SIAM J. SCI. STAT. COMPUT.
Vol. 9, No. 4, July 1988

© 1988 Society for Industrial and Applied Mathematics

LU FACTORIZATION ALGORITHMS ON DISTRIBUTED-MEMORY MULTIPROCESSOR ARCHITECTURES*

GEORGE A. GEIST† AND CHARLES H. ROMINE†

Abstract. In this paper, we consider the effect that the data-storage scheme and pivoting scheme have on the efficiency of *LU* factorization on a distributed-memory multiprocessor. Our presentation will focus on the hypercube architecture, but most of our results are applicable to distributed-memory architectures in general. We restrict our attention to two commonly used storage schemes (storage by rows and by columns) and investigate partial pivoting both by rows and by columns, yielding four factorization algorithms. Our goal is to determine which of these four algorithms admits the most efficient parallel implementation. We analyze factors such as load distribution, pivoting cost, and potential for pipelining. We conclude that, in the absence of loop-unrolling, *LU* factorization with partial pivoting is most efficient when pipelining is used to mask the cost of pivoting. The two schemes that can be pipelined are pivoting by interchanging rows when the coefficient matrix is distributed to the processors by columns, and pivoting by interchanging columns when the matrix is distributed to the processors by rows.

Key words. parallel algorithms, distributed-memory multiprocessors, *LU* factorization, Gaussian elimination, hypercube

AMS(MOS) subject classifications. 65F, 65W

1. Introduction.

This paper describes four approaches for implementing *LU* factorization on a distributed-memory multiprocessor, specifically a hypercube. Our goal is to determine whether the choice of storage scheme for the coefficient matrix and pivoting strategy appreciably affects the efficiency of parallel factorization and, if so, which of the four algorithms is to be preferred. The empirical results presented in the sequel were obtained by implementing the factorization algorithms on an Intel iPSC hypercube.

A number of papers have appeared in recent years describing various approaches to parallelizing *LU* factorization, including Davis [4], Chamberlain [2], and Geist [7]. The present work is motivated primarily by Geist and Heath [8] and Chu and George [3]. In most of these earlier papers, row storage for the coefficient matrix was chosen principally because no efficient parallel algorithms were then known to exist for the subsequent triangular solutions if the coefficient matrix was stored by columns. Recently, Romine and Ortega [16], Romine [15], Li and Coleman [11] [12], and Heath and Romine [10] have demonstrated such algorithms, removing triangular solutions as a reason for preferring row storage. In addition, if the coefficient matrix is stored by rows then pivoting by interchanging rows involves extra communication, since the elements which must be searched are scattered among the processors. With column storage, no additional communication is required. Hence, column storage for the coefficient matrix warrants further investigation.

One alternative method that has been suggested for the solution of linear systems on distributed-memory multiprocessors is *QR* factorization (see Ortega and Voigt [14]). *QR* factorization is inherently stable and thus avoids the complication of pivoting. Since the operation count for *QR* factorization is twice that of *LU* decompo-

* Received by the editors February 24, 1987; accepted for publication September 15, 1987.

† Mathematical Sciences Section, Oak Ridge National Laboratory, P.O. Box Y, Oak Ridge, Tennessee 37831. The research of this author was 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.

sition, QR factorization will only be competitive if the efficiency of LU factorization with pivoting is less than half the efficiency of QR factorization. We show that the parallel LU factorization algorithms presented in this paper have efficiencies of over 85 percent. Given this result, parallel QR factorization is not considered a competitive alternative to LU factorization.

2. LU factorization with row storage and row pivoting. The first algorithm we discuss is LU factorization with row interchanges on a matrix which has been assigned to the processors by rows, which we will refer to as RSRP. The algorithm is given in Fig. 1. At each major stage of the algorithm, the pivot row must first be

```
for k = 0 to n − 1
    determine pivot row
    update permutation vector
    if (I own pivot row)
        broadcast pivot row
    else
        receive pivot row
    for (all rows i > k that I own)
        l_ik = a_ik / a_kk
        for j = k + 1 to n − 1
            a_ij = a_ij − l_ik a_kj
```

FIG. 1. *The* RSRP *algorithm.*

determined. This requires communication among all the processors, since the pivot column is scattered. An effective strategy for performing global communication on a hypercube is through the use of a minimal spanning tree embedded in the hypercube network (for an illustration, see Geist and Heath [8]). This allows information either to be disseminated (fanned-out) from one processor to all, or collected (fanned-in) from all processors into one, in $\log_2 p$ steps. In the current context, each processor searches its portion of the pivot column for the element of maximum modulus. The leaf nodes of the spanning tree send these local maxima to their parents. The parents compare these received values to their own local maxima, forwarding the new maxima up the tree. When the fan-in is complete, the pivot row will have been determined by the root processor in the spanning tree, which must then send this information back down the tree. Finally, the processor that contains the pivot row must fan it out to the other processors. Hence, three logarithmic communication stages are performed before updating of the submatrix can begin. Two stages are sufficient if the entire row corresponding to the local maximum is sent in the first fan-in; however, the resulting large increase in communication volume was observed to cause an increase in execution time on the iPSC for $n > 500$.

Note that pivoting is carried out implicitly in the above algorithm; that is, no explicit exchange of matrix elements takes place. This has the benefit of requiring no added communication, but at the risk of incurring a poor distribution of the load. Even under the assumption that the rows are distributed evenly to the processors, the execution time for factorization can vary widely depending upon the order of the distribution. For example, Geist and Heath [8] observe that blocking (assigning n/p consecutive rows to each processor) causes a 50 percent degradation in factorization time relative to wrapping (assigning row i to processor $i \pmod p$). They also report

that a random distribution of the rows to the processors (the effect of implicit pivoting) usually causes a 5 to 15 percent degradation in execution time relative to wrapping.

To illustrate the overhead of pivoting, we have used the above algorithm to factor both a diagonally dominant matrix which is wrap-mapped and a random matrix, both of order 1024. Since no pivoting is actually performed on the diagonally dominant matrix, wrapping is preserved by the algorithm. As a further illustration of pivoting overhead, we also present the time for factorization with the pivot search removed. The results are summarized in the first column of Table 1. The total overhead for pivoting in the algorithm (including the penalty for load imbalance) is 129.4 seconds, of which only 24.6 seconds is due to the pivot search. The remaining 104.8 seconds, 11 percent of the total factorization time, is due solely to the poor load balance produced by the order of selection of the pivot rows.

TABLE 1
Results for the RSRP *algorithm.*

Matrix of order 1024 on 32 processors				
	Implicit pivoting	Explicit pivoting (Chu and George)	Dynamic pivoting (Strategy 1)	Dynamic pivoting (Strategy 2)
Diagonally dominant (no pivot search)	816.4	816.4	816.4	816.4
Diagonally dominant (incl. pivot search)	841.0	841.0	841.0	841.0
Random matrix	945.8	921.3	876.5	852.2
Number of exchanges	0	993	471	454

A natural attempt at decreasing this overhead would be to force a wrap mapping by exchanging rows explicitly when necessary. That is, if processor $k \pmod p$ does not contain the kth pivot row, then it exchanges rows with the processor that does. This strategy was first investigated for the hypercube by Chu and George [3], in which it was demonstrated that the extra communication cost required by explicit exchange is more than offset by the gain due to improved load balance.

In order to ensure fairness in the comparisons, we have implemented the Chu and George strategy for pivoting in the algorithm described above. The same random matrix was factored with this new algorithm, and the results given in the second column of Table 1. Even though 993 row exchanges were required (nearly the maximum possible), the explicit exchange strategy performed better than implicit pivoting. However, there is still an 80.3-second penalty for these exchanges (almost 10 percent of the total execution time), compared to only 24.6 seconds for the pivot search.

These results agree with the conclusion given in Chu and George [3], that balancing the load is desirable even at the cost of increased communication. However, load balancing can be achieved with fewer exchanges than is required by the Chu and George pivoting strategy. The large number of exchanges is caused by the requirement that the final distribution of the rows be a wrap mapping. Wrap mapping balances the load effectively, but other mappings are equally effective at load balancing. Hence, we should be able to design a less restrictive explicit pivoting strategy which will reduce the number of exchanges from that required by the Chu and George strategy, while at the same time balancing the load. One possibility is to require that any p consecutive rows be distributed evenly to the p processors. However, this is only a permuted form

of wrapping, and will also produce a large number of exchanges.

A less restrictive rule is to require that rows kp through $(k+1)p-1$ $(0 \leq k < n/p)$ lie in distinct processors for each k, with the order in which they are assigned unconstrained. That is, a processor that already contains one of these pivot rows cannot contain another, and must exchange rows with a processor that does not already contain one. This scheme produces any one of a family of mappings that have the load-balancing properties of wrapping in that the rows assigned to a processor are more or less uniformly distributed in the matrix. This scheme allows considerable leeway in the choice of mapping and, hence, should reduce the number of exchanges required during pivoting. Because the final mapping depends upon the elements of the matrix, we call this pivoting strategy *dynamic pivoting*. The RSRP algorithm with dynamic pivoting included is shown in Fig. 2. (The function *dmap* selects the processor that will contain pivot row k. This value is assigned to the permutation vector $maps_k$).

```
for k = 0 to n − 1
    next = processor containing kth pivot row
    maps_k = dmap(next)
    if ( maps_k ≠ next )
        processors maps_k and next exchange rows
    if ( me = maps_k )
        broadcast pivot row
    else
        receive pivot row
    for (all rows i > k that I own)
        l_ik = a_ik/a_kk
        for j = k + 1 to n − 1
            a_ij = a_ij − l_ik a_kj
```

FIG. 2. *The* RSRP *algorithm with dynamic pivoting.*

The implementation of dynamic pivoting raises a further question. If a processor finds itself with two pivot rows when only one is allowed, with which processor should it exchange rows? Any processor which does not yet contain a pivot row in the current set of p rows is a valid choice. The simplest procedure (Strategy 1) is to scan the list of processors from 0 to $p-1$ until a valid processor is found. This search procedure was implemented on the iPSC, and was found to improve dramatically the performance of LU factorization. The results of dynamic pivoting with Strategy 1 are given in the third column of Table 1. Notice that the number of exchanges is less than half that required by the Chu and George strategy. This reduction in the number of exchanges is directly responsible for the improvement in performance. The overhead for exchanging rows is now only 35.5 seconds, nearly the same as the overhead for the pivot search.

Strategy 1, while conceptually simple, can require communication between distant processors in the hypercube topology. Hence, in selecting the processor with which to exchange rows, a better strategy might be to choose the nearest valid neighbor. A breadth-first search of the minimal spanning tree rooted at a particular node yields a list of processors in increasing order of distance from the node. Such a search strategy (Strategy 2) should decrease the average distance between exchanging processors

while maintaining a low number of exchanges, and hence improve the performance of dynamic pivoting. Strategy 2 was implemented on the iPSC, and the results are shown in column 4 of Table 1. The overhead for performing the exchanges is now only 11.2 seconds, less than half the cost of the pivot search and only about 1 percent of the total execution time for the factorization. The slight decrease observed in the number of exchanges is not significant. We expect that in general, a roughly equal number of exchanges will be required using Strategy 1 or 2.

It is important to make certain that deviating from the wrap mapping does not cause undue overhead during the triangular-solution stages, since this may negate any savings obtained during the factorization. The most efficient parallel algorithms known for the solution of a triangular system on a hypercube rely heavily on the wrap mapping for their performance (see Heath and Romine [10], and Li and Coleman [12]). However, the performance of the *cube fan-out* algorithm is largely unaffected by the choice of mapping. Experiments reveal that for a matrix of order 1024 on a 32-node iPSC, the cube fan-out algorithm is only about 6 seconds slower than the most efficient algorithms. Hence, unless several systems with the same coefficient matrix are to be solved, the 69 seconds saved by using dynamic pivoting more than offsets the 12-second increase in the time required to perform the triangular solutions.

A significant amount of extra communication is required for explicit row pivoting when the coefficient matrix is stored by rows. Chu and George [3] were able to show that the improvement in the final distribution of the load makes the extra communication worthwhile. Furthermore, the improvements to the Chu and George strategy presented in this section show that, even for large n, row pivoting with row storage increases the execution time only slightly over the case where no pivoting is done at all. In the sequel, we shall refer only to the RSRP algorithm in which dynamic pivoting using strategy 2 is used, since this is the best form of this algorithm.

3. LU factorization with column storage and row pivoting. The second algorithm we will describe is *LU* factorization with row pivoting when the coefficient matrix is distributed among the processors by columns, which we will refer to as CSRP. The algorithm is given in Fig. 3. This algorithm is quite similar to the RSRP algorithm, except that the updating of the submatrix is done by columns rather than by rows. In the taxonomy of Dongarra *et al.* [5] this is the *kji*-form, as opposed to the *kij*-form of *LU* factorization used in the RSRP algorithm (see Ortega and Romine [13]). Since the coefficient matrix is stored by columns, the computation of the column of multipliers at each stage is done serially by the processor containing the pivot column. This will reduce the efficiency of the factorization unless this serial phase can be masked.

Pivoting by rows with storage by columns has several implications. First, the way in which the columns are mapped to the processors remains unchanged by pivoting. This is in contrast to the previous case, where obtaining a good mapping after pivoting required that the rows be reshuffled. Hence, we can ensure a good load balance by initially wrapping the columns onto the processors. Second, the pivot column lies entirely within a single processor, implying that the search for the element with maximum modulus must be carried out serially. However, while this increases the number of serial phases in the algorithm, it eliminates the communication required by the previous algorithm during the pivot search. It is unclear *a priori* how this trade-off affects the relative performance of the two algorithms. It has been shown that the communication required for row pivoting when the coefficient matrix is stored by rows does not unduly degrade the performance of *LU* factorization; however, it is

```
for k = 0 to n − 1
    if (I own column k)
        determine pivot row
        interchange
        for i = k + 1 to n − 1
            l_ik = a_ik/a_kk
        broadcast l and pivot index
    else
        receive l and pivot index
        interchange
        for (all columns j > k that I own)
            for i = k + 1 to n − 1
                a_ij = a_ij − l_ik a_kj
```

FIG. 3. *The* CSRP *algorithm.*

conceivable that eliminating the communication entirely from the pivoting stage will improve efficiency.

The algorithm described above was implemented on the Intel iPSC, and the results are given in the first column of Table 2. (The results for factoring the diagonally dominant matrix including the pivot search are identical to those of the random matrix.) While there is a slight increase in execution time over the RSRP algorithm for the nonpivoting case (due to the serial computation of the multipliers), there is a drastic increase in the running time when pivoting is included. Clearly, the cost of performing a serial search far exceeds the communication cost of the parallel search in the RSRP algorithm.

The explanation for the large difference in the cost of serial versus parallel pivoting is simple. The cost of serially searching the pivot column is (on average) approximately $(n/2)s$, where s is the cost of comparing two floating-point numbers. The average cost of the parallel search is approximately $(n/2p)s + c \log p$, where c is the cost of communicating a floating-point value between neighboring processors. Even with c large, as n grows the cost of the serial search is about p times as much as the cost for the parallel search, since the communication term becomes negligible.

This disparity in the cost of pivoting between the RSRP and CSRP algorithms means that unless there is some way to reduce the cost of serial pivoting (and serial computation of the multipliers), the CSRP algorithm will not be competitive. Fortunately, most of the serial overhead in the CSRP algorithm can be masked through the use of pipelining. We use the term pipelining to mean a reduction in latency obtained when a processor, rather than continuing its current computation, sends already computed values to other processors. The degree of pipelining is defined by the amount of such information sent. For example, a high degree of pipelining is achieved if the processor containing the next pivot column, before updating its portion of the submatrix, first computes and sends each multiplier one at a time. This minimizes the latency that prevents the other processors from beginning their computations, but it drastically increases the communication cost. A moderate degree of pipelining occurs when the processor containing the next pivot column, before updating its portion of the submatrix, first computes and then sends the whole column of multipliers. This is the scheme used to produce the results given in column 2 of Table 2. It should

be noted that pipelining is infeasible in the RSRP algorithm since the pivoting stage requires the cooperation of all the processors.

TABLE 2
Results for the CSRP *algorithm.*

Matrix of order 1024 on 32 processors		
	Basic algorithm	Pipelined algorithm
Diagonally dominant (no pivot search)	843.3	802.7
Random matrix	929.7	804.2

As the results in Table 2 indicate, the large latency time induced by the serial pivot search and serial computation of the multipliers in the CSRP algorithm has been almost entirely eliminated by pipelining. The cost of pivoting is now a negligible percentage of the total factorization time. If we now compare the factorization time of the CSRP algorithm (including pipelining) with that of the RSRP algorithm, we see that the CSRP algorithm is 48 seconds faster, approximately 6 percent of the total execution time.

4. LU factorization with column pivoting. *LU* factorization using column pivoting is advocated in Barrodale and Stewart [1] in the context of interpolation problems, and further described in Chamberlain [2]. Barrodale and Stewart's version of the algorithm involves an extra search phase to take advantage of solving systems in which several components of the solution vector are known to be quite small. Since we are concerned with efficient implementation of *LU* factorization for general systems, we will eliminate this phase of the algorithm.

The algorithm, which we refer to as RSCP, consists of searching the current pivot row for the element with maximum modulus, and then exchanging columns to bring this element to the diagonal. The RSCP algorithm can quickly be seen as nothing more than the dual of the CSRP algorithm and hence the same techniques would apply. When implemented on the iPSC, it yielded the same results. Hence, there is no reason to pursue this algorithm further.

As might be expected, *LU* factorization with column storage and column pivoting, which we refer to as CSCP, is the dual of the RSRP algorithm, and would yield results identical to those listed in §2. However, one difference in the resulting factors of the two algorithms should be noted. *LU* factorization using either the RSRP or the CSRP algorithm yields a matrix L all of whose entries are less than or equal to 1. The RSCP and CSCP algorithms produce the reverse situation, in which the elements of U are less than or equal to 1. Since the back substitution phase of Gaussian elimination solves the triangular system $Ly = b$ and then $Ux = y$, this difference can have an effect upon the error obtained in the solution. If L contains large elements (as in RSCP and CSCP), then rounding error can occur in the solution of $Ly = b$ which is then propagated through the solution of $Ux = y$. In practice, we have noticed that the error produced by RSCP can be significantly larger than that produced by RSRP.

5. Unrolling the middle loop of LU factorization. The concept of expanding the computation in a looping procedure by writing it out explicitly is an established technique for reducing the amount of integer arithmetic in a numerical algorithm. Since the ratio of the costs of floating-point and integer arithmetic has dropped due

to the advent of floating-point accelerators, a reduction in integer overhead can dramatically improve the performance of an algorithm. Commonly used on both serial and vector computers, the effect that such *unrolling* of a computational loop has on a parallel numerical algorithm has only been recently explored (see Dongarra and Hewitt [6]). Geist and Heath [8] recognized that this technique could be applied to LU factorization on a hypercube without seriously impairing the amount of parallelism obtained.

In the context of LU factorization, unrolling the middle loop corresponds to applying multiple pivot rows at the same time to update the submatrix. For example, instead of applying a single pivot row p to update the rows of the submatrix via

$$\begin{aligned} &\textbf{for } j = i+1 \textbf{ to } n-1 \\ &\quad a_{ij} = a_{ij} - m_{ik}p_j \end{aligned}$$

we can instead apply two pivot rows p and q via

$$\begin{aligned} &\textbf{for } j = i+1 \textbf{ to } n-1 \\ &\quad a_{ij} = a_{ij} - m_{i-1,k}p_j - m_{ik}q_j. \end{aligned}$$

As described in Geist and Heath [8], this will reduce the high-order term in the expression for integer arithmetic cost from $2n^3/3$ to $n^3/2$. This cost can be reduced further by saving more than two pivot rows to be applied simultaneously. In general, the coefficient of the n^3 term in the integer operation count for LU factorization is $(r+1)/3r$ if r pivot rows are applied at a time.

The function $(r+1)/3r$ rapidly approaches a horizontal asymptote, showing that little improvement in the execution time of LU factorization can be expected for $r > 4$. Furthermore, as r increases there is eventually a point at which the incremental reduction in computation is less than the overhead required to save the extra pivot rows. Our experience on the iPSC has been that applying 4 pivot rows at a time minimizes the execution time of LU factorization for a wide range of problem sizes and hypercube sizes. Since the time for a floating-point operation on the iPSC is only about two and a half times the cost of an integer operation, a large savings in execution time can be expected.

A loop-unrolling technique in which various values of r can be chosen has been implemented in each of the variations of the RSRP algorithm, and the results are summarized in Table 3. Because the RSRP algorithm is synchronous rather than pipelined, the implementation of this technique is straightforward.

Note that in each case, the execution time of the algorithm has dropped by almost 25 percent. It should be emphasized that on machines which have a larger discrepancy in the cost of integer versus floating-point operations, the improvement would be less dramatic. The C source code for the RSRP algorithm with dynamic pivoting (Strategy 2) and the loop-unrolling option can be found in Appendix 1 of Geist and Romine [9].

We can apply multiple pivot columns at a time in the CSRP algorithm to achieve a reduction in integer computation as was done for the RSRP algorithm; however, since the pivot search is far more expensive, we cannot achieve competitive factorization times without also pipelining the CSRP algorithm. Unfortunately, while both pipelining and loop-unrolling are effective techniques for reducing execution time, they do not complement each other. Saving pivot columns in order to perform multiple updates reduces the beneficial effects of pipelining. Combining the two techniques also complicates the code considerably.

TABLE 3
Results for loop-unrolling in RSRP

Matrix of order 1024 on 32 processors				
One pivot row applied at a time				
	Implicit pivoting	Explicit pivoting (Chu and George)	Dynamic pivoting (Strategy 1)	Dynamic pivoting (Strategy 2)
Diagonally dominant (no pivot search)	816.4	816.4	816.4	816.4
Diagonally dominant (incl. pivot search)	841.0	841.0	841.0	841.0
Random matrix	945.8	921.3	876.5	852.2
Four pivot rows applied at a time				
	Implicit pivoting	Explicit pivoting (Chu and George)	Dynamic pivoting (Strategy 1)	Dynamic pivoting (Strategy 2)
Diagonally dominant (no pivot search)	604.0	604.0	604.0	604.0
Diagonally dominant (incl. pivot search)	624.9	624.9	624.9	624.9
Random matrix	711.0	715.4	671.8	644.1
Number of exchanges	0	993	471	454

To simplify matters we started by writing the pipelined code (which can be found in Appendix 2 of Geist and Romine [9]) with only two pivot columns applied at a time. In Table 4 we compare the factorization time of this new version of the CSRP algorithm with the execution time of the RSRP algorithm. As the results in Table 4 show, the pipelined CSRP algorithm obtains a smaller improvement than the RSRP algorithm does when loop-unrolling is applied, since multiple updating interferes with the pipelining.

TABLE 4
Results for the RSRP *algorithm.*

Matrix of order 1024 on 32 processors		
One pivot row (column) applied at a time		
	RSRP	CSRP (pipelined)
Random matrix	852.2	804.2
Two pivot rows (columns) applied at a time		
	RSRP	CSRP (pipelined)
Random matrix	698.0	704.8

Table 5 shows the parallel efficiencies of several of the algorithms under consideration. The serial time used here in computing the parallel efficiency is based on the observed execution rate for one processor using a straightforward serial code for LU factorization, coded in C and designed specifically for serial computation on one

processor. For the CSRP and RSRP algorithms with no loop-unrolling this serial execution rate is 0.0286 Mflops. If loop-unrolling is allowed (which also improves the serial code), the serial execution rate increases to 0.040 Mflops. As Table 5 shows, the efficiencies obtained for these algorithms are as high as 97 percent.

TABLE 5
Parallel efficiencies of RSRP *and* CSRP *algorithms*

Matrix of order 1024 on 32 processors			
	RSRP ($r = 1$)	RSRP ($r = 4$)	CSRP (pipelined)
Random matrix	92%	87%	97%

6. Conclusions. We have presented four algorithms for the *LU* factorization of a dense matrix, depending upon the storage of the coefficient matrix and the method of pivoting. The last two algorithms described (which use column pivoting on a matrix stored by rows or columns) were seen to be dual to the first two, and hence we concentrated upon only the first two algorithms. We designed and implemented a number of improvements to these two algorithms, using a randomly generated coefficient matrix of order 1024 as the model problem. We conclude that, in the absence of loop-unrolling, *LU* factorization can be accomplished most rapidly if the coefficient matrix is stored by columns and pivoting is masked by pipelining. If loop-unrolling is allowed and the cost of an integer operation is a substantial fraction of the cost of a floating-point operation, then faster execution is obtained with the coefficient matrix stored by rows and by using dynamic pivoting.

REFERENCES

[1] I. BARRODALE AND G. STEWART, *A new variant of Gaussian elimination*, J. Inst. Maths. Applics., 19 (1977), pp. 39–47.

[2] R. CHAMBERLAIN, *An alternative view of LU factorization with partial pivoting on a hypercube multiprocessor*, in Hypercube Multiprocessors 1987, M. Heath, ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987.

[3] E. CHU AND J. GEORGE, *Gaussian elimination with partial pivoting and load balancing on a multiprocessor*, Parallel Comput., 5 (1987), pp. 65–74.

[4] G. DAVIS, *Column LU factorization with pivoting on a hypercube multiprocessor*, SIAM J. Algebraic Discrete Methods, 7 (1986), pp. 538–550.

[5] J. DONGARRA, F. GUSTAVSON, AND A. KARP, *Implementing linear algebra algorithms for dense matrices on a vector pipeline machine*, SIAM Rev., 26 (1984), pp. 91–112.

[6] J. DONGARRA AND T. HEWITT, *Implementing dense linear algebra algorithms using multitasking on the Cray X-MP-4 (or Approaching the gigaflop)*, SIAM J. Sci. Statist. Comput., 7 (1986), pp. 347–350.

[7] G. GEIST, *Efficient parallel LU factorization with pivoting on a hypercube multiprocessor*, Tech. Rept. ORNL-6211, Oak Ridge National Laboratory, Oak Ridge, TN, 1985.

[8] G. GEIST AND M. HEATH, *Matrix factorization on a hypercube multiprocessor*, in Hypercube Multiprocessors 1986, M. Heath, ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1986, pp. 161–180.

[9] G. GEIST AND C. ROMINE, *LU factorization algorithms on distributed-memory multiprocessor architectures*, Tech. Rept. ORNL/TM-10383, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831, 1987.

[10] M. HEATH AND C. ROMINE, *Parallel solution of triangular systems on distributed-memory multiprocessors*, SIAM J. Sci. Statist. Comput., 9 (1988 (to appear)). Also Tech. Rept. ORNL/TM-10384, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831, March 1987.

[11] G. LI AND T. COLEMAN, *A new method for solving triangular systems on distributed memory message-passing multiprocessors*, Tech. Rept. TR 87-812, Dept. of Computer Science, Cornell University, Ithaca, NY, March 1987.

[12] ———, *A parallel triangular solver for a hypercube multiprocessor*, in Hypercube Multiprocessors 1987, M. Heath, ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987, pp. 539–551. (Also Tech. Rept. TR 86-787, Dept. of Computer Science, Cornell University, Ithaca, NY, October, 1986).

[13] J. ORTEGA AND C. ROMINE, *The ijk forms of factorization methods II: Parallel systems*, Parallel Comput. (to appear), (1988). Also Tech. Rept. RM-87-01, Department of Applied Mathematics, University of Virginia, Charlottesville, VA.

[14] J. ORTEGA AND R. VOIGT, *Solution of partial differential equations on vector and parallel computers*, SIAM Rev., 27 (1985), pp. 149–240.

[15] C. ROMINE, *The parallel solution of triangular systems on a hypercube*, in Hypercube Multiprocessors 1987, M. Heath, ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987, pp. 552–559.

[16] C. ROMINE AND J. ORTEGA, *Parallel solution of triangular systems of equations*, Parallel Comput., 6 (1988), pp. 109–114. Also Tech. Rept. RM-86-05, Department of Applied Mathematics, University of Virginia, Charlottesville, VA.