,j,k)\in G: i+j+k=m+3\}, \qquad m=0,\dots,3n-3.$$

Now for $m\leqslant n$, the number of elements in level set L_m is

$$|L_m|=(m+1)+(m-2)+\cdots+(\mathrm{rem}(m,3)+1)=\frac{m^2}{6}+O(m),$$

where $\mathrm{rem}(m,3)$ denotes the remainder function for division by 3. This accounts for $n^3/18+O(n^2)$ vertices. Consider the events on the corresponding $n/6$ processors.

Suppose the vertices on a given processor are ordered so that all those in L_m compute before any in L_{m+1}. Then, since there are at most $n^2/6+O(n)$ vertices in the last level set on the processor, and roughly $n^2/3$ vertices in the processor, no vertex in the last level set in the processor (and thus no vertex in the next processor) can execute until time greater than $n^2/6+O(n)$ later than the first vertex in the processor executes. This delay repeated $n/6$ times gives a lower bound of $n^3/36+O(n^2)$ on the execution time of the graph on the n processors. ■

4. BLOCK TORUS ASSIGNMENT, OPTIMAL SCHEDULING, AND LOWER BOUNDS

In the last section we considered scheduling algorithms that gave good results with rather general assignment strategies. No assumptions were made about how the network of processors were connected, beyond the restriction, stated in the introduction, that adjacent nodes must lie on adjacent processors. In this section we shall consider the problem of optimal scheduling on specific networks of processors with specific assignment of nodes.

First we must clear up a potential source of confusion. In the last section we spoke of scheduling nodes in the network. In this section it will be more convenient to speak of executing vertices of the precedence graph. However, the reader should keep in mind that the distinction is purely terminological. At any given time at most one vertex associated with a node is ready to be executed, and when it is executed, we say that we have *scheduled the node* or *executed the vertex*.

In this section we shall use nearly optimal scheduling algorithms to derive lower bounds for the time it takes to run the algorithm in Figure 2 on a linear array of processors and on a square grid of processors. These bounds are sharper than those in (2.2) and (2.3). The results are summarized at the end of the section in Table 1. Sample times for various algorithms are given in Table 2 for the case of n processors and $\sqrt{n}$ processors. The scheduling algorithms we suggest here could be implemented on a torus or hypercube or, for the last algorithm, on a ring of processors.

Square Grids of Processors

We shall assume that we have a $p \times p$ grid of processors (so that the total number of processors is $P = p^2$) with nearest-neighbor connections. We shall further assume that the western boundary of the grid is connected to the eastern and the northern boundary to the southern, so that topologically the connections form a torus (Figure 4).

There are two natural strategies for assigning nodes to this configuration of processors. First we can partition the nodes into $m \times m$ blocks, where $m = \lceil n/p \rceil$, and assign each block to a processor in the natural ordering. With this block assignment, processor (i, j) will contain nodes $(m[i-1]+k, m[j-1]+l)$ $(k, l = 1, \ldots, m)$. Second, we may take the computational network in Figure 1 and wrap it around the torus. With this *torus assignment* processor (i, j) will contain nodes $(i + kp, j + lp)$ $(k, l = 0, \ldots, w-1)$, where $w = \lceil n/p \rceil$ (Figure 4).

The following general assignment strategy includes the two described above. Let w and m be integers such that $n = wmp$ (this implicit restriction on n avoids messy boundary conditions, but does not significantly affect the applicability of the results). Partition the computational network into $m \times m$ blocks of nodes, and assign the *blocks* to processors using torus assignment. Then block assignment corresponds to taking $w = 1$, and torus assignment to taking $m = 1$. We shall call the general assignment strategy *block-torus assignment*.

We begin our analysis of block-torus assignment by describing a scheduling algorithm that is nearly optimal under the assumptions that all vertices in the precedence graph take the same time D to execute and that communication is instantaneous. Actually we shall describe a scheduling algorithm for an augmented precedence graph and then show that if the processors simply remain idle when one of the extra vertices is scheduled, the result is nearly optimal for the original graph.

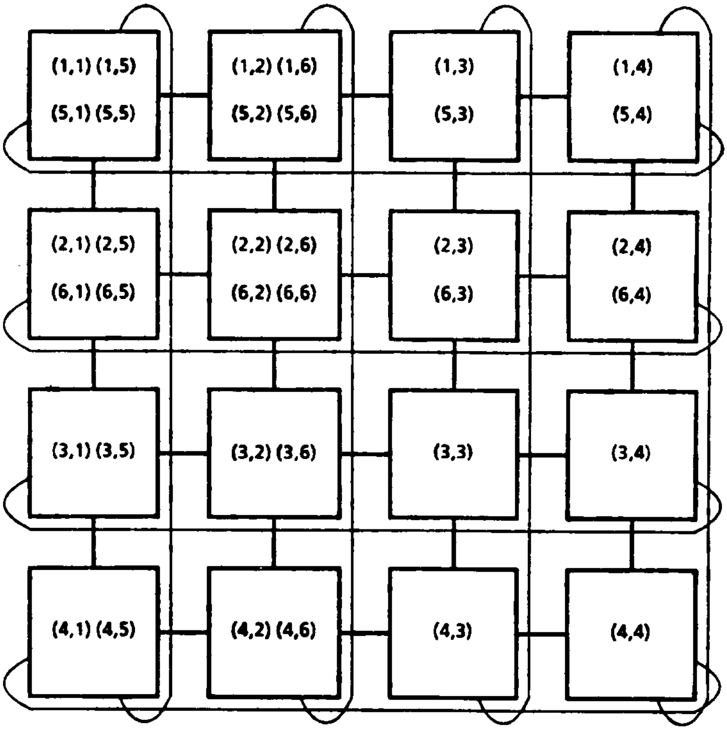


FIG. 4. Torus assignment.

The augmented graph is obtained by adding vertices as follows. For $k = 1, 2, \ldots, mp$ we add vertices (i, j, k) with $1 \leqslant i, j < k$. For $k = mp + 1, mp + 2, \ldots, 2mp$ we add vertices (i, j, k) with $mp + 1 \leqslant i, j < k$. And so on, until for $k = (w - 1)mp + 1, \ldots, wmp$ we add vertices (i, j, k) with $(w - 1)mp + 1 \leqslant i, j < k$. Algorithmically this amounts to starting the kth wave of the Cholesky algorithm with the $(imp + 1, imp + 1)$ element, where $i = \lfloor k/mp \rfloor$.

The crucial feature of this expansion is that block-torus assignment places mostly "real" nodes on the (p, p) processor; no matter what the scheduling strategy for the expanded graph, the (p, p) processor does almost the same work that it would have to do to execute the original precedence graph. The (p, p) processor is the last to finish executing in any scheduling strategy, since, under our restriction that $n = wmp$, vertex (n, n, n) is always assigned to processor (p, p). It follows that if we can devise a scheduling algorithm

```
for l=1 to wm loop
  for r=1 to w loop
    for s=1 to w loop
      for k=(l-1)p+1 to lp loop
        execute all the vertlces ln
        $B(i, j, r, s, k)$ columnwlse;
      end loop;
    end loop;
  end loop;
end loop;
```

FIG. 5. Scheduling algorithm for processor (i, j).

that

(1) activates the (p, p)-processor in the shortest possible time and then

(2) never allows the (p, p) processor to rest until it has processed its last vertex,

then the algorithm will be nearly optimal. We shall now describe a scheduling algorithm that nearly achieves both desiderata.

Let the $m \times m$ blocks on the (i, j) processor be indexed by r (row) and s (column). Let $B(i, j, r, s, k)$ be the set of vertices belonging to wave k that lie in block (r, s) on the (i, j) processor. Then the scheduling algorithm for the (i, j) processor is given in Figure 5.

Some illustrations will show how the algorithm works. The waves are divided into wm clusters G_l of p waves with indices $(l-1)p+1, \ldots, lp$. These clusters traverse the processors as illustrated in Figure 6(a)–(e), where we take $w = 3$, $m = 4$, and $p = 5$. The small blocks represent the basic 4×4 blocks of the matrix, and the double lines represent where the blocks wrap over the torus. Thus the small blocks lying in the same position within the larger blocks are all assigned to the same processor. In the figures, the first of the numbers in a block is the wave k (1 through n) being executed, and the second is the column (1 through m) within the block currently being processed. We assume here that the vertices are executed in lockstep, firing in sequence beginning at the first clock pulse after the data for the column become available.

Figure 6(a) shows the wave cluster G_1 getting started (it is instructive to work through the process a step at a time up to this point). Figure 6(b) shows G_1 wrapping around the torus. Note that the leading edge of the cluster is complementary to the trailing edge, so that all processors are active. Figure 6(c) shows how G_1 passes down across the torus boundary, and Figure 6(d) shows how G_1 leaves the matrix while G_2 follows it in. The clusters G_2 and

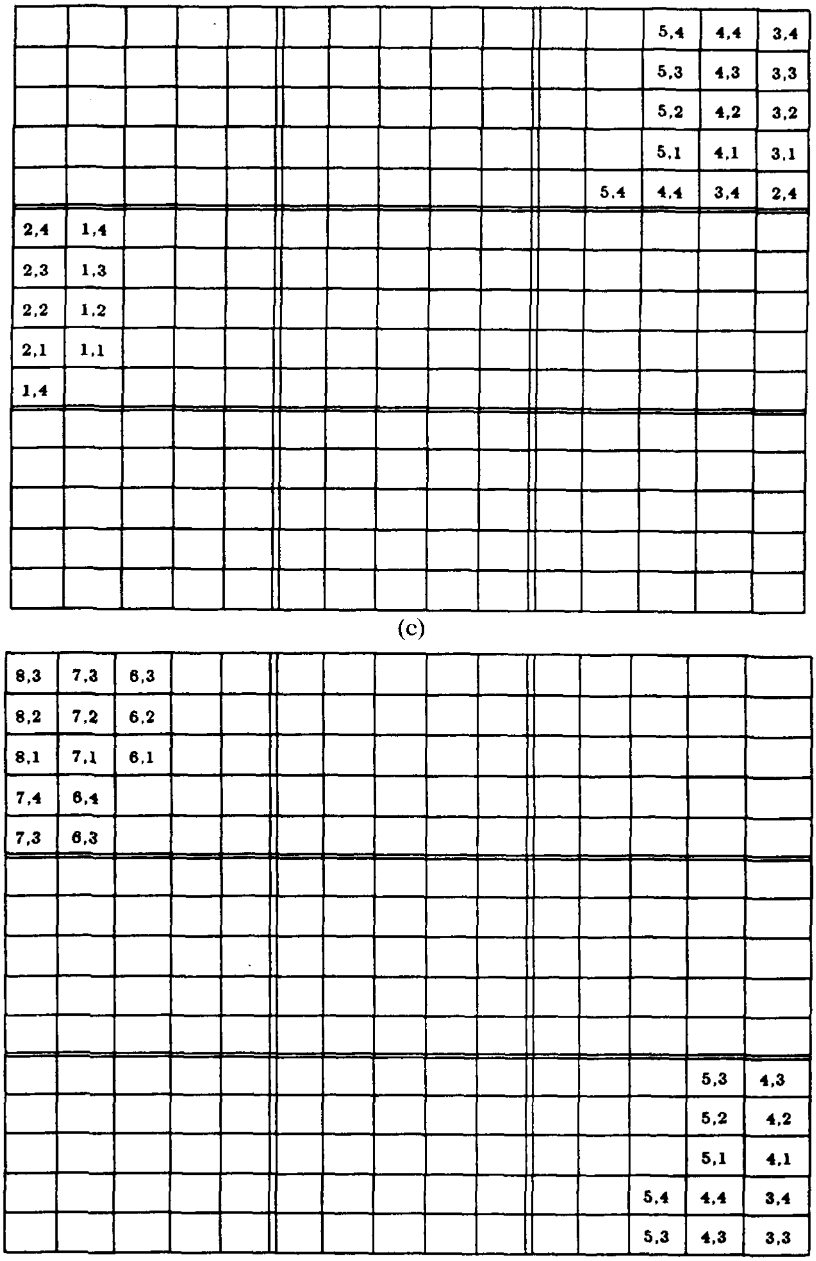
5,3	4,3	3,3	2,3	1,3										
5,2	4,2	3,2	2,2	1,2										
5,1	4,1	3,1	2,1	1,1										
4,4	3,4	2,4	1,4											
4,3	3,3	2,3	1,3											

(a)

			5,2	4,2	3,2	2,2	1,2							
			5,1	4,1	3,1	2,1	1,1							
		5,4	4,4	3,4	2,4	1,4								
		5,3	4,3	3,3	2,3	1,3								
		5,2	4,2	3,2	2,2	1,2								

(b)

FIG. 6. Snapshot at time (a) $19m$, (b) $30m$, (c) $68m$, (d) $191m$, (e) $671m$.



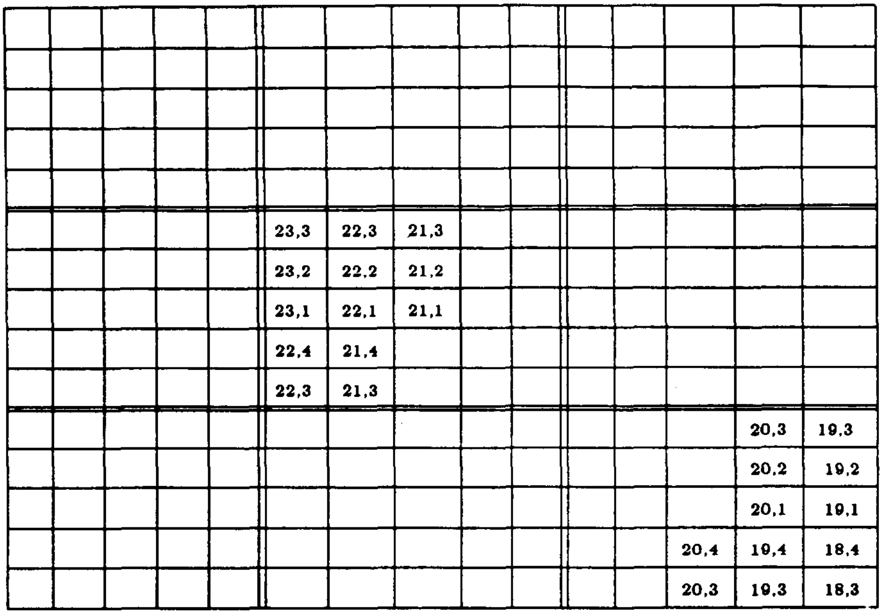
												5,4	4,4	3,4
												5,3	4,3	3,3
												5,2	4,2	3,2
												5,1	4,1	3,1
											5,4	4,4	3,4	2,4
2,4	1,4													
2,3	1,3													
2,2	1,2													
2,1	1,1													
1,4														

(c)

8,3	7,3	6,3												
8,2	7,2	6,2												
8,1	7,1	6,1												
7,4	6,4													
7,3	6,3													
													5,3	4,3
													5,2	4,2
													5,1	4,1
												5,4	4,4	3,4
												5,3	4,3	3,3

(d)

Fig. 6. (*Continued*)



					23,3	22,3	21,3							
					23,2	22,2	21,2							
					23,1	22,1	21,1							
					22,4	21,4								
					22,3	21,3								
													20,3	19,3
													20,2	19,2
													20,1	19,1
												20,4	19,4	18,4
												20,3	19,3	18,3

(e)

FIG. 6. (*Continued*)

G_3 are processed in the same way. However, as G_4 terminates, G_5 starts in the (6,6) block [Figure 6(e)]. In all cases the processors do not stutter—they are ever active.

To see how near we come to the first desideratum, let us first compute a lower bound on how long it takes the first wave to reach the (p, p) processor. Consider the interval that must elapse between the time the first wave arrives at the (i, i) processor and the time that it arrives at the $(i+1, i+1)$ processor. First it must traverse the (i, i) processor, which requires m^2D time. Then two vertices must be executed, one in the southwest corner of the $(i, i+1)$ processor and the other in the northeast corner of the $(i+1, i)$ processor. Since these can be done in parallel, they add only D to the interval. Thus it takes at least $(m^2+1)D$ units of time to traverse the (i, i) processor. Since $p-1$ processors on the diagonal must be traversed, we find that the time required to activate the (p, p) processor is bounded below by

$$S_D = (p-1)(m^2+1)D. \tag{4.1}$$

On the other hand, it is easily seen that the scheduling algorithm requires time

$$(p-1)(m^2+m)D. \tag{4.2}$$

to activate the (p, p) processor. The ratio of (4.2) to the lower bound (4.1) is

$$\frac{1+\dfrac{1}{m}}{1+\dfrac{1}{m^2}}, \tag{4.3}$$

which attains its maximum of 1.2 for $m = 2,3$. Thus as far as the first desideratum is concerned, the scheduling algorithm is nearly optimal.

Although we have constructed our scheduling algorithm so that the second desideratum is satisfied, we have yet to take into account work done on "false" vertices. We begin our investigations by computing how much work must be done on the (p, p) processor when the original Cholesky algorithm is executed. It is easiest here to think in terms of block eliminations. We distinguish three sources of work.

(1) Consider the block (r, s) in processor (p, p). If this block is on or above the main diagonal of the matrix, then it must be eliminated by the $rp-1$ blocks above it, each of which generates m^3D work. If the block is below the main diagonal, then there are $sp-1$ such block eliminations. Summing over all blocks in the processor gives a total of

$$m^3D\sum_{r=1}^{w}\sum_{s=1}^{w}(\min\{r,s\}\,p-1). \tag{4.4}$$

(2) Each of the off-diagonal blocks must eliminate itself at a cost of $m^2(m+1)D/2$ units of work. Hence the contribution from this source is

$$\frac{m^2(m+1)}{2}D(w^2-w). \tag{4.5}$$

(3) Each of the w diagonal blocks must eliminate itself, for a cost of

$$\frac{m(m+1)(2m+1)}{6}Dw. \tag{4.6}$$

Summing (4.4)–(4.6), we get a lower bound for the time required by the (p, p) processor, independent of scheduling algorithm:

$$T_D = \left[\frac{m^3w^3p}{3} + \frac{m^3w^2p}{2} + \frac{m^3wp}{6} - \frac{m^3w^2}{2} - \frac{m^3w}{6} + \frac{m^2w^2}{2} + \frac{mw}{6}\right]D. \tag{4.7}$$

Now consider the work done by the (p, p) processor when the additional vertices are included. The work represented by (4.4) is unaffected. However, the factor $m^2(m+1)/2$ in (4.5) must be replaced by m^3, and the factor $m(m+1)(2m+1)/6$ in (4.6) by m^3. This gives an excess work of

$$\left[\frac{m^3w(w-1)}{2} + \frac{2m^3w}{3}\right]D + O(m^2w^2). \tag{4.8}$$

Comparing (4.8) with (4.7), we see that if w increases as n increases, the amount of extra work becomes asymptotically negligible. However, when w is fixed, so that m increases linearly with n, (4.8) approaches a fixed proportion of (4.7), which can be quite large. For example, when $w = p = 1$, the ratio of (4.7)+(4.8) to (4.7) approaches three; in other words, if we wait for false vertices, our scheduling algorithm takes three times longer than it needs to. However, this ratio drops sharply with increasing p or w. For example, if $p = 1$ and $w = 5$, the ratio is 1.32. For $p = 5$ and $w = 1$, it is 1.15. Moreover, if we run the scheduling algorithm on the original graph with data-flow synchronization rather than lockstep, the ratios will be even lower, since the processor time devoted to false vertices can now be used profitably on real ones.

To recapitulate, we have computed a lower bound (4.7) on the execution time by computing how much work the (p, p) processor must do [the activation time (4.1) is asymptotically negligible]. Moreover, except for small p and w, this bound is nearly attained by our scheduling algorithm. Encouraged by this result, we now proceed to compute lower bounds on the interprocessor communication time in much the same way.

We shall assume that the times required for internal sends and receives are included in the node execution time D. Let R be the *additional* time required to receive a message coming from another processor. Then the time spent by the (p, p) processor receiving external messages may be counted as above:

(1) Consider the block (r, s). Each block elimination requires $2m^2$ numbers from other processors. Therefore, the total amount of input needed for the eliminations in (4.4) is

$$2m^2 \sum_{r=1}^{w} \sum_{s=1}^{w} (\min\{r, s\}\, p - 1). \tag{4.9}$$

(2) Each of the off-diagonal blocks must eliminate itself, which requires $m(m+1)D/2$ units of input. Hence the contribution from this source is

$$\frac{m(m+1)}{2}(w^2 - w). \tag{4.10}$$

Summing (4.9)–(4.10) and multiplying by R, we get a lower bound for the time taken by interprocessor receptions:

$$T_R = \left[\frac{2m^2w^3p}{3} + m^2w^2p - \frac{3m^2w^2}{2} + \frac{m^2wp}{3} - \frac{m^2w}{2} + \frac{mw^2}{2} - \frac{mw}{2}\right]R. \tag{4.11}$$

Although the time for the sends is computed in the same way, the argument is more tedious and the formulas more complicated, since nodes on the boundary of the network do not pass on information. However, as p and w increase, they approach the number of receives. Hence we shall take (4.11) as characterizing the interprocessor communication time.

It is instructive to examine how the bounds behave as n becomes large. First taking $w = 1$ (pure blocking) and making the substitution $m = n/p$, we get the following asymptotic expressions:

$$T_D \simeq n^3\left[\frac{1}{p^2} - \frac{2}{3p^3}\right]D, \qquad T_R \simeq n^2\left[\frac{2}{p} - \frac{2}{p^2}\right]R. \tag{4.12}$$

On the other hand, if we take $m = 1$ (pure wrapping), we get

$$T_D \simeq \frac{n^3}{3p^2}D, \qquad T_R \simeq \frac{2n^3}{3p^2}R. \tag{4.13}$$

From this we see that when we pass from pure blocking to pure wrapping, we decrease the computation time at the cost of increasing the communication time. In principle one could use the lower bound to determine approximate break-even points. However, it must be kept in mind that if $n \gg p$, then the algorithm given here is not the best. For pure blocking, a better algorithm would be one that treats the blocks as units, thus reducing internal communication among the nodes. Here the above analysis holds when D is reinterpreted as the cost of one update of one matrix element and does not include the overhead of internal transmission. For pure torus wrap a better algorithm would be one that recognizes that information passing through a processor will be used by the same processor later. The redundant sends and receives can be eliminated, and we refer to this as the *compact* algorithm. The analysis of such algorithms is similar to the above analysis, and equally tedious. The results are presented in Table 1.

Linear Arrays of Processors

The techniques in this case are analogous to the one for square grids of processors, and we shall only sketch the development. Assume that we have P processors, and let w and m be integers such that $n = wmP$. We partition the matrix into blocks of m columns and assign them sequentially to the processors, wrapping them around to the beginning w times.

There is a nearly optimal scheduling algorithm for the linear array that is analogous to the one for the square grid. Groups of P waves are sent through the processors. As one group falls off the end of the matrix, the next starts in the appropriate place. The algorithm requires few false vertices to keep the Pth processor busy. Hence we shall not greatly underestimate the running time for this array by counting the time required by the Pth processor.

To calculate this work, consider the rPth block of m columns. For $j = 1,\dots, m$ the $(rP - 1)m + j$ column must be eliminated by all the preceding columns and by itself. The cost of being eliminated by the lth column is $(n - l + 1)D$. Hence the total amount of work done by the Pth processor is

$$\frac{T_D}{D} = \sum_{r=1}^{w} \sum_{j=1}^{m} \sum_{l=1}^{(rP-1)m+j} (n - l + 1)$$

$$= \frac{m^3w^3P^2}{3} + \frac{m^3w^2P^2}{4} - \frac{m^3w^2P}{4}$$

$$- \frac{m^3wP^2}{12} + \frac{m^3wP}{4} - \frac{m^3w}{6} + \frac{m^2w^2P}{2} + \frac{mw}{6}. \tag{4.14}$$

TABLE 1

NUMBER OF SEQUENTIAL COMPUTES AND RECEIVES FOR VARIOUS ALGORITHMS

Square grid	$N_D = \frac{m^3w^3p}{3} + \frac{m^3w^2p}{2} + \frac{m^3wp}{6} - \frac{m^3w^2}{2} - \frac{m^3w}{6} + \frac{m^2w^2}{2} + \frac{mw}{6}$
Original algorithm	$N_R = \frac{2m^2w^3p}{3} + m^2w^2p + \frac{m^2wp}{3} - \frac{3m^2w^2}{2} - \frac{m^2w}{2} + \frac{mw^2}{2} - \frac{mw}{2}$
Compact algorithm	$N_R = m^2w^2p + m^2wp - m^2w^2 - m^2w$
Linear array	$N_D = \frac{m^3w^3P^2}{3} + \frac{m^3w^2P^2}{4} - \frac{m^3wP^2}{12} - \frac{m^3w^2P}{4} + \frac{m^3wP}{4} - \frac{m^3w}{6} + \frac{m^2w^2P}{2} + \frac{mw}{6}$
Original algorithm	$N_R = \frac{m^2w^3P^2}{3} + \frac{m^2w^2P^2}{4} - \frac{m^2wP^2}{12} - \frac{m^2w^2P}{2} + \frac{m^2wP}{2} - \frac{m^2w}{2} + \frac{mw^2P}{4} + \frac{mwP}{4} - \frac{mw}{2}$
Compact algorithm	$N_R = \frac{m^2w^2P^2}{2} - \frac{m^2w^2P}{2} + \frac{m^2wP}{2} - \frac{m^2w}{2} + \frac{mwP}{2} - \frac{mw}{2}$

The extra time for receptions for the Pth processor may be calculated by dropping the self-elimination terms in the summation on l in (4.14) and dropping the summation on j. This gives

$$\frac{T_R}{R} = \sum_{r=1}^{w} \sum_{l=1}^{(rP-1)m} (n-l+1)$$

$$= \frac{m^2w^3P^2}{3} + \frac{m^2w^2P^2}{4} - \frac{m^2w^2P}{2} - \frac{m^2wP^2}{12}$$

$$+ \frac{m^2wP}{2} - \frac{m^2w}{2} + \frac{mw^2P}{4} + \frac{mwP}{4} - \frac{mw}{2}. \tag{4.15}$$

Again it is instructive to consider the extreme cases. If we take $w=1$, then we get the following asymptotic expressions:

$$T_D \simeq \frac{n^3}{2P}D, \qquad T_R \simeq \frac{n^2}{2}R. \tag{4.16}$$

If we take $m=1$, then we get the following expressions:

$$T_D \simeq \frac{n^3}{3P}D, \qquad T_R \simeq \frac{n^3}{3P}R. \tag{4.17}$$

Again we find a tradeoff between computation and communication. However, the same caveat applies in interpreting these bounds as interpreting the bounds for the grid of processors: when $n \gg P$ the compact algorithm is better than the one analyzed in this section.

We summarize these scheduling results in Table 1, which presents the computation and communication costs for the scheduling algorithms presented in this section. These results are expressed in terms of the blocksize m, the number of torus wraps w, and the number of processors p or P. In Table 2, we compute the leading-order terms of these costs for some example configurations: n or $\sqrt{n}$ processors with the matrix either fully blocked ($w=1$) or fully wrapped ($m=1$). For n processors and for $\sqrt{n}$ processors, the best of these algorithms is full wrapping on a square grid.

TABLE 2
NUMBER OF SEQUENTIAL COMPUTES AND RECEIVES FOR VARIOUS ALGORITHMS WITH A FIXED NUMBER OF PROCESSORS[a]

	N_D	N_R	N_R^{compact}
n processors			
Square grid, full blocking	$1n^2$	$2n^{3/2}$	$2n^{3/2}$
Square grid, full wrapping	$\frac{1}{3}n^2$	$\frac{2}{3}n^2$	$1n^{3/2}$
Linear array	$\frac{1}{2}n^2$	$\frac{1}{2}n^2$	$\frac{1}{2}n^2$
$\sqrt{n}$ *processors*			
Square grid, full blocking	$1n^{5/2}$	$2n^{7/4}$	$2n^{7/4}$
Square grid, full wrapping	$\frac{1}{3}n^{5/2}$	$\frac{2}{3}n^{5/2}$	$1n^{7/4}$
Linear array, full blocking	$\frac{1}{2}n^{5/2}$	$\frac{1}{2}n^2$	$\frac{1}{2}n^2$
Linear array, full wrapping	$\frac{1}{3}n^{5/2}$	$\frac{1}{3}n^{5/2}$	$\frac{1}{2}n^2$

[a] Leading term only.

REFERENCES

1 R. P. Brent and F. T. Luk, Computing the Cholesky factorization using a systolic architecture, Technical Report *TR* 82-521, Dept. of Computer Science, Cornell Univ., 1982.

2 R. P. Brent and F. T. Luk, A systolic architecture for almost linear-time solution of the symmetric eigenvalue problem, Technical Report TR 82-525, Dept. of Computer Science, Cornell Univ., 1982.

3 Robert Funderlic and Alan Geist, Torus data flow for parallel computation of missized matrix problems, manuscript, Oak Ridge National Lab., Oak Ridge, Tenn., 1985.

4 Alan George, Michael T. Heath, and Joseph Liu, Parallel Cholesky factorization on a multiprocessor, Technical Report ORNL-6124, Oak Ridge National Lab., Oak Ridge, Tenn., 1985.

5 Kai Hwang and Faye A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984.

6 Ilse C. F. Ipsen, Youcef Saad, and Martin H. Schultz, Complexity of dense linear system solution on a multiprocessor ring *Lin. Algebra Appl.* 77:205–239 (1986).

7 S. Y. Kung, VLSI array processors for signal processing, presented at *MIT Conference on Advanced Research in Integrated Circuits*, Cambridge, Mass., 1980, cited in [8].

8 S. Y. Kung, K. S. Arun, D. V. Bhaskar Rao, and Y. H. Hu, A matrix data flow language/architecture for parallel matrix operations based on computational wavefront concept, in *VLSI Systems and Computation* (H. T. Kung, B. Sproull, and G. Steele, Eds.), Computer Science Press, Rockville, MD., 1981, pp. 235–244.

9 Dianne P. O'Leary and G. W. Stewart, Data-flow algorithms for parallel matrix computations, *Comm. ACM* 28:840–853 (1985).

10 Dianne P. O'Leary and G. W. Stewart, On the determinancy of a model for parallel computation, Technical Report 1456, Department of Computer Science, Univ. of Maryland, 1984.

11 Mandayam A. Srinivas, Optimal parallel scheduling of a Gaussian elimination DAG's, *IEEE Trans. Comput.* C-32:1109–1117 (1983).

Received 10 April 1985; revised 26 September 1985