CONF-8508178--1

MATRIX FACTORIZATION ON A HYPERCUBE MULTIPROCESSOR

CONF-8508178--1

DE86 011775

G. A. Geist
and
M. T. Heath

Mathematical Sciences Section
Engineering Physics and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831

* Research sponsored by the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy under contract DE-AC05-84OR21400 with the Martin Marietta Energy Systems, Inc.

[illegible]

Matrix Factorization on a Hypercube Multiprocessor

G. A. Geist and M. T. Heath*

Abstract. This paper is concerned with parallel algorithms for matrix factorization on distributed-memory, message-passing multiprocessors, with special emphasis on the hypercube. We consider both Cholesky factorization of symmetric positive definite matrices and *LU* factorization of nonsymmetric matrices using partial pivoting. We also consider the use of the resulting triangular factors to solve systems of linear equations by forward and back substitutions. Efficiencies of various parallel computational approaches are compared in terms of empirical results obtained on an Intel iPSC hypercube.

1. Introduction. In this paper we study the efficiency of parallel algorithms for matrix factorization on a hypercube multiprocessor. There have been a number of theoretical studies of the parallel complexity of factoring dense matrices on various types of distributed-memory multiprocessors (e.g., [1], [6], [7], [10], [13], [14], [15], [16], [18]), but relatively little actual experience has been reported. Therefore, our study will be largely empirical: we will assess the efficiency of various strategies by numerical experiments on a hypercube, which is a typical distributed-memory multiprocessor.

To avoid unnecessary complications, we initially restrict our attention to the simplest case, that of computing the Cholesky factorization

$$A = L\,L^T$$

where A is a symmetric, positive definite matrix of order n and L is lower triangular. Many of our conclusions, however, are equally applicable to other matrix factorizations. Later in the paper we consider *LU* factorization by Gaussian elimination with partial pivoting for nonsymmetric matrices.

* Mathematical Sciences Section, Oak Ridge National Laboratory, P.O. Box Y, Oak Ridge, Tennessee 37831. Research supported by the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems Inc.

Matrix factorization is an interesting case study for parallel implementation because it has two essential requirements that tend to inhibit parallel efficiency:

1. serial precedence constraints must be imposed on some operations, and
2. global communication is required.

The first requirement means that although many of the computational operations can take place concurrently, there are some operations that must occur in a strict sequential order. The second requirement means that some results computed by a given processor must be made available to all of the other processors. Both of these requirements tend to cause delays in which some processors must wait for necessary prior results to be computed or communicated by other processors.

A detailed study of parallel Cholesky factorization on a shared-memory multiprocessor is contained in [11]. Global communication is a relatively minor issue in such an environment because global data access is available through the common memory (although memory contention can still be a problem). Therefore, [11] focuses primarily on the effect of precedence constraints and subtask work profiles on concurrency and load balancing. It was found that among the six ways of arranging the triple nested loop that defines the Cholesky algorithm, the method having the best combination of concurrency and dynamic load balance for a medium-grain parallel implementation is the column-oriented Cholesky algorithm denoted by *jki* (using the notation of [5]). By medium-grain we mean for this particular problem that the basic units of computation are subtasks of computational complexity $O(n)$ arithmetic operations. A medium-grain approach is most appropriate when the order of the matrix significantly exceeds the number of processors available.

Developing a medium-grain parallel implementation of Cholesky factorization for a distributed-memory multiprocessor is more complicated. In a distributed-memory environment, we must usually be content with a static load balance that is determined in advance of the computation, in contrast to a dynamic load balance that is easily implemented in shared memory by means of a pool of tasks. Even more serious, in a distributed-memory system, communication is accomplished by passing messages among processors, and thus the necessity of global communication becomes a critical factor affecting computational efficiency. Our primary objectives in the present paper are to examine various ways of mapping the matrix onto the processors, to study their effects on load balancing and communication, and to determine how the communication requirements can best be met.

2. The Basic Algorithm.

The *jki* formulation of the Cholesky algorithm is as follows.*

* We number components from 0 to n-1 for compatibility with array indexing in our codes, which are written in the C programming language. This is also consistent with the standard numbering of nodes in a hypercube and simplifies some of the formulas given below.

```
for $j = 0$ to $n-1$ do
begin
    for $k = 0$ to $j-1$ do
        for $i = j$ to $n-1$ do
            $a_{ij} = a_{ij} - a_{ik} * a_{jk}$
    $a_{jj} = \sqrt{a_{jj}}$
    for $k = j+1$ to $n-1$ do
        $a_{kj} = a_{kj} / a_{jj}$
end
```

In the inner loop of this algorithm, column j is modified by each previously computed column k, an operation which for convenience we denote by *cmod* (j, k). After column j is completely modified by all previous columns, it is then divided by the square root of its diagonal element to produce the final column j of L. We refer to the latter operation as *cdiv* (j). Using this notation, the algorithm can be rewritten as

```
for $j = 0$ to $n-1$ do
begin
    for $k = 0$ to $j-1$ do
        cmod $(j, k)$
    cdiv $(j)$
end
```

We will view the *cmod* (j, k) and *cdiv* (j) operations as the basic subtasks to be scheduled on the processors. This means that we will not try to exploit parallelism within the *cmod* and *cdiv* operations, although in a finer-grained parallel implementation this could be done. We note that the *cdiv* operations must take place in sequential order, but that once a given column is completed, it can be used to modify subsequent columns in any order, or even to modify multiple columns concurrently. We will use this property to overlap the *cmod* operations, thereby attaining a high degree of processor utilization.

An implementation of this approach for a shared-memory system is given in [11]. When a processor is assigned the task of computing a given column of L, it first performs all of the necessary modifications by previous columns, then performs the column division. All of the required previous columns are directly accessible to each processor because they are stored in the common global memory. Moreover, by sharing the pool of tasks among all of the processors and assigning columns to processors dynamically, the computational load tends to be automatically balanced among the processors. In a message-passing, distributed-memory system, the pool of tasks cannot easily be shared, so the load balance is entirely dependent on the static assignment of columns to processors. Since all memory is private, each column computed by a given processor must be explicitly sent to the other processors that need it for updating the columns they have been assigned. Thus, in general, global communication is required, and this may or may not be well supported by a particular interconnection network.

Let us assume that each processor has been assigned a subset of the columns of L to compute and that the corresponding columns of the original matrix A

reside in the local memory of the processor. Later we will discuss ways to make this assignment effectively. Then the program that runs on each processor in the message-passing version of the column-Cholesky algorithm is as follows:

```
for $j = 0$ to $n-1$ do
begin
    if col $j$ is one of my cols then cdiv($j$)
    communicate (col $j$)
    for all of my cols $k > j$ do cmod($k$, $j$)
end
```

In this algorithm the procedure *communicate* either sends or receives col j, depending on whether the processor calling *communicate* is responsible for computing col j. In either case, after returning from the *communicate* procedure, every processor now has a copy of col j and uses it to modify any of its columns that may be affected. Note that the *cdiv* operation on a given column will be done only after all of its necessary modifications have taken place. Thus, the proper synchronization of the algorithm is implicit in the flow of completed columns through the network.

For simplicity, we stated the message-passing Cholesky algorithm in a synchronous form: there is a strict alternation between computation and communication. It is possible to overlap the two, however, in an effort to mask some of the communication cost with computation. The philosophy here is to send out results at the earliest possible time in the hope that this will minimize any subsequent waiting for them. In particular, as soon as any column has had all of its modifications completed, the *cdiv* operation should be carried out immediately so that the broadcast of the resulting column of L can be started. Thus, a test is inserted into the *cmod* loop to detect completion of modifications to any column, in which case, the *cdiv* operation is carried out and the results transmitted before continuing with the remaining *cmod* operations. The effect of this strategy is to pipeline the computation of successive columns.

This overlapping of communication with computation enables the pipelined algorithm to maintain a higher level of processor utilization than the synchronous algorithm. The latter has a dip in utilization for each broadcast communication step. The relative duration of these dips depends on the relative speeds of communication and computation, so the difference in performance on a given machine may or may not be significant. These effects will be evident in the experimental results given in section 5.

At this level of specification, the algorithm is independent of the specific type of message-passing multiprocessor architecture. Only the details of the *communicate* procedure depend on the communication supported by a particular interconnection network. We will illustrate below with appropriate communication procedures for hypercube and ring networks. In addition to the communication technique, the other main feature affecting performance of the algorithm is the mapping of the matrix onto the processors. In the sections 4 and 5 we take up the issues of mapping and communication, after discussing our experimental methodology in section 3.

3. Experimental Methodology. The particular type of distributed-memory multiprocessor we consider is the binary hypercube (see, e.g., [19]). We chose the hypercube architecture for our implementation because it is available (hypercubes have been produced by several organizations, including commercial manufacturers), flexible (many other processor interconnection network topologies can be embedded in a hypercube), and reasonably representative of message-passing multiprocessors in general. We give expermental results obtained on an Intel iPSC hypercube.

The individual processors of a hypercube are usually referred to as node processors, or simply nodes. The nodes are numbered $0, \ldots, p-1$, where $p = 2^d$ and d is the dimension of the hypercube. There is an additional processor, usually referred to as the host, which serves as the user interface to the hypercube, downloading compiled code and problem data to the nodes, and receiving results back for display to the user. The host may or may not take an active part during computations on the node processors. In many hypercubes, including the iPSC, the operating system provides automatic routing of messages between arbitrary nodes, whether or not they are directly connected by the network.

Perhaps the most important parameter characterizing any message-passing multiprocessor is the ratio of computation speed to communication speed. The iPSC has relatively high startup cost for communications, and sends messages in relatively large packets (1024 bytes), so that fine-grained algorithms, which do relatively little computation between communications and send relatively small messages, tend to perform poorly. Moreover, reliable communication between the host and nodes is many times slower still, so it is impractical for the host to participate in computations on the hypercube. These considerations have affected our choice of algorithms and the resulting performance, as will be seen below.

Our conclusions throughout the paper are based on numerical experiments. Since timing data for a particular machine have little universal meaning, we state our results in terms of parallel efficiency. The usual definition of parallel efficiency is

$$\text{efficiency} = \frac{\textit{speedup}}{p} = \frac{T_1 / T_p}{p},$$

where p is the number of processors used, T_1 is the execution time for the best sequential algorithm on one processor, and T_p is the execution time for the parallel algorithm on p processors. A practical difficulty with this definition is that most of our test problems are much too large to solve using only the memory of a single processor, and thus we cannot obtain directly the value of T_1 by numerical experiment. We have therefore estimated a value for T_1 based on the measured peak execution rate of an individual processor for the equivalent serial computation.

We use the rate for the "equivalent serial computation" rather than the "best serial code" for the following reason. The best serial code available often has had the benefit of years of fine tuning, often with extensive source-level code optimization such as loop unrolling, efficient use of registers, etc., and perhaps even with some modules coded directly in assembler language. It is impractical to spend the time and energy to perform the same kind of tuning on an experimental parallel code, yet if one does not, then comparisons with the "best serial code" will appear to yield an unfavorable estimated parallel efficiency. A more realistic approach is to use a serial code that implements the best serial algorithm but with the same level of code optimization as the parallel code with which it is to be compared. This does not mean that we simply run the parallel code on a single processor, because the parallel code will often contain overhead that would not be present in a serial code, and may use algorithms or data structures that are fundamentally less efficient for serial computation.

Although in one sense it is important to establish a fair and realistic serial benchmark against which to measure parallel performance, in another sense the result is simply a rescaling of parallel execution times, with the interpretation in terms of "speedup" or "efficiency" somewhat arbitrary. In a message-passing environment, the true parallel efficiency is largely determined by the ratio of computation to communication speeds, and detailed code optimization in effect changes that ratio. Reported parallel efficiencies should be interpreted in this light. In any case, the main point of our experiments is to compare the effectiveness of various options within a basic parallel algorithm rather than to establish any absolute level of efficiency. To that end, we use the execution time of the equivalent serial computation as a convenient time unit.

As we demonstrate below, there are many choices of strategy in implementing Cholesky factorization on a distributed-memory multiprocessor such as a hypercube. To facilitate systematic testing and fair comparisons, the numerical results we give below were obtained with a single program in which various options are implemented for each major issue we study. For a given series of experimental runs exploring a particular issue, all of the other options are fixed at reasonable values.

4. Mapping. The manner in which the matrix is mapped onto the processors will affect the communication requirements, the degree of concurrency, and the load balance among the processors. We would like to minimize communication, maximize concurrency, and have a uniform work load across the processors. These objectives tend to conflict, however, and so we must weigh the trade-offs among them.

With a column-oriented algorithm, the most natural way to partition the matrix for mapping onto the processors is by vertical strips (i.e., by sets of columns). Other partitionings may be more appropriate for other types of algorithms, but in our experience such partitionings (for example, by "patches" or submatrices [6], [7]) are not competitive when used in a column-oriented algorithm. Thus, we wish to consider ways of mapping columns $0, \ldots, n-1$ of the matrix onto processors $0, \ldots, p-1$, where we assume that $n \gg p$.

Perhaps the most obvious systematic mapping is to map a contiguous block of n / p columns onto each processor; we call this *block* mapping. Another possibility is to assign the columns to the processors in the same manner one would deal cards, assigning one column to each processor and then wrapping back to the beginning with further columns; we call this *wrap* mapping. A general expression that includes both possibilities is given by

$$\text{map column } j \text{ onto processor } \lfloor j \,/\, \mathit{blocksize} \rfloor \bmod p,$$

where *blocksize* is the number of contiguous columns to be assigned to each processor. Wrap mapping is given by *blocksize* =1 and block mapping is given by *blocksize* $= \lceil n \,/\, p \rceil$.

Which of these mappings should give the best performance? After completing the last of its assigned columns, each processor becomes idle for the remainder of the factorization. The block mapping therefore causes the processors containing the earlier blocks to be idle much of the time, whereas the wrap mapping keeps all processors busy as long as possible. Thus, we would expect the wrap mapping to yield much higher concurrency and processor utilization than the block mapping. On the other hand, the block mapping has potentially smaller communication requirements, since each completed column needs to be sent only to higher numbered processors, rather than to all processors as in the wrap mapping.

TABLE 1
Execution time (sec) for Cholesky factorization as a function of blocksize ($p = 32, n = 512$).

blocksize	execution time
1	79.4
2	82.7
4	92.1
8	115.8
16	174.7
random mapping	94.1

Although this tradeoff between concurrency and communication could conceivably go either way depending on the relative speeds of communication and computation, it turns out that even with rather slow communication the poor concurrency of block mapping leads to performance that is uniformly inferior to that of the wrap mapping. This is illustrated in Table 1, which gives execution time as a function of blocksize for a problem having n=512 and p=32. In this case, a pure block mapping corresponds to *blocksize*=16. The results shown use the pipelined version of our algorithm with *bcube* communication, as explained in the next section. Also shown in Table 1 is the average execution time for 10 random mappings of columns to processors (for which the standard deviation was 1.94 sec), confirming that some care in choosing the mapping is worthwhile.

We note that there are other ways of achieving a more or less uniform scattering of columns across the processors (e.g., reflection), but wrapping seems as straightforward and effective as any. All of our remaining numerical experiments use the wrap mapping.

5. Communication.

As we have seen, the Cholesky factorization algorithm requires global communication: upon completing each of its assigned columns, a given processor must in general make its results available to all other processors.

Such a communication pattern is referred to as *broadcasting*. The method used to implement broadcasting, and its resulting efficiency, depend on the details of the underlying interconnection network among the processors. In a bus-based system, for example, broadcasting can usually be accomplished as a single *send* operation since all of the processors are listening to the common bus. (On the other hand, write access to the common bus is necessarily serial; i.e., all processors can receive simultaneously, but only one processor can send at a time.) In a completely connected network (e.g., a crossbar switch), broadcasting is accomplished by $p-1$ separate *send* operations, one to each of the other processors. If some processors are not directly connected by the network, then broadcasting requires that the message be forwarded by intermediate processors in order to reach some destinations. In a ring network, for example, broadcast messages must be propagated around the ring, with a resulting delay of at least $p/2$ communication steps.

A hypercube interconnection network offers a number of possibilities for implementing broadcasting. First, the "diameter" of a hypercube network is relatively small: if $p=2^d$, then the largest distance between any two processors is d, and thus a message never has to be forwarded more than d steps. Moreover, in many hypercubes the operating system automatically forwards messages as necessary to reach arbitrary destination nodes. Thus, one option for implementing broadcasting is simply to write the program as though the network were completely connected, with a separate *send* from the source node to each destination node. For convenience, we will refer to this option as *bcast*.

A second possibility is to propagate broadcasts through the hypercube along an embedded ring. A suitable ordering of the nodes into a ring is provided by a binary reflected Gray code [17], which has the property that consecutive nodes in the ring are physically connected in the hypercube. This approach takes relatively little advantage of the interconnections in the hypercube network, and the diameter of the ring is 2^{d-1}. Nevertheless, a ring can be very effective for multiple broadcasts if they are suitably pipelined. We will refer to this option as *ring*.

Finally, it is easy to design a strategy for broadcasting that takes full advantage of the recursive structure and small diameter of the hypercube. This approach is based on embedding a minimal spanning tree in the hypercube network, rooted at the source node of the broadcast. The root node of the broadcast sends the message to all of its neighbors, who in turn send the message to all of their neighbors who have not already received the message, etc., until after d stages all nodes have received the message. An example is shown in Fig. 1, in which the root is assumed to be node 0. We will refer to this option as *bcube*. An implementation in C with an arbitrary node as root is as follows (for definiteness, we use the communication primitives of the Intel iPSC hypercube):

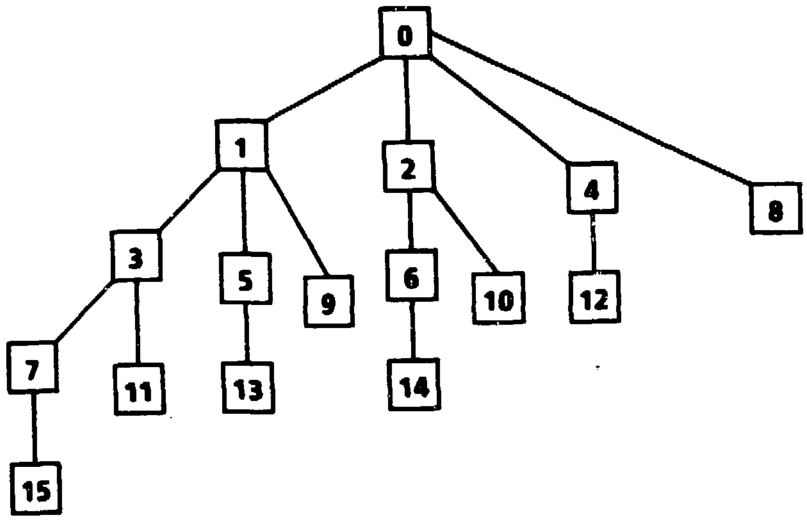

FIG. 1. *Spanning tree for hypercube broadcast.*

```
bcube ( vec, bytes, p, root, ci, pid, msgtype )
       char *vec;
       int bytes, p, root, ci, pid, msgtype;
/*
 *  Broadcast array vec, of length bytes, to all p processors using a
 *  minimum spanning tree with given root.  Channel identifier is ci,
 *  process id is pid, and message is of type msgtype.
 */
{
       int me, cnt, node;

       me = mynode()↑root;
       p /= 2;
       if ( me < p )
       {
              if ( p != 1 ) bcube ( vec, bytes, p, root, ci, pid, msgtype );
              send ( ci, msgtype, vec, bytes, (me+p)↑root, pid );
       }
       else recvw ( ci, msgtype, vec, bytes, &cnt, &node, &pid );
}
```

Since *bcast* and *ring* each require $O(p)$ communication steps, whereas *bcube* requires only $O(\log_2 p)$ steps, *bcube* appears to have a significant performance advantage, and for a single broadcast this is certainly true. In Cholesky factorization, however, there is a whole series of broadcasts, one for each successive column that is completed. In the pipelined version of the algorithm these broadcasts can be overlapped, so the length of any one broadcast is less important than the degree of concurrency attainable. We can usually arrange in *bcast* or *ring* that the processor originating a broadcast always send the message first to the node that "needs it most" (i.e., the processor assigned to compute the next column of L), whereas this may not be convenient or even possible with *bcube*. Thus, for example, with *bcast* or *ring*, the node assigned column 1 would receive column 0 before any other processor, whereupon it can complete column 1 and initiate another broadcast immediately. With *bcube*, on the other hand, the node assigned column 1 may participate in the propagation of the initial fan-out broadcast for $\log_2 p$ steps before resuming computational work.

This enhanced pipelining effect is critically dependent on the order in which the broadcast messages are sent. With *bcast*, the messages must be sent to the processors in the same order as the columns are mapped onto the processors, and this may or may not be convenient depending on details of the implementation. With *ring*, there is no flexibility on the order in which the messages circulate through the ring, so the ordering of the columns must be consistent with the order of the processors in the ring. By contrast, the efficiency of *bcube* is less sensitive to the order in which the columns are mapped onto the processors. For example, the columns could be wrapped onto the processors in any order and the performance of *bcube* would be unaffected, whereas the performance of any fixed implementation of *bcast* or *ring* would be strongly affected.

TABLE 2
Execution time (sec) for Cholesky factorization as a function of communication method and consistency of ordering ($p = 32$, $n = 512$).

	synchronous		pipelined	
comm./ord.	consistent	inconsistent	consistent	inconsistent
bcast	137.7	189.3	109.7	156.9
ring	81.2	134.5	74.4	126.6
bcube	90.0	90.7	79.2	80.2

These effects are illustrated in Table 2, in which we give results for all three communication techniques using two different orderings of the columns on the processors: both orderings have an equally uniform load balance, but one ordering is consistent with the ordering used in implementing *bcast* and *ring*, while the other is not. For the consistent ordering, the performance of *ring* and *bcube* is similar and both are distinctly superior to *bcast*. For the inconsistent ordering, the performance of *bcast* and *ring* is seriously degraded whereas the performance of *bcube* remains about the same. We conclude that *bcube* is somewhat less efficient than an optimally ordered implementation of *ring*, but is markedly superior if the ordering of the columns on the processors is inconsistent with the embedded ring. Thus, the choice of algorithm may depend on whether the user is free to choose the ordering or must work with an ordering that is either fixed (e.g., left over from a previous computational phase) or unpredictable (e.g., due to pivoting for numerical stability in the nonsymmetric case). Table 2 also shows the relative performance of the synchronous and pipelined versions of the codes. We see that pipelining yields a significant gain in performance in all cases. Those options that are least efficient to begin with gain the most from pipelining, and the most efficient gain the least, as might be expected.

Finally, Fig. 2 shows the efficiency of the pipelined *ring* and *bcube* algorithms as a function of n. The serial time used here in computing parallel efficiency is based on an observed execution rate of 0.024 Mflops for one processor using a straightforward serial code for Cholesky factorization, coded in C and designed specifically for serial computation on one processor, but otherwise unoptimized (recall the discussion in section 3). An efficiency of 50%, for example,

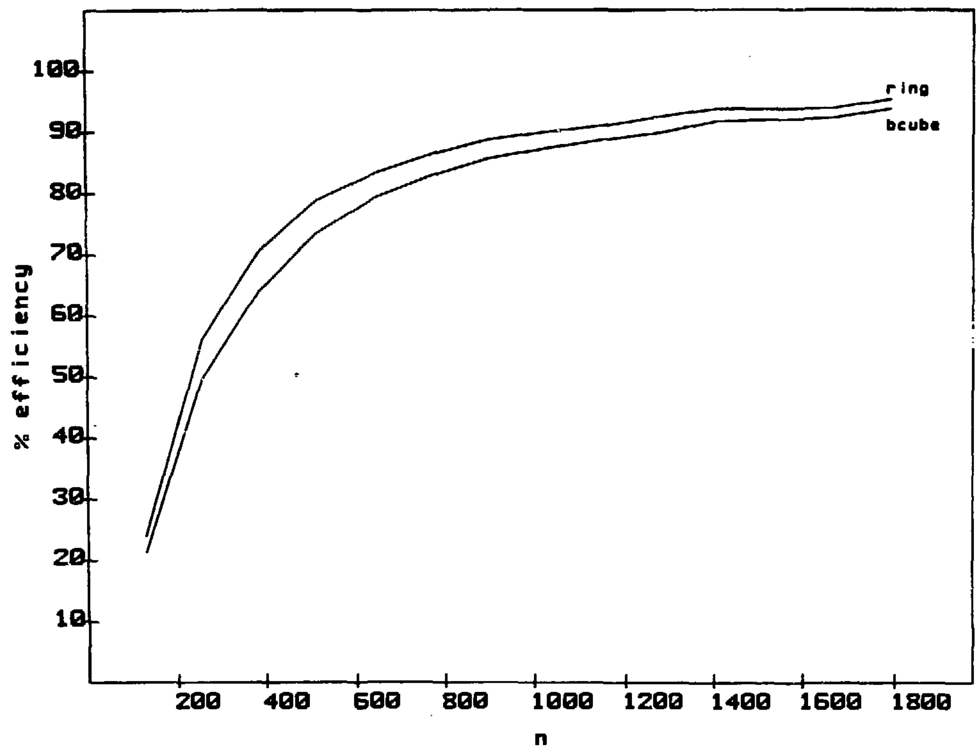

FIG. 2. *Efficiency of Cholesky factorization as a function of n ($p = 32$).*

would mean that the execution rate per processor for the parallel code is 0.012 Mflops. The observed increase in efficiency with problem size is to be expected as communication becomes increasingly dominated by computation.

6. Triangular Solution. The Cholesky factorization is seldom an end in itself. It is most commonly computed in order to solve symmetric positive definite systems of linear equations by forward and back substitutions with the triangular factors L and L^T. Parallel algorithms for triangular solution entail similar precedence constraints and global communication to those necessary for factorization. However, there is an order of magnitude less computation ($O(n^2)$ instead of $O(n^3)$) in forward or back substitution, and so it is correspondingly more difficult to mask the communication cost in order to attain good parallel efficiency. Moreover, the convenience and efficiency of triangular solution are dependent on the method of data access.

Recall that if $A = LL^T$, then we can solve $Ax = b$ by solving the two triangular systems $Ly = b$ and $L^T x = y$ by forward and back substitutions, respectively. If the triangular matrix is stored on the processors by rows, then a parallel algorithm for triangular solution is easily implemented that has similar concurrency and communication pattern to those of the factorization, and it gives relatively good efficiency. For example, in our case L^T is stored on the processors by rows, and so the back substitution can be carried out by the following algorithm, in which each processor contains the components of y and computes the components of the solution x corresponding to the columns it was assigned.

```
for $j = n-1$ to 0 do
begin
    if col $j$ is one of my cols then $x_j = y_j / L_{jj}$
    *communicate* $(x_j)$
    for all of my cols $k < j$ do $y_k = y_k - x_j * L_{jk}$
end
```

Here again for simplicity we have used a synchronous statement of the algorithm, but a pipelined version is also easily implemented. Note that the communication pattern is similar to that of the factorization algorithm, except that only a single number is sent at each stage rather than a whole column. This algorithm works because a given processor has already stored all of the needed elements of L in its local memory during the factorization phase.

If the triangular matrix is stored on the processors by columns, then it is difficult to implement a parallel algorithm without an excessive amount of communication, which seriously impairs parallel efficiency. We can still maintain the same communication pattern by using the following algorithm, in which each processor contains the components of the right hand side b and computes the components of y corresponding to the columns it was assigned.

```
for $j = 0$ to $n-1$ do
begin
    if col $j$ is one of my cols then $y_j = b_j / L_{jj}$
    *communicate* $(y_j$ and col $j)$
    for all of my cols $k > j$ do $b_k = b_k - y_j * L_{kj}$
end
```

Note, however, that the volume of communication is greatly increased because of the necessity to communicate the elements of L that each processor needs but has not previously stored. In fact, this algorithm has the same communication volume as the factorization algorithm, but with an order of magnitude less computation over which to amortize it. As a result, we have found that the trade-off point at which this parallel algorithm outperforms a straightforward sequential forward substitution algorithm is larger than the largest triangular matrix our hypercube can store ($n > 1800$).

Other possibilities exist for parallelizing the forward solution, but these also involve increased communication. For example, one could use a finer-grained algorithm in which individual elements of b are communicated as soon as they are updated. Such an algorithm has excellent concurrency, even with column-oriented storage, but since it requires communication inside the inner loop, it performs extremely poorly on a machine with relatively slow communication. Another possibility would be to use a total exchange algorithm to transpose from column to row storage for the forward substitution, but this would obviously entail a great deal of communication. Moreover, it would either have to be done twice (to return to column storage for the back substitution) or else two copies of the matrix would have to be stored, thereby halving the size of problem that could be solved.

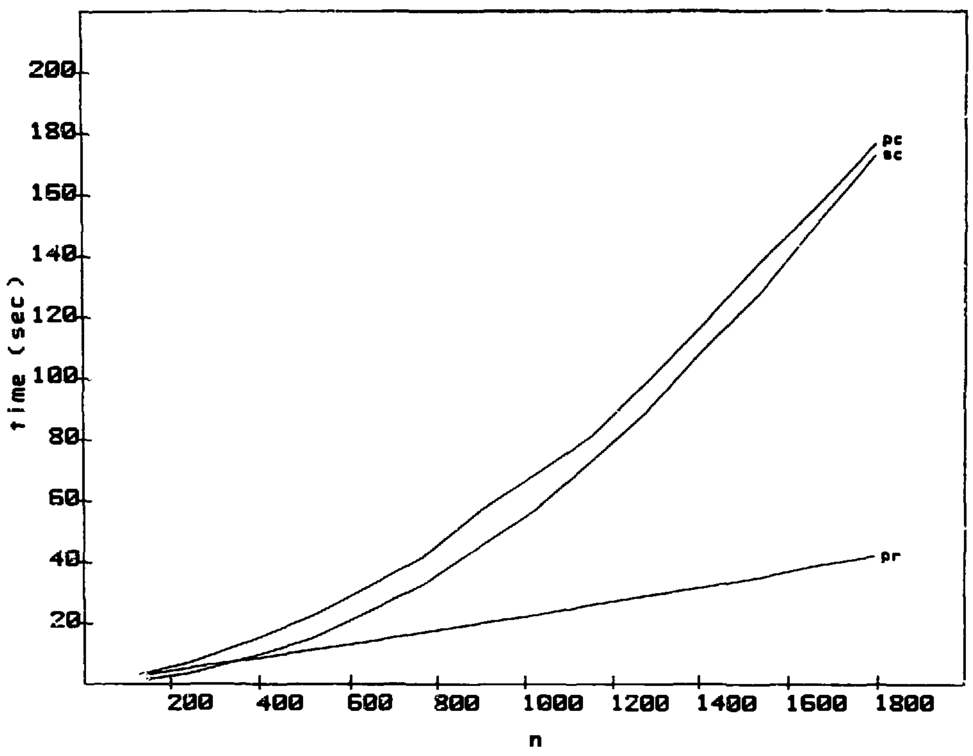

FIG. 3. *Execution time (sec) of triangular solution as a function of n (p = 32). sc = serial by columns, pc = parallel by columns, pr = parallel by rows.*

In order not to incur these high communication costs, we have also implemented a forward substitution algorithm in which the right hand side vector is passed from processor to processor sequentially (see [9] for details). The relative performance of this serial forward solution, the parallel forward solution, and the parallel backward solution are shown in Fig. 3. Thus, we see that due to its very low communication requirements ($O(n)$), the serial algorithm performs quite acceptably for modest sized problems. Moreover, if the right hand side is known in advance of the factorization, then the forward substitution can easily be pipelined with the factorization, using the processors when they would otherwise become idle for the remainder of the factorization after completing their last columns. Nevertheless, in order to be able to handle subsequent triangular solutions efficiently for large problems, parallel algorithms for matrices stored by columns bear further development.

7. LU Factorization.

We now turn to the *LU* factorization of nonsymmetric matrices by Gaussian elimination, in which the principal new difficulty is the necessity of partial pivoting for numerical stability. We again have a choice of storing the matrix on the processors by columns or by rows. Storage by columns would greatly simplify the pivoting procedure, since the search for a pivot element in a given column would be confined to a single processor. On the other hand, this would leave us with both L and U stored by columns, and therefore both the forward and back substitutions would be relatively inefficient for reasons discussed in section 6. Storage by rows would be ideal for the triangular solutions, but significantly complicates the search for pivots, since the necessary information is then spread over all of the processors. Nevertheless, if the *LU* factors are to be used for many right hand sides, the potential payoff resulting from row-oriented storage led us to develop a row-oriented implementation of the factorization with partial pivoting [8]. (For an alternative approach using column storage for the factorization, and transposition to row storage before the triangular solution, see [3].)

The global communication requirements for *LU* factorization of a nonsymmetric matrix are similar to those for Cholesky factorization of a symmetric matrix, except that additional communication is required for determining and distributing pivoting information. In principle, any of the broadcasting methods discussed in section 5 could be used. Recall from Table 2, however, that the performance of *ring* communication depends strongly on the consistency of the ordering, which is unlikely to be attainable when pivoting is required for numerical stability. We have therefore used only *bcube* style communication in implementing *LU* factorization with pivoting.

The overall efficiency of a parallel algorithm for *LU* factorization using row-oriented storage depends on the efficiency of the pivot search. We developed two basic strategies for the pivot search, with several variants of the second. Our first strategy uses the host processor to select the pivot in a manner that almost completely masks the communication cost by overlapping the selection process with computation, provided communication between host and nodes is sufficiently fast. Upon receiving the pivot row, each processor computes its portion of the first column of the reduced matrix, sends the largest element produced to the host, and then resumes computing the remaining portion of the reduced matrix. Meanwhile, the host receives the local maximum from each processor and can therefore determine the global maximum and notify the processor holding the corresponding row to broadcast it to the other processors as the next pivot row. Since the host performs the pivot search while the other processors are completing the computation of the remainder of the reduced matrix, the cost of pivoting is potentially negligible, as was verified by simulation. Unfortunately, the extremely slow communication rate between the host and nodes on the Intel iPSC causes this approach to perform poorly in practice on that machine.

Our second approach relies solely on the node processors to make the pivot selection. At each major step of the elimination, all of the processors must communicate among themselves sufficient information to determine the next pivot row. One way to do this is to use a communication pattern that is the reverse of the spanning tree broadcast discussed earlier. Each leaf node in the tree determines its local maximum in the pivot column (i.e., the largest element in magnitude among those rows the processor holds), which is then sent to its parent node. Each parent node determines its own local maximum, compares it with the local maxima received from its children, and sends the overall local maximum to its parent. After $\log_2 p$ steps, the global maximum has been determined by the root node, which must then broadcast this information back out to the other processors. Finally, the processor holding the pivot row must broadcast the pivot

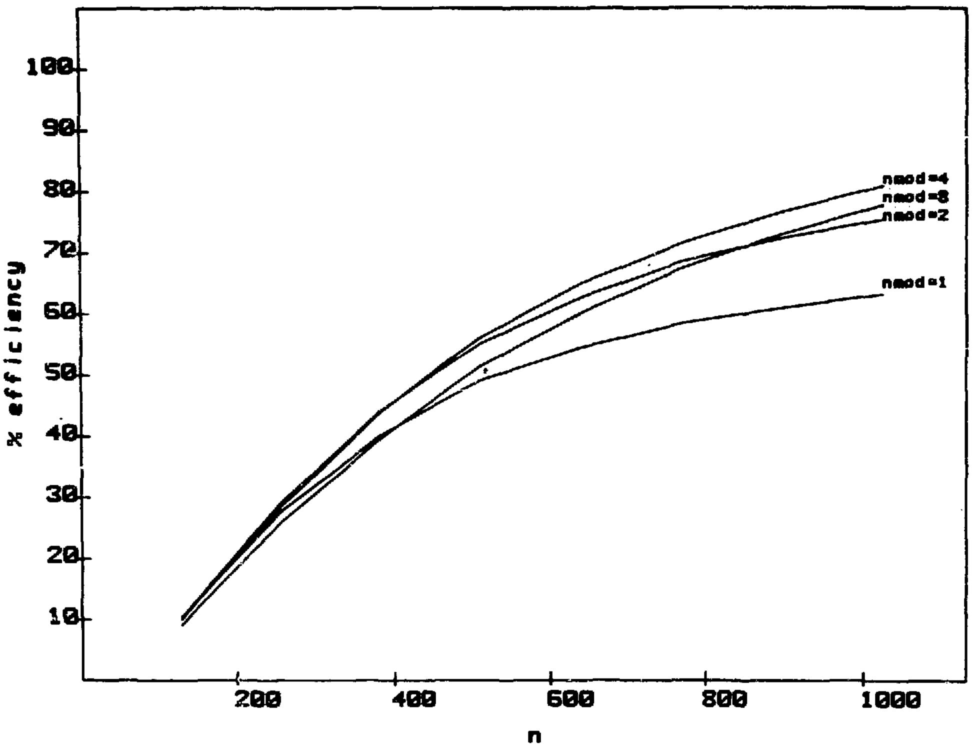

FIG. 4. *Efficiency of LU factorization as a function of n (p = 32).*
nmod = number of rows applied at a time.

row to all of the other processors. Thus, a total of three logarithmic communication stages are needed. This can be reduced to two if each processor sends its entire candidate pivot row in the initial fan-in cascade (rather than just the local maximum element in the pivot column), since then the subsequent broadcast from the root node can send the pivot row to each processor, thereby avoiding the final broadcast. The price, however, is a significant increase in the total communication volume, since many elements will be sent that are not used. In practice, we observed little difference between these two variants, but this may be an artifact of the large packet size on the iPSC, which tends not to penalize large messages.

The pivot selection process prior to each major step of elimination tends to inhibit the use of pipelining techniques of the type used in Cholesky factorization, in which successive eliminations are overlapped. It is possible to overlap eliminations with pivoting, but quite inconvenient, and so we have developed only a synchronous version of our *LU* algorithm. Another design choice is whether to interchange rows according to the order of pivots chosen, as is often done in serial algorithms. In a distributed-memory system, actual physical interchange of rows among processors would incur a significant amount of communication overhead. On the other hand, if rows are not interchanged, then pivoting causes the ordering of rows on the processors to become essentially random, thereby risking a poor load balance as we saw in Table 1. In our experience, the degradation in performance caused by random pivoting compared to wrap mapping is in the range of 5% to 15%, which we deemed to be not substantial enough to warrant the additional complexity and communication required to interchange rows. Chu and George have investigated an explicit interchange strategy in distributed memory [2].

Another way to improve computational efficiency is to save several pivot rows at a time before applying all of them to reduce the remaining unreduced submatrix. In effect, this unrolls the middle loop of the elimination, thereby reducing array indexing overhead and allowing the possibility of more efficient use of hardware registers (provided the compiler is sufficiently intelligent to take such advantage). The price paid is a slight increase in code complexity and the temporary storage needed to accumulate the rows. We have observed significant performance gains from this strategy, however, so we feel that it is worthwhile. Fig. 4 shows comparative performance of applying 1, 2, 4, or 8 rows at a time to the unreduced matrix. As the number of rows applied (amods in the figure) grows, a larger matrix is required in order to overcome startup overhead and realize increased efficiency.

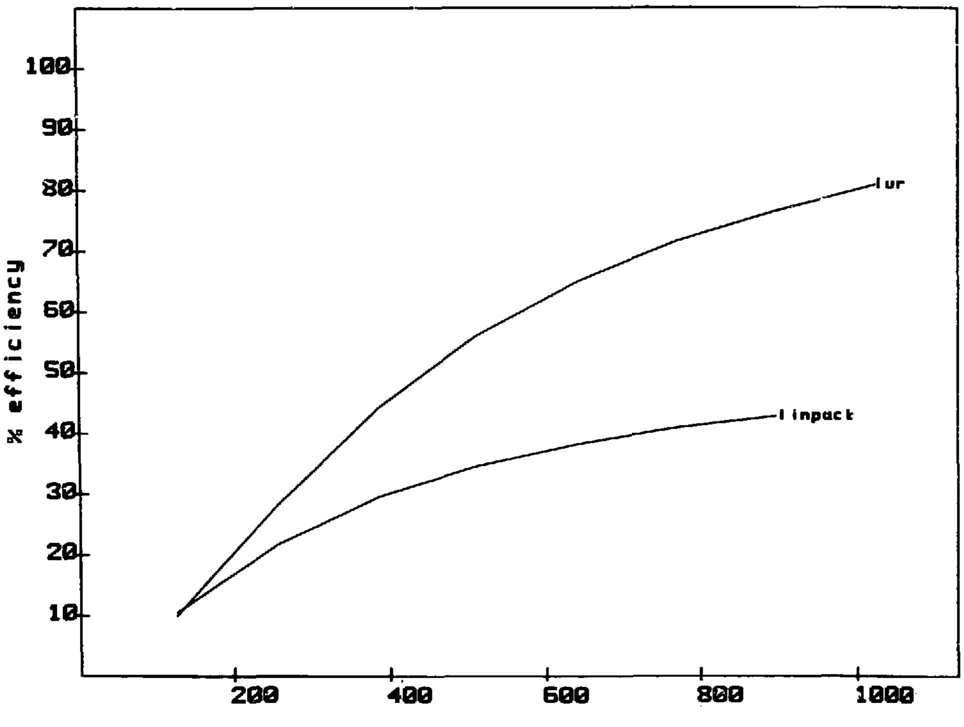

FIG. 5. *Efficiency of LU factorization as a function of n, comparing parallel row LU with parallel LINPACK (p = 32).*

In Fig. 5 the performance of our row-oriented implementation of *LU* factorization with partial pivoting is compared to that of a column-oriented implementation based on SGEFA from LINPACK [4]. We should point out that the latter implementation was not primarily motivated by seeking optimal performance, but rather by the desire to port the serial LINPACK algorithm to a parallel environment with minimal changes. Nevertheless, it is interesting to observe that our row-oriented *LU* factorization algorithm outperforms the column-oriented factorization algorithm, despite the higher cost of pivot selection in the row algorithm.

The serial benchmark rate used in computing parallel efficiency in Figs. 4 and 5 is 0.04 Mflops on one processor. This rate is higher than that used for Cholesky factorization because both our parallel row-oriented *LU* code and the LINPACK code use source-level code optimization techniques (notably loop unrolling) that effectively increase the computational speed, and the same applies to the serial benchmark code. Ironically, this improvement in computational performance has the effect of lowering the estimated efficiency, but this is consistent with the corresponding change in the effective ratio of computation speed to communication speed (i.e., as computation becomes faster relative to communication, communication overhead becomes a relatively larger part of the total time, thereby lowering parallel efficiency).

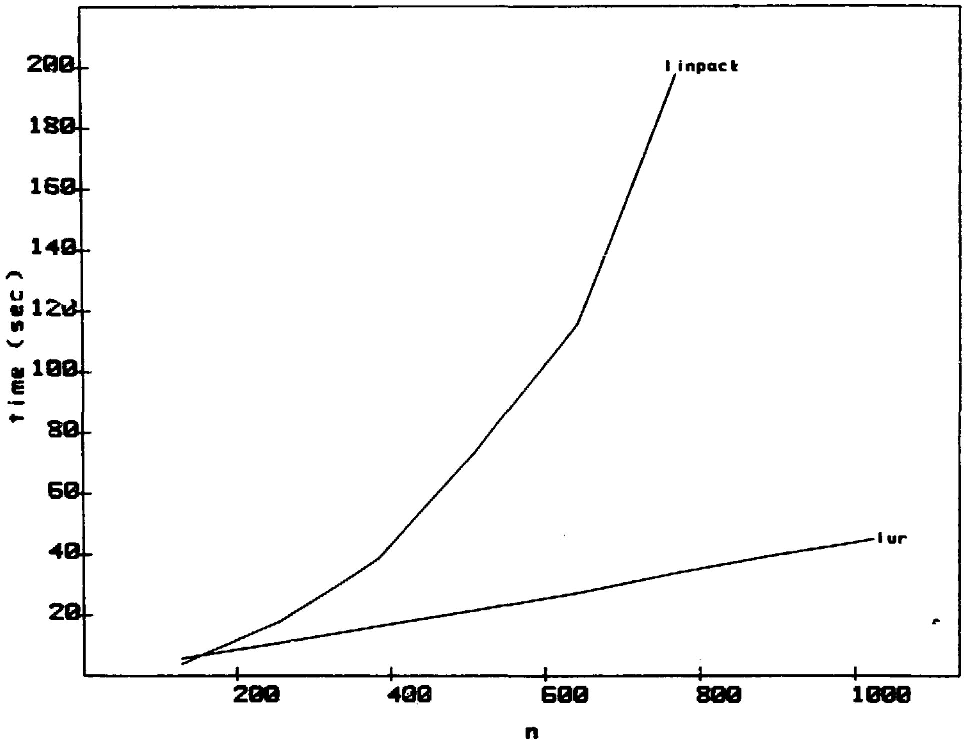

FIG. 6. *Execution time (sec) of triangular solutions (total for forward and back substitutions) as a function of n (p = 32).*

The row-oriented factorization also makes possible very efficient row-oriented triangular solutions, in contrast to the serial triangular solutions used in the LINPACK implementation. We observe in Fig. 6 that the LINPACK triangular solutions are computation bound ($O(n^2)$), whereas our row-oriented triangular solutions are communication bound, so that the execution time of the latter increases only linearly with n.

8. Conclusion. In this paper we have empirically compared numerous strategies for solving symmetric and nonsymmetric linear systems on a hypercube. Although our results are somewhat dependent on the particular characteristics of the Intel iPSC machine used in our experiments, some general observations are likely to be true of a wide variety of possible hypercube designs:

— Both column-oriented and row-oriented factorization algorithms can be highly efficient (in the 80–90% range of the theoretical maximum) if the matrix is large enough, even on a machine with relatively slow communication.

— For mapping the matrix onto the processors, a uniform scattering of the columns or rows, such as that provided by wrap mapping, is preferable to block mapping or any randomly chosen mapping.

— For dense matrix factorization, it pays to take full advantage of the structure of the hypercube network for global communication. In particular, logarithmic, spanning-tree broadcasting is an effective method of implementing global communication that is flexible enough to handle unpredictable or uncontrollable orderings efficiently.

— Using only an embedded ring for communication can also be very effective in a highly regular and homogeneous computation such as matrix factorization, but a ring is less flexible and more sensitive to the mapping of the matrix onto the processors.

— Efficient parallel triangular solutions are much easier to attain with row-oriented storage of the matrix on the processors than with column-oriented storage.

— The communication cost of searching for pivots in LU factorization can be overcome to produce an efficient row-oriented parallel algorithm, which also facilitates subsequent triangular solutions.

Some of the issues we have raised merit further study, including more efficient triangular solutions using column-oriented storage and two-dimensional partitionings of the matrix ("patches" instead of "strips"). We note that many of the issues and techniques we have studied are also pertinent to the factorization of sparse matrices to solve sparse linear systems [12]. We hope to run the same experiments on several other hypercubes to see how relative performance varies when the design parameters change. We have tried to answer a number of specific technical questions concerning matrix factorization on a message-passing multiprocessor. More generally, our computational experience with matrix factorization shows that serial precedence constraints and global communication are not necessarily insurmountable obstacles to high parallel efficiency.

REFERENCES

[1] P. R. Cappello, *Solving dense linear systems on a hypercube automaton*, Tech. Rept. TRCS85-11, Department of Computer Science, University of California, Santa Barbara, California, 1985.

[2] E. Chu and A. George, *Gaussian elimination with partial pivoting and load balancing on a multiprocessor*, Tech. Rept., Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1986.

[3] G. J. Davis, *Column LU factorization with pivoting on a hypercube multiprocessor*, Tech. Rept. ORNL-6219, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1985.

[4] J. J. Dongarra, J. R. Bunch, C. B. Moler and G. W. Stewart, *LINPACK User's Guide*, SIAM, Philadelphia, 1979.

[5] J. J. Dongarra, F. G. Gustavson and A. Karp, *Implementing linear algebra algorithms for dense matrices on a vector pipeline machine*, SIAM Rev., 26 (1984), pp. 91-112.

[6] G. C. Fox, *Matrix operations on the homogeneous machine*, Tech. Rept. HM-5, California Institute of Technology, Pasadena, California, 1982.

[7] G. C. Fox, *Square matrix decompositions - symmetric, local, scattered*, Tech. Rept. HM-97, California Institute of Technology, Pasadena, California, 1984.

[8] G. A. Geist, *Efficient parallel LU factorization with pivoting on a hypercube multiprocessor*, Tech. Rept. ORNL-6211, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1985.

[9] G. A. Geist and M. T. Heath, *Parallel Cholesky factorization on a hypercube multiprocessor*, Tech. Rept. ORNL-6190, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1985.

[10] W. M. Gentleman, *Some complexity results for matrix computations on parallel processors*, J. ACM, 25 (1978), pp. 112-115.

[11] A. George, M. T. Heath and J. Liu, *Parallel Cholesky factorization on a shared-memory multiprocessor*, Linear Algebra Appl., 77 (1986), pp. 165-187.

[12] A. George, M. T. Heath, J. Liu and E. Ng, *Sparse Cholesky factorization on a local-memory multiprocessor*, Tech. Rept. ORNL/TM-9962, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1986.

[13] M. T. Heath, *Parallel Cholesky factorization in message-passing multiprocessor environments*, Tech. Rept. ORNL-6150, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1985.

[14] I. C. F. Ipsen, Y. Saad and M. H. Schultz, *Complexity of dense linear system solution on a multiprocessor ring*, Linear Algebra Appl., 77 (1986), pp. 205-239.

[15] S. L. Johnsson, *Communication efficient basic linear algebra computations on hypercube architectures*, Tech Rept. YALEU/DCS/RR-361, Department of Computer Science, Yale University, New Haven, Connecticut, 1985.

[16] D. P. O'Leary and G. W. Stewart, *Data-flow algorithms for parallel matrix computations*, Comm. ACM, 28 (1985), pp. 840-853.

[17] E. M. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[18] Y. Saad, *Communication complexity of the Gaussian elimination algorithm on multiprocessors*, Linear Algebra Appl., 77 (1986), pp. 315-340.

[19] C. L. Seitz, *The cosmic cube*, Comm. ACM, 28 (1985), pp. 22-33.