Gaussian Elimination with Partial Pivoting and Load Balancing on a Multiprocessor*

Eleanor Chu
Alan George

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

CS-85-48

December 1985

* Research supported in part by Canadian Natural Sciences and Engineering Research Council grant A8111, 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., and by the U.S. Air Force Office of Scientific Research under contract AFOSR-ISSA-85-00083.

Gaussian Elimination with Partial Pivoting and Load Balancing on a Multiprocessor*

Eleanor Chu

Alan George

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

A row-oriented implementation of Gaussian elimination with partial pivoting on a local-memory multiprocessor is described. In the absence of pivoting, the initial data loading of the node processors leads to a balanced computation. However, if interchanges occur, the computational loads on the processors may become unbalanced, leading to inefficiency. A simple load balancing scheme is described which is inexpensive and which maintains computational balance in the presence of pivoting. Using some reasonable assumptions about the probability of pivoting occurring, an analysis of the communication costs of the algorithm is developed, along with an analysis of the computation performed in each node processor. This model is then used to derive the expected speed-up of the algorithm. Finally, experiments using a multiprocessor simulator are presented in order to demonstrate the extent to which the analytical model predicts the simulator results.

*Research supported in part by Canadian Natural Sciences and Engineering Research Council under grant A8111, 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., and by the U.S. Air Force Office of Scientific Research under contract AFOSR-ISSA-85-00083.

Table of Contents

1. Introduction 1

2. The Algorithm 2

   Host Process 5

   Node Process 6

3. Implementation Issues 7

4. Performance Analysis 9

   4.1 Communication Message Complexity 10

   4.2 Communication Volume 13

   4.3 Estimation of Efficiency 15

5. Numerical Experiments 17

6. References 23

7. Appendix A: Control Diagrams 25

8. Appendix B: Program Listing 26

1. Introduction

In this paper we present a concurrent algorithm for solving dense linear systems of equations on a local-memory multiprocessor. The algorithm is based on Gaussian elimination with partial pivoting, and relies significantly upon previous work by Geist [3]. Thus, this paper can be regarded as a natural sequel to that article, and we assume that the reader is familiar with the arguments and results found there.

Briefly, Geist presents arguments to support implementing Gaussian elimination with partial pivoting in a *row-oriented manner*. That is, the rows of the matrix are distributed to each node processor, and the computation is performed row by row. Other alternatives include distributing the matrix by columns or by sub-matrices. A major argument in support of the row-oriented scheme is that it allows the efficient implementation of the triangular solution phase, an advantage not shared by alternate schemes.

A disadvantage is that determining the pivot element is difficult, because the rows are distributed among multiple processors. Geist's solution involves using the (otherwise more-or-less idle) host to aid in determining the pivot row, and he is able to effectively overlap the associated communication cost with the computation.

A potential weakness of Geist's implementation is that it does not deal with the possible "unbalancing" of the computation that could be caused by an unfortunate sequence of pivot choices. He provides experimental evidence that suggests that the cost of this imbalance will normally be low, typically around 10% in utilization and 5% to 14% in execution time in the factorization phase. However, there are examples where it is much higher. We discuss this in more detail in section 2.

Our contribution is to show that Geist's general approach can be modified so that the node processor loads remain balanced, irrespective of the pivot sequence chosen. In addition, we have devoted considerable effort to the design of the software, and our implementation exhibits a high degree of modularity. A control diagram and a listing of the code are included in Appendix A and Appendix B respectively.

Finally, using some reasonable assumptions about the probability of pivoting occurring, a model for the communication costs of the algorithm is developed, along with an analysis of the computation performed in each node processor. This model is then used to derive the expected *speed-up* of the algorithem. Experiments using a multiprocessor simulator are presented in order to demonstrate the extent to which the analytical model predicts the simulator results.

Although the numerical experiments discussed in this paper were conducted on a simulator for a parallel machine based on the hypercube topology, our algorithm neither exploits nor depends on this particular interconnection. The proposed algorithm and its analysis can be applied to any multiprocessor system whose processing nodes can synchronize and communicate with each other, and with the host, via a message passing scheme. Two message-passing primitives, *send* and *await*, are assumed to be available.

2. The Algorithm

We consider the problem of solving systems of linear equations on a local-memory multiprocessor having p processing nodes, where p is assumed to be much less than n, the order of the system. The algorithm is based on a row variant of Gaussian elimination with partial pivoting. We denote the system to be solved by

$$A\, x = b \quad .$$

A serial version of the algorithm (in pseudo code) is given on page 3.

During the procedure, the coefficient matrix A is overwritten by the triangular factors, and at the end of the execution of the algorithm the right hand side b contains the solution.

As usual with parallel algorithms, we would like to achieve a balanced distribution of work load and a low volume of data movement and communication. In addition, we would like our software implementation to have a flexible and convenient calling sequence and a modular design.

A uniform work load distribution and a low communication cost contribute directly to the *speed-up*, which is the ultimate goal of a concurrent algorithm. However, a flexible calling sequence and a modular design are also very important. Ideally, a parallel algorithm for solving a class of problems should be as easy and as flexible to use as its serial counterpart, and any complication in its implementation should, to the extent possible, be made transparent to the user. More on various implementation issues will be discussed in section 3.

It is desirable to separate the algorithm into two parts, namely the factorization phase, and the solution of the triangular systems derived during the factorization. It is often necessary to solve numerous systems of equations which differ only in their right hand sides. With these two phases designed as independent software modules, one need factor the matrix only once in order to solve these systems. The algorithm proposed in this article retains this important feature while maintaining parallelism and work load balance in both phases.

```
for $k$ := 1 to $n-1$ do            /* numerical factorization phase */
begin
    $pivot$ := $k$
    for $i$ := $k+1$ to $n$ do
        if $|a_{ik}| > |a_{kk}|$ then $pivot$ := $i$

    if $pivot \neq k$ then /* interchange row $k$ and row $pivot$ */
        for $j$ := 1 to $n$ do
        begin
            $temp$ := $a_{kj}$
            $a_{kj}$ := $a_{pivot,j}$
            $a_{pivot,j}$ := $temp$
        end

    $perm_k$ := $pivot$

    for $i$ := $k+1$ to $n$ do
    begin
        $a_{ik}$ := $a_{ik}/a_{kk}$
        for $j$ := $k+1$ to $n$ do
            $a_{ij}$ := $a_{ij} - a_{ik} * a_{kj}$
    end
end

Permute $b$ according to $perm$.

for $k$ := 1 to $n-1$ do            /* forward substitution */
    for $i$ := $k+1$ to $n$ do
        $b_i$ := $b_i - b_k * a_{ik}$

for $k$ := $n$ to 1 do              /* backward substitution */
begin
    $b_k$ := $b_k/a_{kk}$
    for $i$ := 1 to $k-1$ do
        $b_i$ := $b_i - b_k * a_{ik}$
end
```

We assume in the sequel that the coefficient matrix is read in or generated by the host process. Since there is no globally shared memory, the data must be distributed among the processing nodes in some way, typically either by rows or by columns. *In either case*, there is a decision to be made concerning the way in which the rows or columns are *mapped onto the processors.* For example, *block-mapping* may be used, where the first n/p rows (or columns) are assigned to processor 1, the next n/p rows (or columns) are assigned to processor 2, and so on. Alternatively, *wrap-mapping* may be used, where consecutive rows (or columns) are assigned to consecutive processors, with assignment "wrapping around" to processor 1 after a row (or column) is assigned to processor p.

To reiterate, there are two issues: whether the data is distributed by rows or columns, and the way in which the rows (or columns) are mapped onto the processors.

Discussion about various mapping strategies can be found in [1,4,5,6]. In the case of column-oriented Cholesky decomposition or column-oriented Gaussian elimination with partial pivoting, the work load distribution is statically determined by the initial data mapping. It was found in [1,4,6] that either wrap-mapping or reflection mapping is quite effective in this context. Since a distribution of the triangular factors by rows is much more desirable in connection with parallel triangular solution, a row-oriented algorithm was proposed in [3] for Gaussian elimination with partial pivoting on a hypercube multiprocessor. However, with the pivot row dynamically chosen during the factorization process, the work load distribution is no longer dictated by the initial data mapping.

For example, irrespective of the way in which the rows are mapped onto the processors, it is possible that the first n/p pivot rows are chosen from the same processor, after which it would remain idle until the factorization was completed. In the worst case, the permutation in effect can turn an initial wrap-mapping into a block-mapping, which can lead to a 50% increase in execution time [3].

Of course in general this is unlikely to occur in practice. Instead, as Geist argues, it is more likely that the pivots will be selected more or less evenly from all of the processors, and the work load will therefore remain roughly balanced. According to the performance results reported in [3] for some random matrices, the penalty of an unbalanced work load caused by the pivoting process appears to be 10% in the average utilization, and the increase in execution time with random pivoting was observed to be in the range of 5—14% in the factorization phase, and somewhat more if the solution phase is included.

The algorithm proposed in this article eliminates this penalty by dynamically balancing the load. It does so by explicitly performing the row interchanges so that each processor node retains approximately the same number of uneliminated rows. We shall see that the modest amount of communication involved in performing the interchanges can be done in such a way that it does not affect the overall execution time very much.

In designing the parallel algorithm, there is also a decision to be made concerning how much communication in addition to data allocation the host should handle. Besides the consideration that the host can only send or receive messages sequentially, we also note that on some machines, the host-to-node communication is much slower than the node-to-node communication. In the algorithm we propose in this article, besides initial data loading we have involved the host only in the process of determining the pivot row for each elimination stage. There is sufficient parallelism in the work done by the host and other nodes at that point to make this a reasonable strategy.

The algorithm is given below in a form which describes the functions of the host and the node processes, but leaves the issue of *how to accomplish them* to section 3, where we will discuss some implementation details.

Host Process

Initialization

1. Compose and broadcast the mapping information to all nodes.
2. Distribute the rows of the coefficient matrix to the nodes according to the chosen mapping scheme.

Factorization

Repeat the following steps for each elimination stage:

1. Receive one pivot candidate from each active node.
2. Determine the pivot row and inform all active nodes of its number.

Triangular solution

1. Permute the right hand side according to the pivoting sequence.
2. Broadcast the permuted right hand side to all nodes.
3. Receive the solution element-by-element from the nodes.

Node Process

Initialization

1. Receive the mapping information from the host.
2. Receive its share of rows from the host.
3. Determine the maximum absolute value of the first column among the rows it receives, and send this value and its row number to the host.

Factorization (k-th step)

1. Wait for the pivot row number from the host.
2. If the pivot row is in this processor, immediately broadcast the pivot row to all active nodes.
3. Check whether the kth row and the pivot row are the same. If they are different but are both located in this node, interchange them (without communication cost). If either the k-th row or the pivot row is located in this node, interchange them via message passing. In any case, at most two nodes are involved in the interchange.
4. Check whether any rows remain to be modified. If so, wait to receive the pivot row. (Otherwise, become inactive until triangular solution phase begins.)
5. After receiving the pivot row, compute the multiplier and update the elements in the following column (i.e., column $k+1$). Send the element of maximum absolute value in this column to the host as pivot candidate for the next stage of elimination.
6. Modify the remaining rows.

Triangular solution

1. Wait for the permuted right hand side.
2. The node process which has the first row of the triangular matrices will immediately broadcast the first element of the right hand side, which is the first element in the solution of the lower triangular system, to every node which requires it in order to modify its share of elements in the right hand side. Therefore, during the kth stage of forward substitution, the node process which has the kth row of data will broadcast the kth element in the modified right hand side to all active nodes. The right hand side will then be modified by all active nodes concurrently. This is repeated until the forward substitution is completed.

3. The node process which has the last row of triangular matrices will initiate the backward substitution by computing the last element of the solution and broadcast it to all active nodes. Similarly, during the k-th stage of backward substitution, the node process which has the $(n-k+1)$*—st* row will compute the $(n-k+1)$*—st* element of the solution and broadcast it to all active nodes. The right hand side will then be modified by all active nodes concurrently. The node process which computes the solution at each stage will also send the element back to the host before proceeding with the next stage. This is repeated until the first element of the solution is computed and sent back to host.

Note that in the algorithm above, the transformations to each row must be carried out sequentially. That is, *row* k is modified by *rows* 1, 2, 3, ..., and $k-1$ in strict order. Fortunately, this still allows sufficient parallelism, and the synchronization thus imposed guarantees that *row* k must have been received by all nodes before the pivot row for *stage* $k+1$ can be determined.

Also note that at the end of the numerical factorization, each node process will have the rows of the triangular factors of a permuted form of A ready for the solution phase, and the matrix A in the host need not be overwritten by the factors.

Although the above algorithm is described for solving a single system, to solve many systems with the same coefficient matrix and a different right hand side, one would need only to repeat the triangular solution phases of the host and node processes for each right hand side.

3. Implementation Issues

In our implementation, we do not exploit the hypercube topology imitated by the simulator. Although the communication cost might not be optimal for the hypercube simulator, this approach will allow our code to be potentially portable to other local-memory multiprocessors which support host-to-node and node-to-node communication in some other way.

Two message-passing primitives, *send* and *await*, are assumed to be available. For consistency with Geist[3], and the simulator we used, we assume that execution of a *send* does not cause the sending process to wait (e.g., for a reply); execution of its program continues as soon as the message leaves the sending node. Execution of an *await*, however, blocks further execution of the receiving process until the message is received.

Our algorithm involves identical programs for each node process. However, the portion of the code actually executed by different nodes is not necessarily identical.

With the help of a dynamic load balancing scheme, the work load distribution of our algorithm is not affected to an appreciable degree by the pivoting process. We therefore adopted the wrap-mapping scheme in our implementation, which had been shown to be effective in yielding high nodal utilization for column oriented algorithms in [1,4,5].

The row-to-node mapping is contained in an array in our implementation, and is therefore a "parameter" in the code. By allowing each node to store the data mapping information, our implementation can be conveniently used with any row mapping scheme. Although the performance results do depend on the particular mapping scheme, our algorithm and the programming logic are both independent of the mapping scheme being used. This approach not only makes the program easier to understand but also allows one to experiment with different mapping schemes by simply calling a different procedure to compose the map.

Since the development and maintenance of parallel numerical software is unlikely to be easier than its serial counterpart, the importance of a good software design remains. Although a modular design usually entails the overhead of more procedure calls, the tradeoff in execution time is worthwhile since such software is easier to understand and to maintain.

The program running on each individual processor is essentially a piece of serial code interleaved with message passing primitives. In particular, the computing done on each node is essentially the basic operations of numerical linear algebra including dot products, elementary vector operations, vector copy and swap, vector norms, vector scaling, and the determination of the index of the vector component of largest magnitude. These basic operations have been implemented as low level subprograms in a package called the BLAS (Basic Linear Algebra Subprograms) [7], and have been used in the implementation of LINPACK [2], a widely used package for solving dense linear algebra problems. We have converted some of the routines in the BLAS to C programs and used them in our implementation of a number of parallel linear algebra algorithms including the one described in this article.

4. Performance Analysis

We begin our analysis by identifying and relating various common performance measurements. As usual, our primary objective is to attain maximum *speed-up*. That is, given a p-processor machine, we would like to solve our problem in time that is as close as possible to a factor of p less than that needed to solve the same problem on a single processor version of the machine, using the best serial algorithm available. We assume that the single processor machine has adequate memory, presumably much more than that available to a single processor in the multiple processor configuration. We also assume that all processors in the machine have the same execution speed.

We adopt the following notation.

T_s: execution time of the best serial program.

T: execution time of the parallel program running on p nodes.

T_a: the average computation time of a node process.

T_c: the average time spent by a node process in sending/receiving messages.

T_i: the average idle time of a node process.

We note that

$$T = T_a + T_c + T_i \tag{4.1}$$

and

$$T \geq T_a \geq \frac{T_s}{p} \tag{4.2}$$

We shall assess an implementation of a parallel algorithm by its *efficiency*, *speedup*, *average nodal utilization*, and *parallel overhead*, where

$$speedup \equiv \frac{T_s}{T} \tag{4.3}$$

$$efficiency \equiv \frac{speedup\ on\ p\ nodes}{p} \tag{4.4}$$

$$nodal\ utilization \equiv \frac{busy\ time\ of\ a\ node\ processor}{T} \tag{4.5}$$

$$average\ nodal\ utilization \equiv \frac{\sum_{k=1}^{p} nodal\ utilization\ of\ node\ k}{p} \tag{4.6}$$

$$= \frac{T_a + T_c}{T}$$

$$= 1 - \frac{T_i}{T}$$

$$\textit{parallel overhead} \equiv \textit{average nodal utilization} - \textit{efficiency} \qquad (4.7)$$

$$\equiv \frac{T_a + T_c - T_s/p}{T}$$

It is now clear that

$$\textit{speedup} = p \text{ and } \textit{efficiency} = 1 \ \textit{iff} \ T_a = \frac{T_s}{p}, \text{ and } T_c = T_i = 0.$$

$$\textit{average nodal utilization} = 100\% \ \textit{iff} \ T_i = 0.$$

Therefore, a parallel program with high average nodal utilization does not necessarily have high speed-up. When a node processor is busy, it is either doing arithmetic computation or sending/receiving data. The idle time of a node processor can be caused by an unbalanced workload and/or by the transmission delays in passing messages. In particular, with the hypercube-like topology, messages tend to traverse different paths of different length and so exhibit varied latencies. We note that the computation and data communication on any individual processor must be carried out sequentially, and the execution time of a parallel program is determined by the process that finishes last.

We shall provide an analytical performance model for the algorithm we proposed in section 2. Comparison will then be made with the actual performance of the program running on a hypercube simulator.

4.1 Communication Message Complexity

We first derive the maximum, minimum and average number of messages sent and received by an individual processor during the phases of numerical factorization and triangular solutions. For concreteness, we consider solving a system of order n on a multiprocessor of p nodes, where $p \leq n$. For convenience, we assume that n is an integral multiple of p.

For the wrap-mapping scheme we have chosen to employ, each node processor will be allocated n/p rows of the coefficient matrix A, which are to be overwritten by the corresponding rows of the triangular factors of a permuted form of A. The map is composed in the host and sent to each node processor. Therefore, the number of messages sent by the host and received by each node processor for this purpose is given by

$$HostSend_A = n + p \tag{4.8}$$

$$NodeRecv_A = \frac{n}{p} + 1 \tag{4.9}$$

During the process of numerical factorization, each pivot row must be broadcast to every node that needs to transform its remaining rows. Therefore, each *row* i with $i \leq n - p$ will be sent to p nodes including itself, and each *row* i with $i = n - p + k$, $1 \leq k \leq p$ will be sent to $p - k$ nodes. (In order to keep our code simple and easy to understand, nodes actually do send rows to themselves. There appears to be little loss in efficiency by doing so.) Since every node processor has n/p rows with exactly one row belonging to the last p rows of the matrix, we have

$$\left(\frac{n}{p} - 1\right) \times p \leq NodeSend_{piv-row} \leq \left(\frac{n}{p} - 1\right) \times p + p - 1$$

That is

$$n - p \leq NodeSend_{piv-row} \leq n - 1, \tag{4.10}$$

with

$$average = \frac{1}{p} \times \sum_{i=1}^{p} (n - p + i - 1)$$
$$\approx n - \frac{p}{2}$$

and

$$n - p \leq NodeRecv_{piv-row} \leq n - 1, \textit{ with an average} \approx n - \frac{p}{2} \tag{4.11}$$

With the dynamic load balancing scheme, we have to account for the possible communication cost of permuting rows residing in two different nodes. If we assume that this is necessary at every elimination stage, then in the worst case we have

$$\frac{n}{p} \leq NodeSend_{perm} \leq n - 1, \textit{ with an average} \approx \frac{n}{2} + \frac{n}{2p} \tag{4.12}$$

and

$$\frac{n}{p} \leq NodeRecv_{perm} = n - 1, \textit{ with an average} \approx \frac{n}{2} + \frac{n}{2p} \tag{4.13}$$

In addition, all node processors will be participating in the process of determining the pivot row at each elimination stage. Each node processor will send its pivot candidate to the host $NodeSend_{select}$ times, and will be informed by the host of the pivot row number $NodeRecv_{select}$ times, where

$$n - p + 1 \leq NodeSend_{select} \leq n - 1, \text{ with an average} \approx n - \frac{p}{2} \tag{4.14}$$

$$n - p + 1 \leq NodeRecv_{select} \leq n - 1, \text{ with an average} \approx n - \frac{p}{2} \tag{4.15}$$

$$\begin{aligned} HostRecv_{select} &= (n-p) \times p + \sum_{j=2}^{p} j \\ &= (n-p) \times p + \frac{(p-1) \times (p+2)}{2} \\ &= np - \frac{1}{2}p^2 + \frac{1}{2}p - 1 \end{aligned} \tag{4.16}$$

$$\begin{aligned} HostSend_{select} &= (n-p) \times p + \sum_{j=2}^{p} j \\ &= (n-p) \times p + \frac{(p-1) \times (p+2)}{2} \\ &= np - \frac{1}{2}p^2 + \frac{1}{2}p - 1 \end{aligned} \tag{4.17}$$

Combining equations (4.8) to (4.17), the total number of messages sent and received by the host and each node processor during the factorization phase is

$$\begin{aligned} HostSend_{fact} &= (n + p) + \left(np - \frac{1}{2}p^2 + \frac{1}{2}p - 1 \right) \\ &= np + n - \frac{1}{2}p^2 + \frac{3}{2}p - 1 \end{aligned} \tag{4.18}$$

$$HostRecv_{fact} = np - \frac{1}{2}p^2 + \frac{1}{2}p - 1 \tag{4.19}$$

$$2n + \frac{n}{p} - 2p + 1 \leq NodeSend_{fact} \leq 3n - 3 \tag{4.20}$$

$$2n + 2\frac{n}{p} - 2p + 2 \leq NodeRecv_{fact} \leq 3n + \frac{n}{p} - 2 \tag{4.21}$$

The host then initiates the triangular solution phase on the node processors by sending them the permuted right hand side $\tilde{b}$, which is sent to each node as a whole, because the saving in sending p messages of n real numbers each instead of n messages of one real number each could be significant when $p \ll n$. During the process of forward and backward substitutions, each computed element of the solution must be sent to each node process which needs to modify the remaining elements in the right hand side. The final solution is then sent back to the host. We thus have

$$2\times\left(\frac{n}{p}-1\right)\times p+\frac{n}{p}\leq NodeSend_{solv}\leq 2\times\left\{\left(\frac{n}{p}-1\right)\times p+p-1\right\}+\frac{n}{p}$$

That is

$$2n+\frac{n}{p}-2p\leq NodeSend_{solv}\leq 2n+\frac{n}{p}-2 \tag{4.22}$$

$$2n-2p+1\leq NodeRecv_{solv}\leq 2n-1 \tag{4.23}$$

$$HostSend_{solv}=p \tag{4.24}$$

$$HostRecv_{solv}=n \tag{4.25}$$

From equations (4.10) to (4.15), (4.22) and (4.23), we obtain the average number of messages sent and received by each node processor, as shown below.

$$AvgSend_{nodefct}\approx 2.5n+0.5\frac{n}{p}-p \tag{4.26}$$

$$AvgRecv_{nodefct}\approx 2.5n+1.5\frac{n}{p}-p \tag{4.27}$$

$$AvgSend_{nodesolv}\approx 2n+\frac{n}{p}-p-1 \tag{4.28}$$

$$AvgRecv_{nodesolv}\approx 2n-p \tag{4.29}$$

4.2 Communication Volume

The communication volume is determined by the number of messages, the size of each message, and the path length traversed by each message. We now give the type and the average size of each message as below.

message type	average size
Map	n integers
Row of A	n reals
Local max	1 real
Pivot row number	1 integer
Pivot row	$n/2$ reals
Row permuted	n reals
Right hand side	n reals
Solution component	1 real

The hypercube simulator was run on a DEC VAX 11/780, where the size of an integer is 2 bytes, and the size of a single-precision floating point number is 4 bytes. The average communication volume of each node process, assuming a path length of one, is given below in bytes.

From (4.10), (4.12) and (4.14), we obtain

$$\begin{aligned} AvgSndVol_{nodefct} &\approx 4\times\left(n - \frac{p}{2}\right)\times\frac{n}{2} + 4\times\left(\frac{n}{2} + \frac{n}{2p}\right)\times n \\ &\quad + 4\times\left(n - \frac{p}{2}\right)\times 1 \\ &\approx 4n^2 + 2\frac{n^2}{p} - np + 4n - 2p \end{aligned} \tag{4.30}$$

From (4.9), (4.11), (4.13) and (4.15), we obtain

$$\begin{aligned} AvgRcvVol_{nodefct} &\approx 4\times\frac{n}{p}\times n + 2\times 1\times n + 4\times\left(n - \frac{p}{2}\right)\times\frac{n}{2} \\ &\quad + 4\times\left(\frac{n}{2} + \frac{n}{2p}\right)\times n + 2\times\left(n - \frac{p}{2}\right)\times 1 \\ &\approx 4n^2 + 6\frac{n^2}{p} - np + 4n - p \end{aligned} \tag{4.31}$$

From (4.28), we obtain

$$AvgSndVol_{nodesolv} \approx 4\times\left(2n + \frac{n}{p} - p - 1\right)\times 1 \tag{4.32}$$

$$\approx 8n + 4\frac{n}{p} - 4p - 4$$

From (4.29), we obtain

$$AvgRcvVol_{nodesolv} \approx 4\times(2n - p)\times 1 + 4\times 1\times n \qquad (4.33)$$

$$\approx 12n - 4p$$

4.3 Estimation of Efficiency

Our aim is to provide an analytical performance model for the parallel program running on the hypercube simulator. From equations (4.1) to (4.7), we see that the *speed-up*, *efficiency*, *average nodal utilization*, and the *parallel overhead* can all be determined given T_s, the execution time of the best serial program, T_a, the average node computation time, T_c, the average node communication time, and T_i, the average node idle time. Our task is therefore to provide estimates for T_s, T_a, T_c, and T_i. Since the performance model is for a parallel program running on a hypercube simulator, the following simulator characteristics reported in [4] are noted in modelling and interpreting the performance results.

a) The hypercube simulator supports only one process per processor. We therefore make no distinction between process and processor in our model.

b) The run time reported by the simulator is a count of the number of VAX assembler-level instructions executed by the user code.

c) Since the hypercube connection is not complete, message passing between some pairs of nodes is accomplished by routing messages through a sequence of intermediate nodes. Although the arrival time of a message at the destination node is approximated in the simulator by imposing a transmission delay on each message that is proportional to the message path length and the length of the message (in bytes), the cost of interrupting each of the intermediate nodes and the cost of relaying the messages by each intermediate node are both ignored by the simulator.

d) The *send* and *await* message passing primitives provided by the simulator are each counted as one instruction only, as is any subroutine call to the *C* library.

e) The simulator allows the user to specify a desired ratio of communication time to computation time. A ratio of 1.0 means that a floating-point number can be transmitted between adjacent nodes in about the time required for one floating-point arithmetic operation by a node processor.

f) The simulator assumes that passing a message between the host and any node costs the same as passing the same message between an arbitrary pair of adjacent nodes.

Our analysis involves estimates in units of additive and multiplicative floating-point operations. For Gaussian elimination with partial pivoting together with the triangular solution, we approximate the total additive and multiplicative operations by

$$T_s \approx \frac{2}{3}n^3 + \frac{3}{2}n^2 \tag{4.34}$$

and

$$T_a \approx \frac{T_s}{p} \tag{4.35}$$

According to simulator characteristic d) above, the average communication time, T_c, can be approximated by the sum of the average number of *send*'s and *receive*'s performed by an individual node in the factorization and triangular solution phase. From equations (4.26) to (4.29), we have

$$T_c \approx 9n + 3\frac{n}{p} - 4p - 1 \tag{4.36}$$

In approximating the average idle time, T_i, we shall assume a perfect load balancing and estimate T_i by the transmission delays only. Since each node processor must receive its share of data before the computation can begin, the initial delay due to data loading is taken into account in our model. During the course of computation, we assume that whenever the data is needed the first byte of the data message has just arrived at the destination. The latter assumption implies that the delay time will be as long as having the rest of the data transmitted between two adjacent nodes.

According to simulator characteristics e) and f) above, if we let α denote the ratio of communication time to computation time, and recalling that one floating point number is 4 bytes on a VAX 11/780, we have

$$T_i \approx \frac{\alpha}{4} \times \left(4n^2 + \left(AvgRcvVol_{nodefct} - \frac{4n^2}{p} \right) + AvgRcvVol_{nodesolv} \right) \tag{4.37}$$

$$\approx \frac{\alpha}{4} \times \left(8n^2 + 2\frac{n^2}{p} - np + 16n - 5p \right)$$

$$\approx \alpha \times \left(2n^2 + \frac{n^2}{2p} \right)$$

We can now estimate the *analytical efficiency* as shown below.

$$analytical\ efficiency \approx \frac{\frac{T_s}{p}}{T_a + T_c + T_i} \tag{4.38}$$

$$\approx \frac{\frac{2n^3}{3p} + \frac{3n^2}{2p}}{\frac{2n^3}{3p} + \frac{3n^2}{2p} + 9n + 3\frac{n}{p} + \alpha \times \left(2n^2 + \frac{n^2}{2p} \right)}$$

$$\approx \frac{4n + 9}{4n + 12\alpha p + 3\alpha + 9}, \quad \text{if } \alpha \geq 1 \text{ and } p > 1.$$

In particular, taking $\alpha = 1$ we have

$$analytical\ efficiency_{\alpha=1} \approx \frac{4n + 9}{4n + 12p + 12}. \tag{4.39}$$

We note that if $p = 1$, then α is effectively zero, and the $O(n)$ average communication time, T_c, cannot be ignored. Thus, we have

$$analytical\ efficiency_{p=1} \approx \frac{4n^2 + 9n}{4n^2 + 9n + 72} \tag{4.40}$$

5. Numerical Experiments

Our numerical experiments were designed to demonstrate

1. the effectiveness of the implemented load balancing scheme,
2. the extent to which the actual logical communication count reported by the simulator is predicted by the message complexity approximations we derived in section 4.1,

3. how the actual performance of the parallel program on the hypercube simulator corresponds to the *analytical efficiency* derived in section 4.

Our experiments were performed on a VAX 11/780 running a binary hypercube simulator written by T.H. Dunigan of the Oak Ridge National Laboratory. The run time reported is a count of the number of VAX assembler-level instructions executed by the user code. We have used 1.0 as the ratio of communication time to computation time in all our experiments. For further details about the simulator, and background information about hypercube multiprocessors, see [4].

Table 1 shows the performance results from two implementations of the dynamic load balancing scheme for the numerical factorization phase. Strategy 1 can be viewed as a straightforward implementation of the serial algorithm described in section 2 for a parallel machine. That is, the interchange of rows resulting from the pivoting process is made before each elimination stage begins. Strategy 2 allows the pivot row to be broadcast before the interchange is actually made. By exploiting the fact that messages do not have to be received in the same order as they arrive at the destination, strategy 2 can be implemented with clean logic. Our experimental results in Table 1 show that the average nodal utilization is increased by up to 9% and the execution time is decreased by as much as 12.5% when strategy 2 is adopted. In the following tables, 'migrations' indicate the number of row interchanges between different nodes.

Table 1
Factorization Phase

			Strategy 1			Strategy 2		
n	p	migrations	time	utilization	sp_up	time	utilization	sp_up
64	4	49	383012	85%	2.9	365508	89%	3.0
128	4	90	2391259	92%	3.5	2313673	95%	3.6
128	8	112	1462134	82%	5.7	1357387	88%	6.1
128	16	122	1153358	60%	7.2	1009124	69%	8.2

Table 2 compares the results from performing Gaussian elimination with partial pivoting on randomly generated matrices and diagonally dominant matrices. Since row interchanges will not occur in the latter case, the difference in their respective performance indicates the effectiveness of our dynamic load balancing scheme. From the experimental results presented in Table 2, we see that the execution time is increased only 1%, and the average nodal utilization is not affected at all for $n = 128$ with $p = 4$ and $p = 8$. Since the interchange of rows actually happens 90 and 112 times respectively on the randomly generated matrices for these two cases, the dynamic load balancing scheme appears to be very effective. We note that the utilization results reported in [3, Fig.3] for these two cases are approximately 87% and 79%. It was also reported in [3] that random pivoting appeared to cause utilization 10% lower and execution time 5—14% longer than the diagonally dominant case.

Table 2

Factorization Phase

		Random Matrices				Diagonally Dominant Matrices		
n	p	migrations	time	utilization	sp_up	time	utilization	sp_up
64	1	0	1147491	99%	0.96	1111102	99%	0.94
64	4	49	365508	89%	3.0	350439	91%	3.0
128	1	0	8372077	99%	0.99	8226315	99%	0.97
128	4	90	2313673	95%	3.6	2282619	95%	3.5
128	8	112	1357387	88%	6.1	1338271	88%	6.0
128	16	122	1009124	69%	8.2	944246	73%	8.5

Table 3 compares the experimental results of performing factorization alone against the results including the triangular solution phase. The reported average nodal utilization for each test problem indicates that high parallelism is effectively maintained in both phases. Since the amount of arithmetic computation performed on each node for forward/backward substitution is $O(n^2/p)$, its effect on masking communication is less compared to the $O(n^3/p)$ computation performed in the factorization phase, the lower *overall speed-up* is expected. For the four test problems, the percentage of run time dedicated to the solution phase varies from 9.6% for $n = 128$, $p = 4$, with $n/p = 32$ to 17% for $n = 128$, $p = 16$, with $n/p = 8$.

Table 3

			Factorization			Factorization + Solve		
n	p	migrations	time	utilization	sp_up	time	utilization	sp_up
64	4	49	365508	89%	3.0	435974	91%	2.7
128	4	90	2313673	95%	3.6	2559283	96%	3.3
128	8	112	1357387	88%	6.1	1575928	89%	5.4
128	16	122	1009124	69%	8.2	1217544	72%	7.0

Table 4 and Table 5 compare the performance of our row-oriented algorithm with the performance of a column-oriented algorithm, which we have implemented to perform Gaussian elimination with partial pivoting on a local-memory multiprocessor in the same spirit as the column-oriented sparse Cholesky algorithm described in [5]. In either case, the *speed-up* is measured using the same serial program. The experimental results presented in Table 4 indicate that by incorporating the load balancing scheme, the row-oriented algorithm has achieved higher *speed-up* than the column-oriented algorithm in the factorization phase. Note that the apparently higher utilization of the column-oriented algorithm does not result in higher *speed-up*. As noted earlier, high utilization is not a sufficent condition for high *speed-up*.

Table 4

Factorization Phase

		Row-oriented Algorithm				Column-oriented Algorithm		
n	p	migrations	time	utilization	sp_up	time	utilization	sp_up
64	4	49	365508	89%	3.0	400512	96%	2.8
128	4	90	2313673	95%	3.6	2521241	98%	3.3
128	8	112	1357387	88%	6.1	1478066	95%	5.6
128	16	122	1009124	69%	8.2	1007964	85%	8.2

Table 5 compares the performance of the row-oriented algorithm and the column-oriented algorithm including the triangular solution phase. We note that in order to carry out the forward/backward substitution in parallel while retaining the factorization phase and the triangular solution phase as two independent tasks, the rows of the triangular factors need to be distributed among the nodes at the end of the column-oriented factorization algorithm. The results presented in Table 5 show that the need to redistribute data before starting the solution phase makes the column-oriented algorithm less desirable in both *speed-up* and *average nodal utilization.*

Table 5

Factorization + Solve

		Row-oriented Algorithm				Column-oriented Algorithm		
n	p	migrations	time	utilization	sp_up	time	utilization	sp_up
64	4	49	435974	91%	2.7	536138	75%	2.2
128	4	90	2559283	96%	3.3	2989983	83%	2.8
128	8	112	1575928	89%	5.4	1933489	72%	4.4
128	16	122	1217544	72%	7.0	1501243	56%	5.6

Table 6 compares the average logical communication count obtained from numerical experiments using random matrices against the formulae for $AvgSend_{nodefct}$ and $AvgSend_{nodesolv}$ derived in section 4, i.e., equations (4.26) and (4.28). Since interchange of rows was assumed to be always necessary in our model, and the maximum number of interchanges for an individual node was obtained from the worst case analysis, the analytical approximation is expected to be an upper bound for all cases. Our experimental results in table 6 appear to verify our expectation. We note that the overestimation is within 10% in all cases reported in Table 6.

Table 6

Factorization + Solve

n	p	n/p	AvgSend nodefct+nodesolv	Measured average count
16	2	8	79	73
20	2	10	100	95
30	3	10	143	136
40	4	10	186	175
50	5	10	229	213
100	10	10	444	405
32	2	16	163	149
40	2	20	205	191
60	2	30	310	296
64	2	32	331	312
80	2	40	415	391
100	2	50	520	493

Table 7 compares the *analytical efficiency* given by equation (4.39) for $p>1$, $\alpha=1$ and equation (4.40) for $p=1$ against the actual efficiency from numerical experiments on random matrices of different order on different number of processors. The *analytical efficiency* appears to closely approximate the *simulator efficiency* when $p \ll n/p$.

Table 7

Factorization + Solve

n	p	n/p	Analytical efficiency	Measured efficiency
16	2	8	0.73	0.51
128	16	8	0.73	0.44
20	2	10	0.77	0.59
30	3	10	0.77	0.60
40	4	10	0.77	0.60
50	5	10	0.77	0.61
100	10	10	0.77	0.62
32	2	16	0.84	0.76
64	4	16	0.84	0.68
128	8	16	0.84	0.68
40	2	20	0.86	0.80
50	2	25	0.90	0.86
64	2	32	0.91	0.89
128	4	32	0.91	0.83
80	2	40	0.92	0.92
100	2	50	0.94	0.94
10	1	10	0.87	0.58
64	1	64	0.99	0.90
128	1	128	1.00	0.95

6. References

[1] G. J. DAVIS, "Column LU factorization with pivoting on a hypercube multiprocessor", Technical Report 6219, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831 (1985).

[2] J. J. DONGARRA, C. B. MOLER, J. R. BUNCH, AND G. W. STEWART, *LINPACK users' guide,* SIAM, Philadelphia (1980).

[3] G. A. GEIST, "Efficient parallel LU factorization with pivoting on a hypercube multiprocessor", Technical Report ORNL-6211, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831 (1985).

[4] G. A. GEIST AND M. T. HEATH, "Parallel Cholesky factorization on a hypercube multiprocessor", Technical Report 6190, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831 (1985).

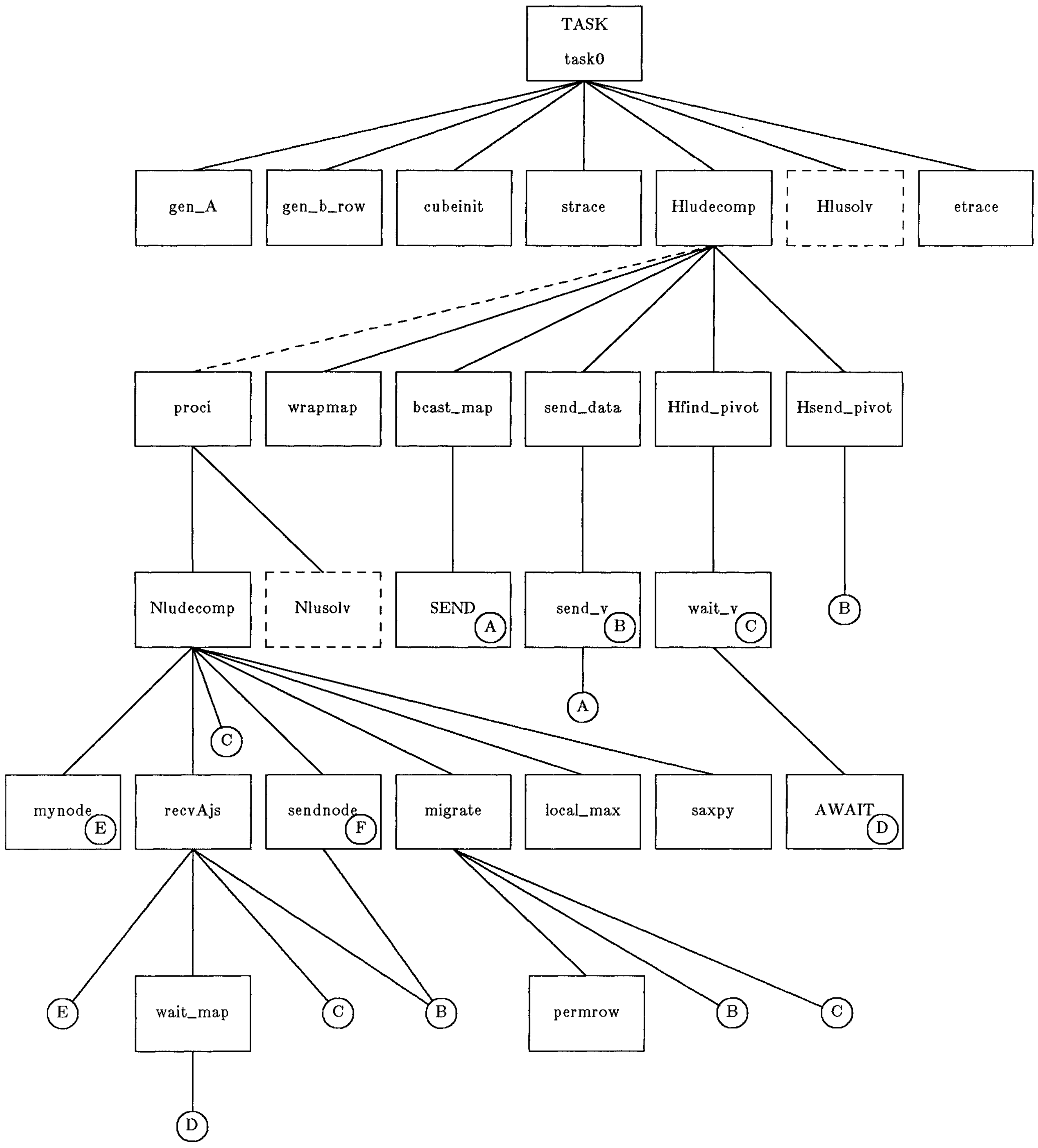
[5] J. A. GEORGE, M. T. HEATH, J. W-H. LIU, AND E. G-Y. NG, "Sparse Cholesky factorization on a local-memory multiprocessor", Research Report CS-86-02, Department of Computer Science, University of Waterloo (1986).

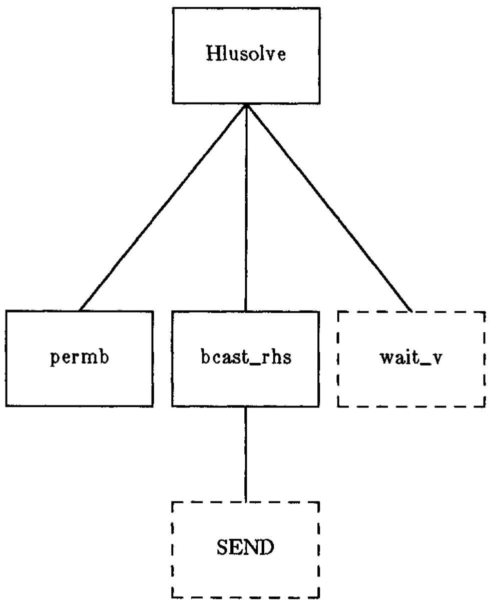
[6] M. T. HEATH, "Parallel Cholesky factorization in message passing multiprocessor environments", Technical Report ORNL-6150, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831 (1985).

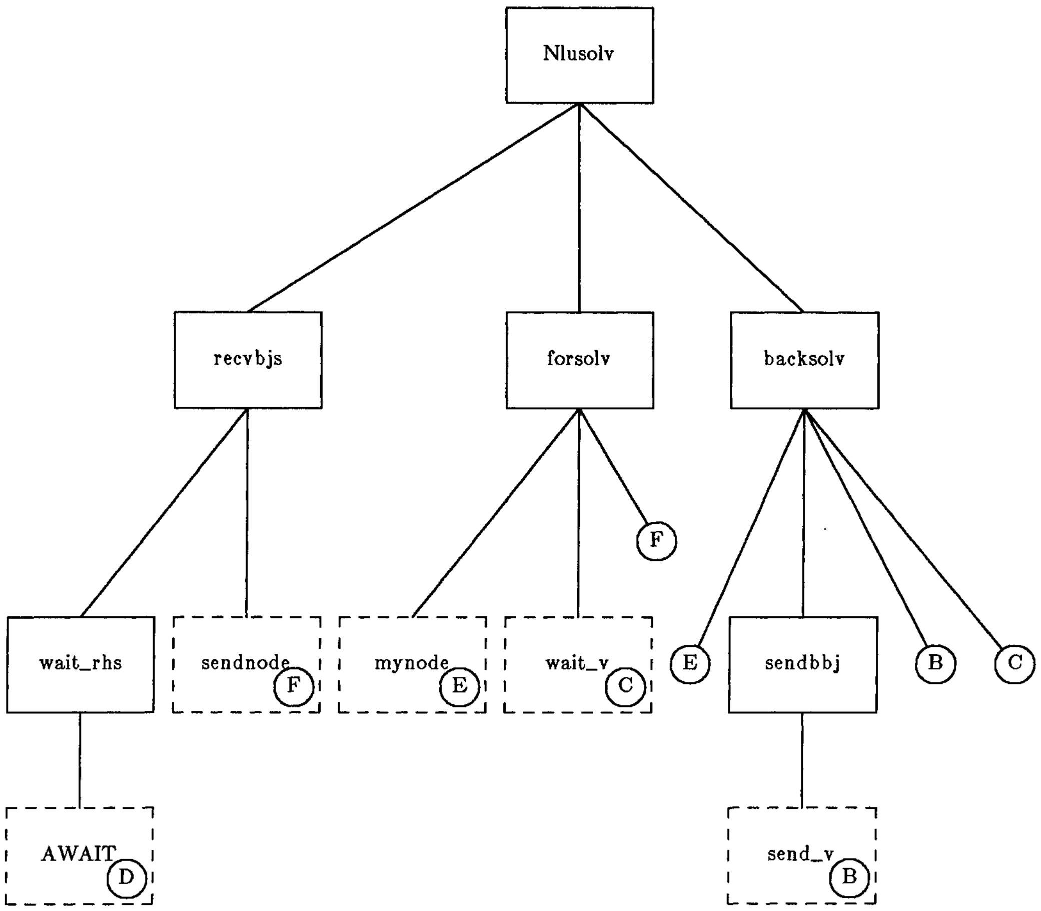
[7] C. LAWSON, R. HANSON, D. KINCAID, AND F. KROGH, "Basic linear algebra subprograms for Fortran usage", *ACM Trans. Math Software* **5**, pp. 308-371 (1979).

7. Appendix A: Control Diagrams

Control Relations of Procedures for Parallel Gaussian Elimination with Partial Pivoting and Triangular Solutions




Hlusolve
permb
bcast_rhs
wait_v
SEND


Nlusolv
recvbjs
forsolv
backsolv
F
wait_rhs
sendnode
F
mynode
E
wait_v
C
E
sendbbj
B
C
AWAIT
D
send_v
B

8. Appendix B: Program Listing

```
/*    C Programs Implementing Parallel Gaussian Elimination          */
/*    with Partial Pivoting, and Forward/Backward Substitution       */
/*    on a Local-Memory Multiprocessor Simulator                     */

#include </u/sparspak/simulator/local/intel.h>
#include <stdio.h>
#include <math.h>

#define STACK 10000
#define REAL float                  /* float: 4 bytes on vax 11/780 */
#define LONG long                   /* long:  4 bytes on vax 11/780 */
#define SHORT short                 /* short: 2 bytes on vax 11/780 */
#define msg_MAP        1
#define msg_MTXA       2
#define msg_lmax       3
#define msg_pivot      4
#define msg_kstg       5
#define msg_kpivot     6
#define msg_rowk       7
#define msg_rhs        8
#define msg_fbj        9
#define msg_bbj       10
#define msg_soln      11

TASK proci();

char *calloc();
double sdot();

int NDIM, NPROC;
```

```
/*
---------------------------------------------------------------------
*/

TASK
task0()
/* host process: the driver program */
{
    extern int NDIM, NPROC;

    SHORT *map, *perm;
    LONG  *jbegin;
    REAL  *Ajs, *b;
    int    i, mknt;

    scanf("%d %d", &NDIM, &NPROC);
    Ajs = (REAL *)calloc(NDIM*NDIM+1, sizeof(REAL));
    b = (REAL *)calloc(NDIM+1, sizeof(REAL));
    perm = (SHORT *)calloc(NDIM+1, sizeof(SHORT));
    map = (SHORT *)calloc(NDIM+1, sizeof(SHORT));
    jbegin = (LONG *)calloc(NDIM+1, sizeof(LONG));

    for (i = 1; i <= NDIM; i++)
        jbegin[i] = 1 + (i-1)*NDIM;

    gen_A(Ajs, NDIM, NDIM);

    gen_b_row(Ajs, NDIM, b);

    cube_init(1.0);

    strace("tfile");

    Hludecomp(NPROC, NDIM, map, perm, jbegin, Ajs);

    Hlusolv(NDIM, b, perm);

    etrace();

    mknt = 0;
    for (i = 1; i <= NDIM-1; i++)
        if ( map[i] != map[perm[i]] ) mknt++;
    printf("migration occurs %d times \n", mknt);

    printf("solution is \n");
    for (i = 1; i <= NDIM; printf("%f\n", b[i++]) );

    free(Ajs);
    free(b);
    free(perm);
    free(map);
    free(jbegin);

}
```

```
/*
-----------------------------------------------------------------
*/

Hludecomp(nproc, ndim, map, perm, jbegin, Ajs)
int nproc, ndim;
SHORT map[], perm[];
LONG jbegin[];
REAL Ajs[];
{
    int nrows, p, i, j, kstage, kpivot, lth;

    for (p = 0; p < nproc; p++)
        tfork(proci, STACK);

    nrows = ndim;

    wrapmap(nrows, map, nproc);                    /* compose the map */

    lth = nrows + 1;
    bcast_map(msg_MAP, nproc, lth, map);         /* broadcast the map */

    lth = ndim;
    send_data(msg_MTXA, map, nrows, lth, Ajs);     /* distibute data */

    lth = ndim;
    for (kstage = 1; kstage <= ndim-1; kstage++) /* determine pivot */
    {
        Hfind_pivot(msg_lmax, nproc, ndim, map, kstage, &kpivot);
        Hsend_pivot(msg_pivot, nproc, ndim, map, kstage, kpivot);
        perm[kstage] = kpivot;
    }
}

/*
---------------------------------------------------------------------
*/

Hlusolv(ndim, b, perm)
int ndim;
SHORT perm[];
REAL b[];
{
     extern int NPROC;
     int i, lth, bk;
     REAL temp;

     permb(ndim, b, perm);          /* Host permutes right hand side */

     lth = ndim + 1;
     bcast_rhs(msg_rhs, NPROC, lth, b);/* broadcast right hand side */

     lth = 1;
     for (i = 1; i <= ndim; i++)          /* receive solution element */
     {
         wait_v(msg_soln, &bk, &lth, &temp);
         b[bk] = temp;
     }
}
```

```
/*
-----------------------------------------------------------------------
*/

Hfind_pivot(typ_lmax, nproc, ndim,  map, kstage, pkpivot)
int  typ_lmax, nproc, ndim, kstage, *pkpivot;
SHORT map[];
{

    int dest, k, rowno, *marker, lth;
    REAL pivot, temp;

    marker = (int *)calloc(nproc, sizeof(int) );

    for (k = 0; k < nproc; k++)
        marker[k] = 0;

    pivot = 0;
    for (k = kstage; k <= ndim; k++)          /* determine pivot row */
    {
        dest = map[k];
        if ( marker[dest] == 0 )
        {
            lth = 1;
            wait_v(typ_lmax, &rowno, &lth, &temp); /* receive pivot  */
            if (temp > pivot)                      /* candidates     */
            {
                pivot = temp;
                *pkpivot = rowno;
            }
            marker[dest] = 1;
        }
    }
    free(marker);
}

/*
-----------------------------------------------------------------------
*/

Hsend_pivot(typ_pivot, nproc, ndim, map, kstage, kpivot)
int typ_pivot, nproc, ndim, kstage, kpivot;
SHORT map[];
{

    int dest, k, *marker;
    REAL temp;

    marker = (int *)calloc(nproc, sizeof(int) );

    for (k = 0; k < nproc; k++)
        marker[k] = 0;

    for (k = kstage; k <= ndim; k++)  /* inform active nodes of */
    {                                 /* pivot row              */
        dest = map[k];
        if ( marker[dest] == 0 )
          send_v(typ_pivot, dest, kpivot, NULL, &temp);
        marker[dest] = 1;
    }
    free(marker);
}
```

```
/*
----------------------------------------------------------------------
*/

Nludecomp(Ajs, map, jbegin, work)
SHORT map[];
LONG  jbegin[];
REAL  Ajs[], work[];
{
  extern int NDIM, NPROC;

  int i, j, jstrt, k, kpivot, kstg, lth, me, nrows, nxtrow, rowno;
  REAL amax, pivot, temp;

  me = mynode();

  recvAjs(Ajs, &nrows, map, jbegin);         /* receive map and data */

  kstg = 1;                                  /* Numerical factorization */
  while ( nrows > 0 )
  {
    lth = NULL;
    wait_v(msg_pivot, &kpivot, &lth, &temp);
    if ( map[kpivot] == me )
    {
      jstrt = jbegin[kpivot] + kstg - 1;
      sendnode(msg_rowk, map, kstg, kstg, NDIM-kstg+1, &Ajs[jstrt]);
    }
    migrate(Ajs, jbegin, map,  kstg, kpivot, msg_kstg, msg_kpivot,me);
    if ( map[kstg] == me ) nrows--;
    if ( nrows > 0 )                         /* rows remain to be modified */
    {
      lth = NDIM;
      wait_v(msg_rowk, &k, &lth, &work[1]);

      local_max(Ajs, jbegin, map, kstg, &work[1], &rowno, &amax, me);
      if ( kstg <= NDIM-2 )
      {
        send_v(msg_lmax, HOST,  rowno, 1, &amax);
        nxtrow = kstg+1;
        for (j = nxtrow; j <= NDIM; j++)
        {
          if ( map[j] == me )   /* modify row j */
          {
            jstrt = jbegin[j];
            pivot = Ajs[jstrt+kstg-1];
            saxpy(&Ajs[jstrt+kstg+1], &work[3], -pivot, NDIM-kstg-1);
          }
        }
      }
      kstg++;
      if ( kstg == NDIM & map[NDIM] == me )
        nrows--;
    }
  }
}
```

```
/*
------------------------------------------------------------------
*/

Nlusolv(Ajs, b, map, jbegin)
SHORT map[];
LONG jbegin[];
REAL Ajs[], b[];
{
      int nbjs;

      recvbjs(b, &nbjs, map);             /* receive right hand side */

      forsolv(Ajs, jbegin, b, nbjs, map);   /* forward substitution */

      backsolv(Ajs, jbegin, b, nbjs, map); /* backward substitution */

}
```

```
/*
--------------------------------------------------------------------------
*/

backsolv(Ajs, jbegin, b, nbjs, map)    /* back substitution in a node */
int  nbjs;
SHORT map[];
LONG jbegin[];
REAL Ajs[], b[];
{
  extern int NDIM;

  int  i, j, kstg, lth, me, nextb;
  REAL temp;

  me = mynode();

  kstg = NDIM;

  if ( map[1] == me ) nbjs++;

  if ( map[NDIM] == me )       /* start other nodes as soon as possible *
  {
    b[NDIM] /= Ajs[jbegin[NDIM]+NDIM-1]; /* compute the last solution *
    sendbbj( msg_bbj, map, NDIM, NDIM, 1, &b[NDIM]);  /* inform nodes *
    send_v(msg_soln, HOST, NDIM, 1, &b[NDIM]);           /* inform host *
    nbjs--;
  }

  while ( nbjs > 0 )
  {
    lth = 1;
    wait_v(msg_bbj, &j, &lth, &temp);       /* receive solution element *
    for ( i = 1; i <= j-1; i++)
    {
      if ( map[i] == me )                              /* modify b[i] *
        b[i] -= (temp*Ajs[jbegin[i]+j-1]);
    }
    kstg--;
    if ( map[kstg] == me )
    {
      b[kstg] /= Ajs[jbegin[kstg]+kstg-1];          /* compute solution
      if ( kstg != 1 )
        sendbbj(msg_bbj, map, kstg, kstg, 1, &b[kstg]);/* inform nodes
      send_v(msg_soln, HOST, kstg, 1, &b[kstg]);          /* inform host
      nbjs--;
    }
  }
}
```

```
/*
-------------------------------------------------------------------------------
*/

bcast_map(type, nproc, lth, sx)        /* broadcast map to nodes */
int type, nproc, lth;
SHORT sx[];
{
    int i, l;
    REAL *p;

    l = lth*sizeof(SHORT);
    for (i = 0; i < nproc ; i++ )
        SEND(i, type, l, sx, NULL, p);
}

/*
-------------------------------------------------------------------------------
*/

bcast_rhs(type, nproc, lth,  x)  /* broadcast right hand side */
int type, nproc, lth;
REAL x[];
{
    int i, l;
    REAL *p;

    l = lth*sizeof(REAL);
    for (i = 0; i < nproc ; i++ )
        SEND(i, type, l, x, NULL, p);
}
```

```
/*
---------------------------------------------------------------------
*/

forsolv(Ajs, jbegin, b, nbjs, map) /* forward substitution in a node */
int  nbjs;
SHORT map[];
LONG  jbegin[];
REAL Ajs[], b[];
{
    extern int NDIM;

    int  i, j, kstg, lth, me, nextb;
    REAL temp;

    me = mynode();

    kstg = 1;
    while ( nbjs > 0 )      /* elements in rhs remain to be modified */
    {
        lth = 1;
        wait_v(msg_fbj, &j, &lth, &temp); /* receive solution element */
        nextb = j + 1;
        for ( i = nextb; i <= NDIM; i++)
        {
            if ( map[i] == me )                         /* modify b[i] */
                b[i] -= (temp*Ajs[jbegin[i]+j-1]);
        }
        kstg++;
        if ( map[kstg] == me )
        {
            if ( kstg != NDIM )      /* inform other nodes of solution */
                sendnode(msg_fbj, map, kstg, kstg, 1, &b[kstg]);
            nbjs--;
        }
    }
}

/*
---------------------------------------------------------------------
*/

gen_A(x, ncols, mrows)    /* generate random matrices */
int mrows, ncols;
REAL x[];
{
    int i;

    for (i = 1; i <= mrows*ncols; i++)
        x[i] = (REAL) random()/1028318;
}
```

```
/*
----------------------------------------------------------------------
*/

/*
    Given a square matrix A of dimension ndim stored in one dimensional
    array x row by row, gen_b_row() generates right hand side
    b so that the system Ax = b has a solution of all one's.
*/

gen_b_row(x, ndim, b)
int ndim;
REAL x[], b[];
{
    int i, j, irow;

    for (i = 1; i <= ndim; i++)
    {
        irow = (i-1)*ndim;
        b[i] = 0;
        for (j = 1; j <= ndim; j++)
            b[i] += x[irow+j];
    }
}
```

```
/*
--------------------------------------------------------------------------------
*/

/*
    local_max() scales the pivot column to obtain multipliers and
    modifies the first element of remaining rows, send the local
    max to HOST in order to determine the pivot row for next stage
*/

local_max(Ajs, jbegin, map, kstg, rowk, prowno, pamax, me)
int kstg, *prowno, me;
SHORT map[];
LONG jbegin[];
REAL Ajs[], *pamax, rowk[];
{
    extern int NDIM;
    int i, nxtrow, jstrt;
    REAL temp, pivot, s;

    pivot = rowk[0];
    s = rowk[1];
    nxtrow = kstg + 1;
    *pamax = 0.;
    for (i = nxtrow; i <= NDIM; i++)
    {
         if ( map[i] == me )
         {
             jstrt = jbegin[i] + kstg - 1; /* element on pivot column */
             Ajs[jstrt] /= pivot;        /* overwrite A with multiplier */
             Ajs[jstrt+1] -= Ajs[jstrt]*s;
             temp = (REAL) fabs( (double) Ajs[jstrt+1]  );
             if ( temp > *pamax )            /* find local maximum which */
             {                              /* is candidate for nxt pivot */
                 *pamax = temp;
                 *prowno = i;
             }
         }
    }
}
```

```
/*
--------------------------------------------------------------------------
*/

migrate(Ajs, jbegin, map,  kstg, kpivot, typ_kstg, typ_kpivot, me)
int kstg, kpivot, typ_kstg, typ_kpivot;
SHORT map[];
LONG jbegin[];
REAL Ajs[];
{
    extern int NDIM;
    int k, lth;

    if ( kstg != kpivot )  /* row interchange occurs */
    {
        if ( map[kstg] == me & map[kpivot] == me )
            permrow(&Ajs[jbegin[kstg]], &Ajs[jbegin[kpivot]], NDIM);
        else if ( map[kstg] == me )
        {
          lth = NDIM;
          send_v(typ_kpivot, map[kpivot], kpivot, lth,&Ajs[jbegin[kstg]])
          wait_v(typ_kstg, &k, &lth, &Ajs[jbegin[kstg]]);
        }
        else if ( map[kpivot] == me )
        {
          lth = NDIM;
          send_v(typ_kstg, map[kstg], kstg, lth,&Ajs[jbegin[kpivot]]);
          wait_v(typ_kpivot, &k, &lth, &Ajs[jbegin[kpivot]]);
        }
    }
}

/*
--------------------------------------------------------------------------
*/

permb(ndim, b, perm)  /* permute right hand side according to perm */
int ndim;
SHORT perm[];
REAL b[];
{
    int i, k;
    REAL pivot;

    for (k = 1; k < ndim; k++)
    {
        i = perm[k];
        if (i != k )
        {
            pivot = b[i];
            b[i] = b[k];
            b[k] = pivot;
        }
    }
}
```

```
/*
---------------------------------------------------------------------------
*/

permrow(rowk, rowi, lth) /* swap row i and row k on the same node */
int lth;
REAL rowk[], rowi[];
{
    int i;
    REAL *pk, *pi, temp;

    pk = rowk;
    pi = rowi;
    for (i = 1; i <= lth; i++)
    {
        temp = *pi;
        *pi++ = *pk;
        *pk++ = temp;
     }
}

/*
---------------------------------------------------------------------------
*/

TASK
proci()          /* driver of a node process */
{
    extern int NDIM, NPROC;

    int    nrows;
    SHORT  *map;
    LONG   *jbegin;
    REAL   *Ajs, *work, *b;

    nrows = NDIM/NPROC + 1;
    map = (SHORT *)calloc(NDIM+1, sizeof(SHORT));
    jbegin = (LONG *)calloc(NDIM+1, sizeof(LONG));
    Ajs = (REAL *)calloc(NDIM*nrows+1, sizeof(REAL));
    work = (REAL *)calloc(NDIM+1, sizeof(REAL));
    b = (REAL *)calloc(NDIM+1, sizeof(REAL));

    Nludecomp(Ajs, map, jbegin, work);

    Nlusolv(Ajs, b, map, jbegin);

    free(b);
    free(work);
    free(Ajs);
    free(jbegin);
    free(map);

}
```

```
/*
-----------------------------------------------------------------------------------
*/

recvAjs(Ajs, nrows, map, jbegin)  /* receive map and data from host */
int  *nrows;
SHORT map[];
LONG jbegin[];
REAL Ajs[];
{
     extern int NDIM;

     int  i, j, k, lth, me, nr, rowno;
     REAL *pAj,  amax, temp;

     me = mynode();

     lth = NDIM + 1;
     wait_map(msg_MAP, &lth, map);

     for (j = 0; j <= NDIM; j++)                    /* initialize jbegin */
         jbegin[j] = 0;

     nr = 0;
     for (j = 1; j <= NDIM; j++)
         if (map[j] == me) nr++;

     *nrows = nr;                        /* number of rows I am to receive */

     amax = 0.0;
     pAj = &Ajs[1];
     for (k = 1; k <= nr; k++)                        /* receive nr rows */
     {
         lth = NDIM;
         wait_v(msg_MTXA, &j, &lth, pAj);
         temp = (REAL) fabs( (double) *pAj );
         if ( temp > amax ) /* determine local maximum in first column */
         {
             amax = temp;
             rowno = j;
         }
         jbegin[j] = 1 + (k-1)*lth;
         pAj += lth;
     }
     send_v(msg_lmax, HOST, rowno, 1, &amax); /* send pivot candidate */
                                              /* to host             */
}
```

```
/*
--------------------------------------------------------------------
*/

recvbjs(b, pnbjs, map) /* receive permuted right hand side from host */
int *pnbjs;
SHORT map[];
REAL b[];
{
     extern int NDIM;

     int  j, k, me, nb, lth;
     REAL  *pbj;

     lth = NDIM+1;
     wait_rhs(msg_rhs, &lth, b);

     me = mynode();

     nb = 0;
     for (j = 1; j <= NDIM; j++)
         if (map[j] == me) nb++;

     *pnbjs = nb;  /* number of bj's I should own */

     if ( map[1] == me ) /* broadcast first solution element to nodes */
     {
         (*pnbjs)--;
         sendnode(msg_fbj, map, 1, 1, 1, &b[1]);
     }
}

/*
--------------------------------------------------------------------
*/

saxpy(x, y, s, lth)
int lth;
REAL x[], y[], s;
{
    REAL *px, *py;

    px = x;
    py = y;
    for ( ; lth > 0; lth--, px++, py++)
        *px += s*(*py);
}
```

```
/*
---------------------------------------------------------------------------
*/

send_data(type, map, num_vecs, lth, x)
int type, num_vecs, lth;
SHORT map[];
REAL x[];
{
    int j, proc;
    REAL *px;

    px = &x[1];     /* broadcat data to nodes according to map */
    for (j = 1; j <= num_vecs; j++)
    {
        proc = map[j];
        send_v(type, proc, j, lth, px);
        px += lth;
    }
}

/*
---------------------------------------------------------------------------
*/

sendnode(msg_type, map, proc, colno, lth, vj)
int msg_type, proc, colno, lth;
SHORT map[];
REAL vj[];
{
    extern int NDIM, NPROC;

    int  dest, i, k, nxtcol, *marker;

    marker = (int *)calloc(NPROC, sizeof(int));

    for (k = 0; k < NPROC; k++)
        marker[k] = 0;

    nxtcol = proc + 1;
    for ( k = nxtcol; k <= NDIM; k++)
    {
        dest = map[k];
        if ( marker[dest] == 0 )
        {
            send_v(msg_type, dest, colno, lth, vj);
            marker[dest] = 1;
        }
    }

    free(marker);
}
```

```
/*
---------------------------------------------------------------------
*/

send_v(type, dest, colno, vlth, x)
int type, dest, colno, vlth;
REAL x[];
{
    int lth;

    lth = vlth*sizeof(REAL);
    SEND(dest, type, sizeof(int), &colno, lth, x);
}

/*
---------------------------------------------------------------------
*/

sendbbj(msg_type, map, proc, kstg, lth, vj)
int msg_type, proc, kstg, lth;
SHORT map[];
REAL vj[];
{
    extern int NPROC;

    int  dest, i, k, *marker;

    marker = (int *)calloc(NPROC, sizeof(int));

    for (k = 0; k < NPROC; k++)
        marker[k] = 0;

    for ( k = 1; k <= kstg-1; k++)
    {
        dest = map[k];
        if ( marker[dest] == 0 )
        {
            send_v(msg_type, dest, kstg, lth, vj);
            marker[dest] = 1;
        }
    }

    free(marker);
}

/*
---------------------------------------------------------------------
*/

wait_map(type, lth, sx)
int type, *lth;
SHORT sx[];
{
    int bufsize, l;
    REAL *p;

    l = NULL;
    bufsize = (*lth)*sizeof(SHORT);
    AWAIT(type, &bufsize, sx, &l, p);
    *lth = bufsize/sizeof(SHORT);
}
```

```
/*
--------------------------------------------------------------------
*/

wait_rhs(type, lth, x)
int type, *lth;
REAL x[];
{
    int bufsize, l;
    REAL *p;

    l = NULL;
    bufsize = (*lth)*sizeof(REAL);
    AWAIT(type, &bufsize, x, &l, p);
    *lth = bufsize/sizeof(REAL);
}

/*
--------------------------------------------------------------------
*/

wait_v(type, colno, x_lth, x)
int type, *colno, *x_lth;
REAL x[];
{
    int ilth, lth;

    lth = (*x_lth)*sizeof(REAL);
    ilth = sizeof(int);
    AWAIT(type, &ilth, colno, &lth, x);
    *x_lth = lth/sizeof(REAL);
}

/*
--------------------------------------------------------------------
*/

/*
    wrapmap(ncols_or_nrows, map, nrpoc) sets up the column or row
    to processor map function in a wrap around manner.
*/

wrapmap(ncols_or_nrows, map, nproc)
int ncols_or_nrows, nproc;
SHORT map[];
{
    int i;

    for (i = 1; i <= ncols_or_nrows; i++)
        map[i] = (i-1) % nproc;
}
```