Experiments with Multicomputer LU-Decomposition

E. F. Van de Velde

CRPC-TR89045
April, 1989

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Experiments with Multicomputer LU-Decomposition

Eric F. Van de Velde*
Applied Mathematics 217-50
Caltech
Pasadena, CA 91125

April 26, 1989
CRPC-89-1

Abstract

We present a concurrent LU-decomposition algorithm based on implicit pivoting of both rows and columns. Most pivoting strategies are thus easily incorporated. This algorithm is implemented such that it is, to a large extent, independent of the data distribution.

With this program, we study the performance of concurrent LU-decomposition as a function of data distribution and pivoting strategy. We show that LU-decomposition with some pivoting strategies is both faster and numerically more stable than LU-decomposition without pivoting. Experimental evidence on the Ametek 2010 shows that, for performance considerations, pivoting is equivalent to randomizing the data distribution.

1 Introduction

Multicomputer implementations of the LU-decomposition algorithm, can be categorized by the distribution of the coefficient matrix over the concurrent processes and by the pivoting strategy. Chamberlain [1], Chu and George [3], Geist and Heath [5], and Moler [9] have examined varying combinations

*This research is supported in part by Department of Energy Grants Nos. DE-FG03-85ER25009 and DE-AS03-76ER72012. This material is based upon work supported by the NSF under Cooperative Agreement No. CCR-8809615. The government has certain rights in this material.

of row- and column-oriented distributions with row and column pivoting. Fox et al. [4], Hipes and Kupperman [7] distribute the matrix over a two dimensional grid of processes, based on the observation that this minimizes the communication cost.

The particular distribution has far reaching consequences for the user. E.g., the initialization of the matrix may be an expensive and complicated computation incompatible with the distribution imposed by the LU-decomposition, typically a library routine. The matrix could be initialized in one distribution and decomposed in another, but a possibly expensive data redistribution is then necessary between the initialization and LU-decomposition step.

For this reason, we prefer to regard the matrix distribution as a given. We develop our library algorithms such that they are correct for a large class of distributions rather than for just one distribution. When the library subroutine is called, the distribution is as much part of the data as is the matrix itself. The redistribution step is thus avoided. Of course, some distributions are more efficient than others. We quantify this effect through experimentation.

Data distribution is only one factor influencing performance. Our experience shows that — for scientific computing — it is the most important factor because the work is generally some increasing function of the amount of data. Optimizing a concurrent program is often equivalent to optimizing the data distribution. Changing the latter often requires a major adaptation of the whole program. Hence, concurrent program optimization is necessarily a global operation and cannot be achieved through the optimization of individual components. Our approach to this problem is to write programs that are valid for many data distributions. In current programming environments, this is rather difficult and cumbersome for arbitrary programs. It is a tractable proposition, however, for many important components of scientific computing. In [16], we derive many such algorithms. Our concurrent LU-decomposition is one example of how this is achieved with currently available tools.

Pivoting is traditionally used to accomplish two goals: numerical stability and, for sparse matrix computations, limiting the fill. Our experimental results in Sections 9 and 10 show that pivoting is also an effective randomizer of the computation and acts as a load balancer. In some cases concurrent LU-decomposition is faster with than without pivoting. With pivoting, the performance of LU-decomposition is less sensitive to the particular matrix distribution. In some cases, pivoting may correct load imbalance problems to the extent that efficiency of the LU-decomposition would not be a consideration in the choice of distribution. If pivoting is to be a viable load balancing technique, the choice of pivot must have a maximum flexibility. We use an implicit pivoting technique that can incorporate a wide range of exisiting pivoting strategies. We shall also introduce two new strategies.

All of our experiments were performed on both the Ametek 2010 and the Intel iPSC-2. The obtained results were essentially equivalent as far as the performance of our LU-decomposition program is concerned. There were, of course, performance differences between the two machines. For readability and brevity, we report here on our Ametek 2010 results only.

The main parts of this paper are contained in Sections 2, 3, 8, 9, and 10. Sections 2 and 3 describe the LU-decomposition and pivoting algorithms. In Section 8, we give a performance analysis. In Sections 9 and 10, we report on our experiments. Sections 4 and 5 discuss implementation details that influence some execution times. In Sections 6 and 7 we briefly describe the operating system and the architecture of the Ametek 2010.

2 The LU-Decomposition Algorithm

The LU-decomposition algorithm with implicit row and column pivoting is based on a reformulation of the classical result:

Theorem 1 *For any real $M \times N$ matrix A, there exists an $M \times M$ permutation matrix R and an $N \times N$ permutation matrix C such that:*

$$RAC^T = \hat{L}\hat{U},$$

where $\hat{L}$ is $M \times M$ unit lower triangular and $\hat{U}$ is $M \times N$ upper triangular.

For a proof, we refer to basic numerical analysis texts. In [15], we prove the following related theorem:

Theorem 2 *For any real $M \times N$ matrix A, there exists an $M \times M$ permutation matrix R and an $N \times N$ permutation matrix C such that:*

$$A = LC^T I_{N,M} RU \tag{1}$$

where RLC^T is unit lower triangular and RUC^T is upper triangular. Both L and R are $M \times N$ matrices. The matrix $I_{N,M}$ is the $N \times M$ identity matrix.

```
$\mathcal{M} := \{m : 0 \le m < M\}$ ;
$\mathcal{N} := \{n : 0 \le n < N\}$ ;
for $k = 0, 1, \ldots, \min(M, N) - 1$ do begin
    {Pivot Strategy and Bookkeeping.}
    do pivot search and find $a_{rc}, r[k], c[k]$ ;
    $\mathcal{M} := \mathcal{M} \setminus \{r[k]\}$ ;
    $\mathcal{N} := \mathcal{N} \setminus \{c[k]\}$ ;
    if $a_{rc} = 0.0$ then terminate ;

    {Calculation of the Multiplier Column.}
    for all $m \in \mathcal{M}$ do
        $a[m, c[k]] := a[m, c[k]]/a_{rc}$ ;

    {Elimination.}
    for all $(m, n) \in \mathcal{M} \times \mathcal{N}$ do
        $a[m, n] := a[m, n] - a[m, c[k]]a[r[k], n]$
end
```

Figure 1: LU-Decomposition with Implicit Pivoting.

The matrix formed by the first M columns of RLC^T and the matrix RUC^T satisfy Theorem 1 in the role of $\hat{L}$ and $\hat{U}$ respectively. Even though L and U have complicated structure, linear systems of the form $Lx = b$ or $Ux = b$ are solved easily by a modified backsolve algorithm (see [14]). For all practical purposes, the matrices L and U are as acceptable as the matrices $\hat{L}$ and $\hat{U}$.

The matrices L and U of Theorem 2 are obtained from A by an LU-decomposition with implicit complete pivoting. A full search of all feasible matrix entries is rarely performed and a partial search is preferred. Provided the LU-decomposition runs to completion, the matrix A is still factored in the form of Equation 1. For any partial pivoting strategy there exist matrices for which the LU-decomposition does not complete because a zero pivot is found prematurely. For nearby matrices, this partial pivoting strategy is numerically unstable. Though not proven, it is accepted that the extent of the pivot search determines the class of matrices for which the LU-decomposition is numerically stable.

In Figure 1, we use an informal notation to display the LU-decomposition algorithm with implicit pivoting. In this program, the array a represents the matrix. The integer arrays r and c are the pivot indices, i.e., the pivot of the k-th decomposition step is given by $a[r[k], c[k]]$. The variable a_{rc} is the current pivot value. The index sets $\mathcal{M}$ and $\mathcal{N}$ are the sets of feasible rows and columns. Initially, all rows and all columns are feasible. After a pivot is chosen from the feasible entries, the assignments $\mathcal{M} := \mathcal{M} \setminus \{r[k]\}$ and $\mathcal{N} := \mathcal{N} \setminus \{c[k]\}$ make its row and column infeasible. If the pivot value is zero, the LU-decomposition is terminated. The multipliers are computed in the feasible rows of the pivot column. The elimination takes place over all feasible matrix entries.

On multicomputers, concurrency is implemented by communicating sequential processes. For the purposes of this algorithm, the number of processes is fixed. Each process has a unique identifier, which is used as an address in the exchange of messages. It is often convenient to map the system supplied identifier into a user defined identification. E.g., for vector calculations the processes are organized as a one dimensional process grid. The user identification for each process is then a number p between 0 and $P-1$, where P is the number of processes. A multicomputer program operating on a vector, say of dimension M, must distribute the vector entries over the P processes. A distribution allocates each vector entry to a particular process, e.g., it maps the m-th entry to process p. The collection of entries allocated to one process form a local vector. Each entry of this local vector corresponds to exactly one entry of the global vector, e.g., the i-th local entry is the m-th global entry. A distribution is thus a map from the global index m, where $0 \leq m < M$, to an index pair (p, i) consisting of a process number and a local index. Two often used maps are the *linear distribution* given by:

$$\lambda(m) = (p := \max(\left\lfloor \frac{m}{L+1} \right\rfloor, \left\lfloor \frac{m-R}{L} \right\rfloor)),\ \ m - pL - \min(p, R) \qquad (2)$$

and the *scatter distribution* given by:

$$\sigma(m) = m \bmod P, \left\lfloor \frac{m}{P} \right\rfloor \qquad (3)$$

where $L = \left\lfloor \frac{M}{P} \right\rfloor$ and $R = M \bmod P$. An example is displayed in Table 1: the index range $0 \leq m < 10$ is distributed over 4 processes.

For matrix calculations, we organize the processes in a rectangular grid such that each process is identified by a two dimensional coordinate (p, q),

	Linear		Scatter	
Global	Process	Local	Process	Local
0	0	0	0	0
1	0	1	1	0
2	0	2	2	0
3	1	0	3	0
4	1	1	0	1
5	1	2	1	1
6	2	0	2	1
7	2	1	3	1
8	3	0	0	2
9	3	1	1	2

Table 1: Four Fold Linear and Scatter Distribution.

where $0 \leq p < P$ and $0 \leq q < Q$. The total number of processes is thus $P \times Q$. We define the *p-th process row* and the *q-th process column* as the collection of processes given by $\{(p,q) : 0 \leq q < Q\}$ and $\{(p,q) : 0 \leq p < P\}$ respectively. The concurrent LU-decomposition is derived from the sequential one by imposing a distribution of the matrix over the $P \times Q$ processes. We define a matrix distribution as the Cartesian product of two vector distributions μ and ν. The rows of the matrix are distributed by μ over the P process rows. Similarly, the columns are distributed by ν over the Q process columns. Thus, if $\mu(m) = (p,i)$ and $\nu(n) = (q,j)$, the global matrix entry with row and column indices m and n is found in process (p,q) as local matrix entry $a_{i,j}$.

The distributions μ and ν are generally user supplied. However, even using the linear and the scatter distributions only, a matrix can be distributed in a variety of ways. Consider, e.g., the distribution of a matrix over four processes. We may organize the processes in a 4×1, a 2×2, or a 1×4 process grid. We may also apply either linear or scatter distribution to rows and columns. This results in the 8 distributions:

4×1 linear-,	2×2 linear-linear,
4×1 scatter-,	2×2 scatter-linear,
1×4 -linear,	2×2 linear-scatter,
1×4 -scatter,	2×2 scatter-scatter,

where we list the process grid, the row, and the column distribution. (Note that the linear and the scatter distribution are the same when $P = 1$ or $Q = 1$.) A 5×7 matrix $A = [a_{m,n}]$ distributed over a 2×2 process grid with a linear row and a scatter column distribution is stored as given by:

$$\begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix} := \begin{bmatrix} \begin{bmatrix} a_{0,0} & a_{0,2} & a_{0,4} & a_{0,6} \\ a_{1,0} & a_{1,2} & a_{1,4} & a_{1,6} \\ a_{2,0} & a_{2,2} & a_{2,4} & a_{2,6} \end{bmatrix} & \begin{bmatrix} a_{0,1} & a_{0,3} & a_{0,5} \\ a_{1,1} & a_{1,3} & a_{1,5} \\ a_{2,1} & a_{2,3} & a_{2,5} \end{bmatrix} \\ \begin{bmatrix} a_{3,0} & a_{3,2} & a_{3,4} & a_{3,6} \\ a_{4,0} & a_{4,2} & a_{4,4} & a_{4,6} \end{bmatrix} & \begin{bmatrix} a_{3,1} & a_{3,3} & a_{3,5} \\ a_{4,1} & a_{4,3} & a_{4,5} \end{bmatrix} \end{bmatrix}$$

where $A_{p,q}$ is the submatrix of A stored in process (p, q). The local indices (i, j) corresponding to the global indices (m, n) are determined by the position of the entry $a_{m,n}$ in its submatrix.

The program of Figure 1 is parallelized such that it is valid for any distribution of the matrix represented with functions like μ and ν. In Figure 2, we display the program for process (p, q) of the concurrent LU-decomposition. This program is formally derived from the sequential one in [15]. The transition from the sequential to the concurrent program can also be understood intuitively. The concurrent program no longer has global feasible sets $\mathcal{M}$ and $\mathcal{N}$. Instead, the index sets $\mathcal{I}$ and $\mathcal{J}$ keep track of the local feasible rows and columns. Initially, all rows local to the process row and all columns local to the process column are feasible. Both the sequential and concurrent version have $\min(M, N)$ sequential steps. In both, the pivot search returns the pivot value and its global indices. The concurrent program immediately converts the latter into local indices with the distributions μ and ν. This determines that the pivot is located in process $(\hat{p}, \hat{q})$ and that its local indices there are $[\hat{\imath}, \hat{\jmath}]$. The pivot row is made infeasible in process row $\hat{p}$ by deleting local row $\hat{\imath}$ from the feasible row set $\mathcal{I}$. Similarly, the pivot column is made infeasible in process column $\hat{q}$ by deleting local column $\hat{\jmath}$ from the feasible column set $\mathcal{J}$. The decomposition is terminated if the pivot value, known in all processes, is equal to zero. The multipliers are calculated in the multiplier column and the elimination takes place over all feasible entries. The only major additions to the algorithm are the broadcasts of the pivot row and the multiplier column. The pivot row is broadcast from process row $\hat{p}$ to all other process rows. The multiplier column is broadcast from process column $\hat{q}$ to all other process columns.

```
$\mathcal{I} := \{i : 0 \leq i < I\}$ ;
$\mathcal{J} := \{j : 0 \leq j < J\}$ ;
for $k = 0, 1, \ldots, \min(M, N) - 1$ do begin
    {Pivot Strategy and Bookkeeping.}
    do pivot search and find $a_{rc}, r[k], c[k]$ ;
    $\hat{p}, \hat{\imath} := \mu(r[k])$ ;
    $\hat{q}, \hat{\jmath} := \nu(c[k])$ ;
    if $p = \hat{p}$ then $\mathcal{I} := \mathcal{I} \setminus \{\hat{\imath}\}$ ;
    if $q = \hat{q}$ then $\mathcal{J} := \mathcal{J} \setminus \{\hat{\jmath}\}$ ;
    if $a_{rc} = 0.0$ then terminate ;

    {Broadcast the Pivot Row.}
    if $p = \hat{p}$ then begin
        for all $j \in \mathcal{J}$ do $a_r[j] := a[\hat{\imath}, j]$ ;
        send $a_r[j : j \in \mathcal{J}]$ to $(\bullet, q)$ ;
    end
    else receive $a_r[j : j \in \mathcal{J}]$ from $(\hat{p}, q)$ ;

    {Broadcast the Multiplier Column.}
    if $q = \hat{q}$ then begin
        for all $i \in \mathcal{I}$ do $a_c[i] := a[i, \hat{\jmath}]/a_{rc}$ ;
        send $a_c[i : i \in \mathcal{I}]$ to $(p, \bullet)$ ;
    end
    else receive $a_c[i : i \in \mathcal{I}]$ from $(p, \hat{q})$ ;

    {Elimination.}
    for all $(i, j) \in \mathcal{I} \times \mathcal{J}$ do $a[i, j] := a[i, j] - a_c[i]a_r[j]$
end
```

Figure 2: Concurrent LU-Decomposition with Implicit Pivoting.

3 Pivoting Strategies

In this section, we describe the pivoting strategies of our experiments.

The simplest strategy is to have no strategy at all. In the k-th elimination step, entry $a[k,k]$ of the matrix A is chosen as pivot. With the *no pivoting* strategy the sequence of pivot indices $(r[0],c[0])$, $(r[1],c[1]),\ldots,(r[M-1],c[M-1])$ is thus given by $(0,0)$, $(1,1),\ldots,(M-1,M-1)$. We generalize this by allowing an arbitrary predetermined sequence of pivot positions. The only a priori requirement on such a sequence is that each row and column index occurs at most once. We call this generalization *preset pivoting*. It is clear that no pivoting is just a special case. Any *static strategy* suffers from numerical instability. In some circumstances this may be an acceptable risk, in others enough might be known about the matrix to guarantee that the errors remain small.

Most pivoting strategies are *dynamic*, i.e., the decision which entry should be the pivot in the k-th step of the decomposition is postponed until the pivot is actually needed. *Complete pivoting* searches all feasible entries for the one that is largest in absolute value. This requires a search over $(M-k)(N-k)$ entries. Only LU-decomposition with complete pivoting is proven to be numerically stable, i.e., for any matrix the computed LU-decomposition is the LU-decomposition of a nearby matrix. By restricting the search of the k-th pivot to the feasible entries of the k-th column, the extent of the search is reduced to $(M-k)$ entries. This is called *row pivoting*. The k-th column does not offer a numerical advantage over other feasible subsets of size $(M-k)$. There are two other obvious partial pivoting choices that limit the search to $(M-k)$ entries: *column pivoting* and *diagonal pivoting*. Row pivoting is more popular only because it is somewhat easier to implement.

We have also used two intrinsically concurrent partial pivoting strategies. Let the matrix be distributed over a $P \times Q$ process grid. When searching a column (in row pivoting), only one process column is active. Without any overhead, the other process columns could each search another arbitrary feasible column. This increases the extent of the search and, hence, the extent of the class of matrices for which the LU-decomposition is numerically stable. Similarly, column pivoting can be enhanced. We refer to these concurrent alternatives as *multirow* and *multicolumn pivoting*.

Numerical stability is not the only criterion. In sparse matrix computations, the amount of fill generated by a particular choice of pivot is also a consideration. Many static and dynamic strategies have been developed to

```
$a_{rc} := 0$ ; $r[k] := -1$ ; $c[k] := -1$ ;
if $\mathcal{J} \neq \emptyset$ and $\mathcal{I} \neq \emptyset$ then begin
    {Select a Feasible Column and Find Maximum in it.}
    $\hat{\jmath}$ := any element of $\mathcal{J}$ ;
    $h := 0.0$ ;
    for $i \in \mathcal{I}$ do
        if $|a[i,\hat{\jmath}]| > h$ then begin $\hat{\imath} := i$ ; $h := |a[i,\hat{\jmath}]|$ end ;
    $a_{rc} := a[\hat{\imath},\hat{\jmath}]$ ; $r[k] := \mu^{-1}(p,\hat{\imath})$ ; $c[k] := \nu^{-1}(q,\hat{\jmath})$
end ;
{Find Global Maximum.}
$t := pQ + q$ ;
for $d = 0, 1, \ldots, \log_2(PQ) - 1$ do begin
    send $a_{rc}, r[k], c[k]$ to $t\bar{\vee}2^d$ ;
    receive $a^1_{rc}, r^1, c^1$ from $t\bar{\vee}2^d$ ;
    if ($|a_{rc}| < |a^1_{rc}|$) or
        ($|a_{rc}| = |a^1_{rc}|$ and $r[k] < r^1$) or
        ($|a_{rc}| = |a^1_{rc}|$ and $r[k] = r^1$ and $c[k] < c^1$)
    then begin
        $a_{rc} := a^1_{rc}$ ; $r[k] := r^1$ ; $c[k] := c^1$
    end
end
```

Figure 3: Concurrent Multirow Pivoting.

minimize fill, see, e.g., [6]. Some use generated fill as sole criterion, others combine fill and numerical stability considerations. To date, we have used only simple strategies similar to those of the full matrix case. Even with these, a significant performance gain over solving the full matrix system is obtained.

The concurrent implementation of any of the above pivoting strategies is fairly straightforward. As an example, we display multirow pivoting in Figure 3. The matrix distribution, the pivot history (as recorded by $\mathcal{I} \times \mathcal{J}$) and the pivoting strategy determines the local search set. For multirow pivoting, this happens to be all feasible entries of an arbitrary feasible column. A local search determines a local pivot candidate and its local indices. These are converted into global indices with the inverse functions of μ and ν. If the search set is empty, the local pivot candidate has the value zero and its indices have an invalid value like -1.

After the local search, the actual pivot is computed with a concurrent maximization over all processes. We use a *recursive doubling* procedure. This computes and simultaneously distributes the result, which consists of the pivot value and its global row and column indices. All processes participate in this recursive doubling procedure. For this part of the computation, the process grid structure is not relevant and it is notationally convenient to identify each process with a single process number instead of a process coordinate pair. We use the mapping:

$$t := pQ + q$$

but any bijection from $\{(p,q) : 0 \le p < P \text{ and } 0 \le q < Q\}$ to $\{t : 0 \le t < PQ\}$ would do. The expression $t\bar{\vee}2^d$ denotes the integer obtained from t by flipping bit number d of its binary expansion.

Recursive doubling compares successive pairs of pivot candidates and keeps the best one encountered thus far. For recursive doubling to work properly, the comparison function must be associative, see, e.g., [13]. The complicated comparison is to ensure associativity in case two or more matrix entries are equal in absolute value.

4 The Sparse Matrix Representation

The sparse LU-decomposition algorithm does not differ from the full version, except that the access to matrix entries is considerably more difficult. The address of an entry $a[m,n]$ is no longer computable with an easy integer expression, instead the entry is located by a search through a sparse matrix representation. In this section, we discuss our representation and the rationale behind its choice.

Let A be the $M \times N$ sparse matrix, which we want to distribute over a $P \times Q$ process grid. As for the full case, we use the distribution functions μ and ν to allocate entry $a[m,n]$ to a particular process (p,q). If A is sparse and the distribution functions μ and ν are arbitrary, we generally expect that each local matrix is sparse. One can certainly construct combinations of sparse matrices and distributions for which this is not true. The *generic case*, however, is that the local matrices are sparse. As we are interested to keep the correctness of our algorithms independent of the distributions, our main interest lies with the generic case. Each local matrix is thus represented in a

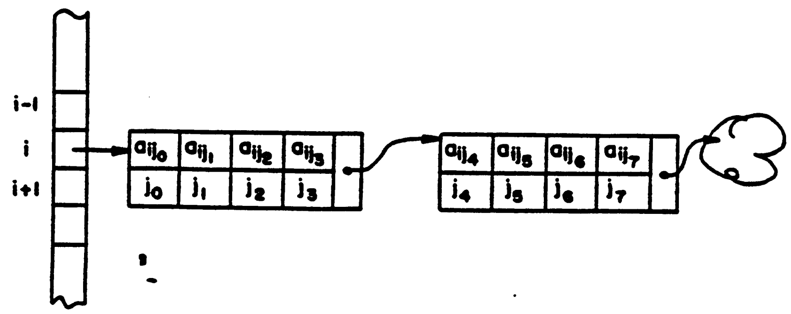

i-1
i
i+1
a_{ij_0} a_{ij_1} a_{ij_2} a_{ij_3}
j_0 j_1 j_2 j_3
a_{ij_4} a_{ij_5} a_{ij_6} a_{ij_7}
j_4 j_5 j_6 j_7

Figure 4: Compact Row Oriented Sparse Matrix Representation.

separate sparse structure. Any considerations regarding the implementation of this structure are of a purely sequential nature.

In sparse matrix representations, we must distinguish between structured and unstructured sparse matrices. Often, exploitation of a particular fill structure leads to simpler and more efficient representations. However, even if the matrix A has a particular structure, the local matrices generally have not. Again, it is possible for some matrices to construct particular distributions such that the local matrices are structured. The generic case, however, is that the local matrices are unstructured.

In our sparse matrix representation, we have tried to minimize the amount of bookkeeping information stored along with each entry. For each row of the matrix, we keep a set of nonzero entries in that row. The column index is stored along with each entry. A disadvantage of such a representation is that rows and columns are treated asymmetrically. Hence, a different cost is associated with accessing a row or a column. Because row accesses occur more frequently in our algorithms, we have optimized the structure accordingly.

The row set is represented as a list of blocks, see Figure 4. Each block contains a specified small number of entries together with their column index. E.g., a block might contain eight double precision locations for the matrix entries and eight integers for the column indices. In addition to this, a block contains a pointer to the next block in the list. Hence, one extra pointer is needed for every eight entries. The minimum memory allocation for each row is one block. Hence, some memory is wasted in the tail of the last block in the list. The entries of the row set are not kept in any sorted order as the cost of sorting far exceeds its benefits.

To access this structure conveniently, we use four routines: expand_row, compact_row, expand_column, compact_column. Like a gather operation, the expand routines translate a sparse row or column into a dense array and use an index set to indicate the locations of the nonzero entries. The compact routines perform the opposite operation and are thus similar to a scatter operation. With these routines, the elimination step takes the form:

```
$\mathcal{J}_d = \mathcal{J}$ ;
for $i \in \mathcal{I}_d$ do begin
    for $j \in \mathcal{J}_d$ do $a_e[j] := 0.0$ ;
    expand row $i$ into $a_e, \mathcal{J}_d$ ;
    $\mathcal{J}_d := \mathcal{J} \cap \mathcal{J}_d$ ;
    for $j \in \mathcal{J}_d$ do $a_e[j] := a_e[j] - a_c[i] * a_r[j]$ ;
    compactify $a_e[j], j \in \mathcal{J}_d$ into row $i$
end
```

The index set $\mathcal{I}_d$ is the intersection of $\mathcal{I}$, the feasible row indices, and the set of row indices of nonzero multipliers (determined previously). Similarly, $\mathcal{J}_d$ is the intersection of $\mathcal{J}$, the feasible column indices, and the set of column indices of nonzero pivot row entries.

5 Index Sets

The programs of Figures 1 and 2 almost immediately translate into implementations. The only complication is the management of arbitrary index sets. We use a *bit map representation* such that indices are easily activated and deactivated by setting or unsetting a bit. The intersections and unions of index sets are easily computed with logical bit operations.

A bit map representation is not effective to loop through all active indices. An obvious implementation runs through all indices and tests the bit map whether it is active or not, i.e.:

```
for $m = 0, 1, \ldots, M-1$ do
```

```
if $m \in \mathcal{M}$ then begin
    ...
end
```

The conditional test prevents the vectorization of such a loop. For this reason, we convert the bit map into a more effective representation before looping through all active indices. An array of indices $r[l]$, where $0 \leq l < L$, is constructed such that all m satisfying $r[l] \leq m < r[l+1]$ for some even l are active. Conversely, all m satisfying the same inequalities for some odd l are inactive. A loop over all active indices is then implemented as follows:

```
for $l = 0, 2, \ldots, L-2$ do
    for $m = r[l], r[l]+1, \ldots, r[l+1]-1$ do begin
        ...
    end
```

The cost of the conversion and the overhead associated with the implementation of the loops is small, see Section 9. For large active ranges the interior loop vectorizes well. Only when the ranges are highly fragmented, does overhead become noticeable. True vectorization requires scatter-gather hardware.

6 The Reactive Kernel and Cosmic Environment

Our algorithms are implemented in C to run under the Reactive Kernel/-Cosmic Environment operating system, developed by Seitz et al. [10]. Here we briefly describe those parts relevant to our application. For details on the design of this system and its rationale we refer to [10].

The Reactive Kernel is the system running on each node of a multicomputer and allows a user to spawn and kill processes, send and receive messages. The Cosmic Environment runs under standard Berkeley 4.2BSD UNIX and provides the same capabilities for processes running on a network of work stations. The Cosmic Environment is also the interface between a user's work station and the multicomputer. The Reactive Kernel is currently available on the Cosmic Cube, the Intel iPSC-1, and Intel iPSC-2 hypercubes and on the Ametek 2010 multicomputer.

In our applications, we use six fundamental operations of the RK/CE system: *spawn, xmalloc, xfree, xsend, xmsend* and *xrecvb*. The spawn function creates a process on a multicomputer node. A process so created is identified through its unique ID = (node,pid), which consists of the node number and a process identification number. Spawn may initiate several processes per node, a feature that is not used in our current experiments.

Message memory is allocated with the routine *xmalloc*. The statement:

```
msg = (double *)xmalloc(10*sizeof(double)) ;
```

allocates space from the message heap for ten double precision values. The statement:

```
xfree(msg) ;
```

returns this space to the heap for future use. A buffer, once allocated, is sent to process (node,pid) with:

```
xsend(msg,node,pid) ;
```

This statement simultaneously frees the buffer msg, i.e., for the sending process xsend has the same effect as xfree. This avoids all synchronization problems that might arise from refilling a buffer before the send operation is actually completed. The function xmsend sends identical data to a list of processes. Such broadcasts are efficiently implemented at the level of the Reactive Kernel. The statement:

```
msg = xrecvb() ;
```

is equivalent to an xmalloc, except that the length of the buffer is determined by the arriving message. Note that xrecvb does not provide any selectivity: the next message must be accepted and the routine does not return until a message has arrived.

If message selectivity is important for a particular application, as it is for ours, such capability must be layered on top of the reactive primitives. We have a number of communication routines that implement process grids. These also buffer incoming messages that cannot be processed immediately. We omit a discussion of this straightforward sequential code.

7 Multicomputer Hardware

The experiments on which we report in this paper were performed on the Ametek 2010. In this section, we give a brief description of its hardware and list performance information relevant for the interpretation of our computational experiments. For details we refer to [12].

Operation	Operations per Second
x+=y	4.64468e+05
x*=y	3.29924e+05
x/=y	2.50501e+05
x=x+y*z	3.88350e+05
x[i]=x[i]+y[i]*z[i]	6.95846e+04
i+=j	9.34579e+06
i*=j	5.24659e+05
i=i+j*k	9.43396e+06

Table 2: Number of Operations per Second for the Ametek 2010 Processor. The variables x, y, z, x[i], y[i], and z[i] represent double precision values, i,j, and k are integers.

The Ametek 2010 is a multicomputer and thus consist of a collection of independent computers connected by a communication network. The CPU of the Ametek 2010 nodes is the Motorolla 68020. Each node has a 4 Mbyte memory. At the time of writing this paper, Caltech's Ametek 2010 configuration has 16 nodes. Table 2 displays node performance in operations per second for typical operations expressed in C. Overhead due to looping was made negligible in these performance measurements. A suggestion of Kennedy [8] helped us to improve significantly the accuracy of these timings. From Table 2, we may conclude that the Ametek 2010 nodes should execute at a sustained performance level between 50 and 100 kFLOPS. We used the optimizing GNU C-compiler for all the experiments in this paper.

The topology of the communication network is a rectangular mesh. It features *wormhole routing*: a message header reserves all necessary channels between source and destination and, subsequently, the message is transmitted. The principal advantage is that the communication time is effectively independent of the network distance between source and destination node. (Network distance is important for the header only.) Every Ametek 2010 node features a microprogrammed second processor to manage send and receive queues. Figure 5 displays the communication time as a function of the length of the message. These times were obtained by sending a message 1,000 times around a ring of 16 processes, one process to each multicomputer node. This time was measured and divided by 16,000. This gives an average time for the combined cost of sending and receiving a message. These timings were performed with the reactive primitives (solid curve) and

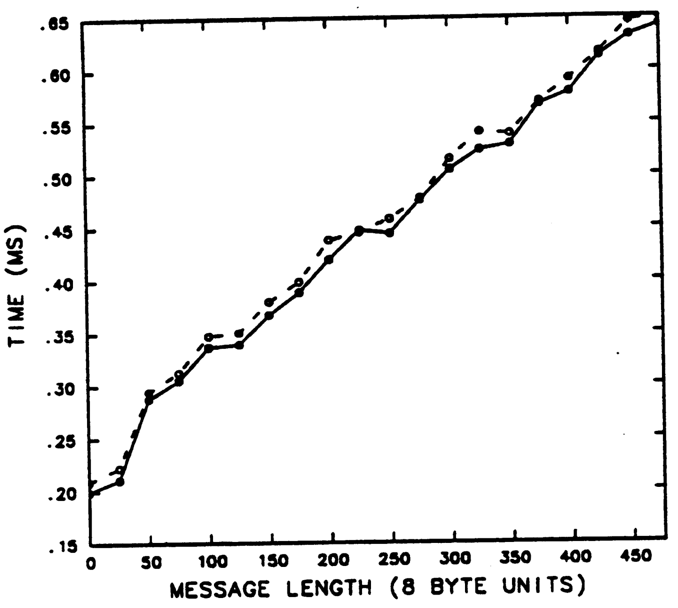


Figure 5: Communication Time as a Function of Message Length on the Ametek 2010.

with our communication routines layered on top of the reactive primitives (dashed curve). This shows that the overhead is negligible for all but the shortest messages.

We shall model the communication time as a linear function of message length, i.e.:

$$\tau_C(L) = \tau_S + \tau_M L. \tag{4}$$

The value $\tau_C(L)$ is the *communication time* of a message of length L. The value τ_S is the *start up time* and τ_M is the *marginal time*. Figure 5 essentially confirms this assumption, although there are some deviations. This simple timing does not capture the possibility of a congested network. Then, messages interfere with one another and the simple linear relationship breaks down. Efficient programs avoid such congestion.

8 Performance Analysis

The performance analysis of our programs requires a substantial number of simplifying assumptions. The real performance is greatly influenced by a subtle interplay between the distribution functions μ and ν and the pivot positions as determined by the pivoting strategy and the matrix. This is difficult to capture in a simple formula. We restrict ourselves to a best case analysis, showing that the ideal concurrent performance is reachable, at least in some limiting sense. The experimental results of Sections 9 and 10 establish the limits of the validity of this analysis. We only deal with square full matrices. Most conclusions carry over to the sparse matrix case, but the simplifying assumptions only hold in a far more restricted set of circumstances.

We assume that the feasible rows and columns are uniformly distributed over the process grid throughout the computation. This is the case if the matrix distribution and the pivoting strategy are such that the locations of pivot rows and columns in the process grid are essentially random. For any matrix distribution and for any pivoting strategy, all rows and columns eventually become infeasible. Some processes will become idle sooner than others. This effect is negligible for coarse grained computations, i.e., when the problem is large compared with the number of nodes ($MN \gg PQ$).

We assume that a broadcast over P processes takes a factor $\log_2 P$ times longer than a simple send and receive operation. This is actually an overestimate as broadcasts are handled efficiently by the system, see Section 6.

These assumptions lead to the following estimate for the execution time of the k-th LU-decomposition step on a $P \times Q$ process grid, each process executing on a dedicated processor:

$$\begin{aligned} T_{PQ}^{k} &= \tau_C\left(\frac{M-1-k}{P}\right)\log_2 P \\ &+ \frac{M-1-k}{P}\tau_A + \tau_C\left(\frac{M-1-k}{Q}\right)\log_2 Q \\ &+ \frac{(M-1-k)^2}{PQ}\tau_A. \end{aligned}$$

The first term corresponds to the broadcast of the pivot row, the second and third term to the calculation and broadcast of the multiplier column and the last term to the elimination. The value τ_A is the *arithmetic time*, i.e., the average time for a floating point operation. For simplicity, we have not included the computation and communication cost of the pivoting strategy.

Summing the above for $k = 0, \ldots, M-1$ and using Equation 4, we obtain the following execution time for LU-decomposition:

$$\begin{aligned} T_{PQ} &\approx M \log_2(PQ)\tau_S \\ &+ \left(\frac{\log_2 P}{P} + \frac{\log_2 Q}{Q}\right)\frac{M^2}{2}\tau_M \\ &+ \left(\frac{M^2}{2P} + \frac{M^3}{3PQ}\right)\tau_A. \end{aligned} \tag{5}$$

The start up time leads to an overhead term that increases logarithmically with the total number of processes. However, this term is a lower order term with respect to the matrix dimension M. The marginal term of the communication cost is minimized by choosing $P = Q$. Because of this term, the best performance is generally obtained with two dimensional process grids. The arithmetic time has two terms. The smallest order term results from the multiplier computation which is distributed over P processes only. This load imbalance can be avoided by choosing $Q = 1$. However, for large problems this term is negligible compared to the main computational cost, which is proportional to M^3 and speeds up linearly with the number of processes. The leading term of this equation is nothing but the execution time of the best sequential algorithm:

$$T_1 \approx \frac{M^3}{3}\tau_A$$

divided by the total number of processes PQ.

The effectiveness of a concurrent formulation is usually examined with the aid of speed up and efficiency calculations. As a practical measure, speed up is often plagued by ambiguities. The best sequential algorithm may be well defined, its best implementation is not. As an extreme example, it is unreasonable to use a sequential assembly language program as a standard for a concurrent program written in some high level language. A good sequential standard has the same level of code optimizations as the concurrent program. This is not as clear cut a criterion as one would wish. Measured speed up often depends on irrelevant implementation details of the sequential program.

Considering this pitfall, we do not display our results as speed up graphs. Instead, we give the observed execution times as a function of the number of nodes. Speed up graphs do have the advantage that ideal concurrency corresponds to a linear speed up, i.e., $T_{PQ} = \frac{T_1}{PQ}$. Following a suggestion of

Seitz [11], we display the execution times in a log-log plot, concurrent execution time versus number of processes. Ideal concurrency then translates into:

$$\log T_{PQ} = \log T_1 - \log(PQ),$$

i.e., a straight line with slope -1. Without losing this linear relationship, we may use $\log_{10}$ for the execution times and $\log_2$ for the number of nodes. In graphs, we display the ideal speed up curve with a solid line. We list separately the measured speed up and efficiency of some computations. For this reason, it is important to verify that the quoted sequential times are reasonable, i.e., are compatible with the elementary timings of Section 7.

9 Experiments with Full Matrices

The cost of pivoting obviously depends on the strategy. It also depends on the matrix itself because it determines the order in which rows and columns become infeasible. This and the distribution determine the load balance. Hence, every matrix has a different performance characteristic. For most matrices and most pivoting strategies the pivot locations in the process grid are more or less random. There are notable exceptions, however. E.g., the pivots of diagonally dominant matrices with complete or partial pivoting are on the main diagonal. The matrix used in all our full matrix experiments is given by:

$$A = [a_{m,n}] = [\cos((m+1)*(n+1))],$$

where $0 \leq m, n < 300$. The symmetry of this matrix is never used in any of the algorithms. This matrix is easily generated and behaves well numerically, which is necessary for timing many pivoting strategies, some of which are numerically unstable. We are not aware of any special properties this matrix might have regarding its pivot locations. A limited number of tests with other matrices confirmed that the obtained performance results are typical.

The main subject of this study is the influence of pivoting strategy and data distribution on performance. Other variables like the size of the problem are kept constant. The main application of concurrent computers is to solve large problems. Our principle goal is to make the computations efficient in this regime. However, measuring the efficiency of a concurrent computation requires timings over a range of machine sizes. Large problems cannot be solved on a small number of nodes due to memory and time

Program	Execution Time (ms)	FLOPS
Classical	1.63001e+05	5.52144e+04
Index Sets	1.68948e+05	5.32708e+04

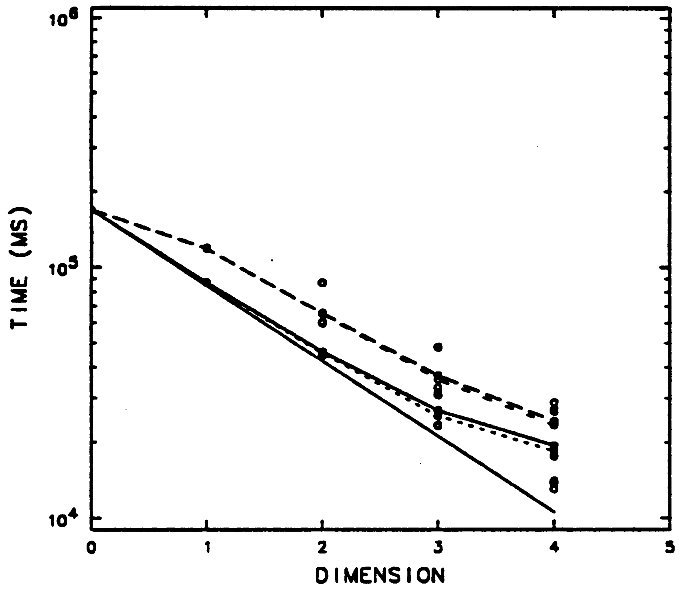
Table 3: Execution Times and Floating Point Operations per Second for Two Sequential LU-Decomposition Programs.

restrictions. Because we did not want to introduce any artificial times (obtained through extrapolation of timings of small problems), our standard problem is relatively small. For our full matrix timings, we used a 300 by 300 double precision matrix. The efficiencies for problems of this size are easily surpassed by larger problems.

Before turning our attention to the main experiments, we address two questions regarding the sequential program, the standard with which the concurrent program is compared. First, based on the elementary timings of Section 7, do we attain the expected performance? Second, is the overhead associated with index set manipulations reasonable? Table 3 displays the execution times and floating point performance for two implementations of LU-decomposition without pivoting. The first program is a classical implementation not involving index set manipulations. The second is an implementation of the program displayed in Figure 1. LU-decomposition without pivoting requires approximately $M^3/3$ floating point operations. The third column of Table 3 gives the estimated number of floating point operations per second based on this estimate. Floating point performance of both programs is somewhat lower than that of the indexed variables in Table 2. This is due to the higher complexity of realistic programs over that of simple benchmarks. Comparing the two implementations of LU-decomposition, it is clear that the overhead of index set manipulations is negligible.

9.1 No Pivoting

In Figure 6, we graph the concurrent execution times of LU-decomposition without pivoting. For each machine size, we display times for every possible matrix distribution that can be obtained by a combination of the linear and scatter distributions of rows and columns. As discussed in Section 8, the solid straight line corresponds to ideal speed up. The other curves correspond to one dimensional distributions, i.e., $P \times 1$ or $1 \times Q$ process grids with either linear or scatter distribution. The table in Figure 6 summarizes



No Pivoting		
	Minimum	Maximum
Speed Up	5.9	13.0
Efficiency	36.6%	81.3%
Process Grid	4 x 4	4 x 4
Row Distribution	linear	scatter
Column Distribution	linear	scatter

Figure 6: LU-Decomposition Without Pivoting.

the characteristics of the least and most effective 16 node timings.

LU-decomposition without pivoting does not satisfy the assumptions of Section 8. However, because the pivot positions of subsequent decomposition steps are so regular, its behavior can be predicted without introducing any simplifications. A scatter distribution of both rows and columns ensures that successive decomposition steps make infeasible matrix rows (columns) in different process rows (columns). As a result, all processes are kept active for a maximum time. This property is specific to this combination of the scatter distribution and the no pivoting strategy. The 4×4 process grid configuration minimizes the marginal term of the performance estimate, see Equation 5. The combination of a 4×4 process grid, and row and column scatter distribution thus ensures optimal performance. The speed up obtained for this execution is thus determined solely by the intrinsic load imbalance of the computation (eventually all rows and all columns become inactive) and by the communication cost of the multicomputer.

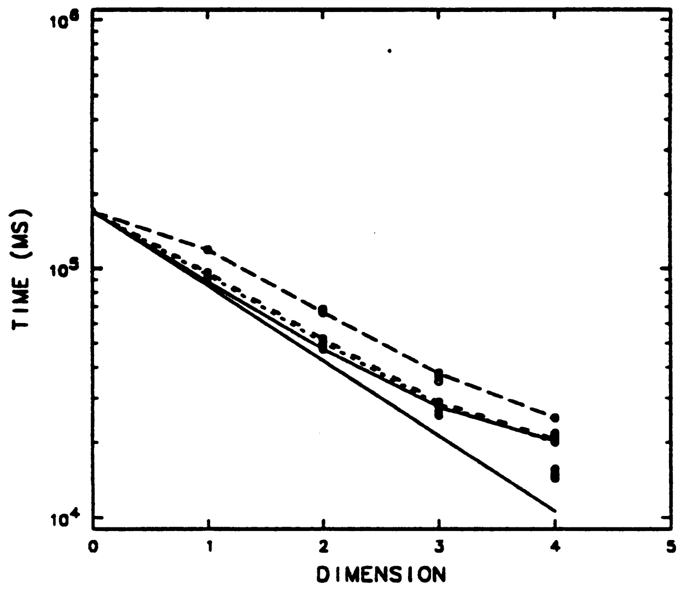
9.2 Row Pivoting

The performance of LU-decomposition with row pivoting is displayed in Figure 7. Characteristics of best and worst performers on the 16 node machine are summarized in the accompanying table.

While row pivoting has a negligible impact on the execution time of the sequential program, it influences considerably the concurrent execution. The worst performance has improved by about 5.0% in efficiency. The execution times for different distributions are more clustered. The underlying matrix distribution still determines performance to a large extent but execution times are less sensitive to it: the efficiency range dropped from 44.7% to 31.4%. Comparing the best case of each, row pivoting is somewhat less efficient than no pivoting. As indicated earlier, the latter minimizes load imbalance because matrix distribution and pivoting strategy interfere constructively. With a dynamic pivoting strategy this could only happen through a remarkable coincidence.

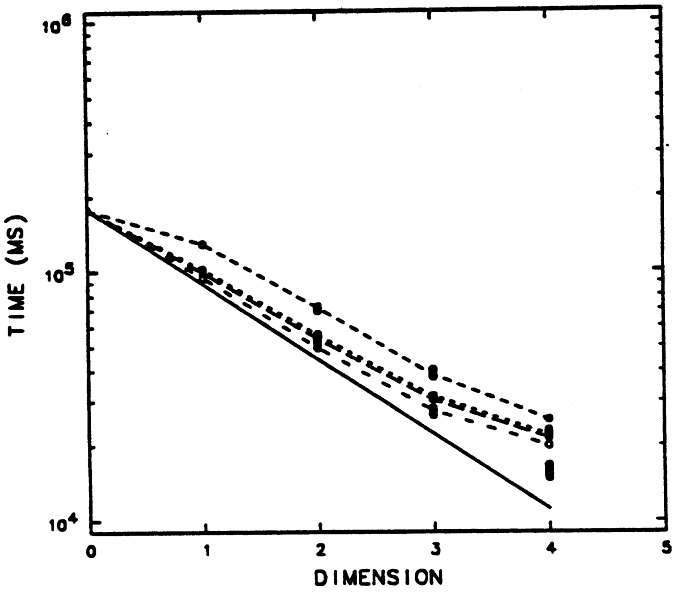
9.3 Column Pivoting

Figure 8 summarizes the performance of LU-decomposition with column pivoting. For the calculation of speed up and efficiency, it is important to point out that the sequential execution time for column pivoting is somewhat larger than that for row pivoting. This is because the LU-decomposition



Row Pivoting		
	Minimum	Maximum
Speed Up	6.8	11.8
Efficiency	42.4%	73.8%
Process Grid	1 x 16	4 x 4
Row Distribution		scatter
Column Distribution	linear	scatter

Figure 7: LU-Decomposition With Row Pivoting.



Column Pivoting		
	Minimum	Maximum
Speed Up	6.7	11.5
Efficiency	42.0%	71.8%
Process Grid	16 × 1	4 × 4
Row Distribution	linear	scatter
Column Distribution		scatter

Figure 8: LU-Decomposition With Column Pivoting.

program is more sensitive to fragmentation of the column index set than to fragmentation of the row index set. This minor technical detail is easily corrected by reversing the order of the two nested loops in the elimination step. One then obtains a sequential execution time nearly identical to row pivoting. The displayed times in Figure 8 are obtained without carrying out this loop reversal because we wanted to vary as few parameters as possible during the experiment. To keep the speed ups and efficiencies of the tables in Figures 8 and 7 comparable, we used the sequential row pivoting time as a standard. The minor differences in performance between column and row pivoting in favor of the latter are due to the index set effect. Note that our test matrix is symmetric: for general matrices the results of row and column pivoting are not as strongly correlated.

9.4 Diagonal Pivoting

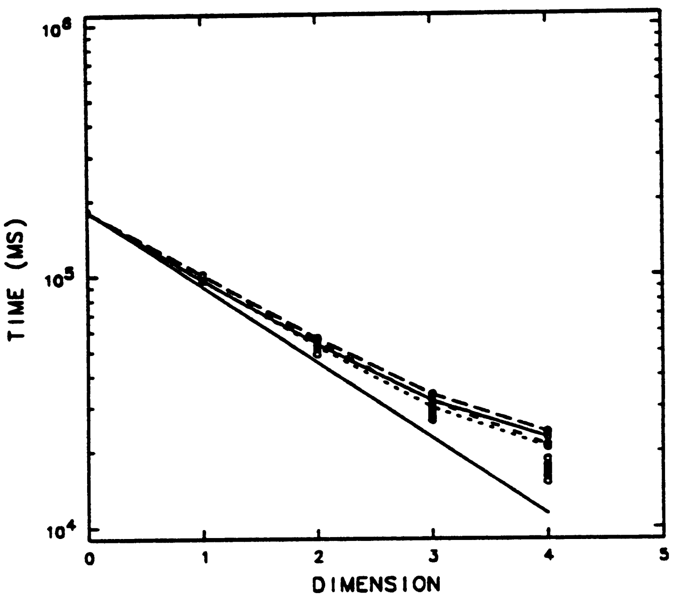
As in Section 9.3, the sequential time is higher than with row pivoting due to fragmentation of the index sets. In this case, a simple loop reversal cannot remedy the problem as fragmentation occurs in both the row and the column index sets. Again, we have used row pivoting as the sequential standard to compute speed ups and efficiencies.

9.5 Complete Pivoting

Complete pivoting guarantees numerical stability at a considerable expense. From Figure 10, it is clear that the pivot search cost plays a major role. The sequential execution time is slowed down by a factor of about 1.62 with respect to row pivoting. The efficiency range is much narrower than the previous two cases, 22.8%. The best efficiency approaches that of the best no pivoting case, which, as pointed out before, is optimal. Both these observations show that load imbalance problems are reduced by complete pivoting. The cost of complete pivoting, however, overshadows load imbalance considerations.

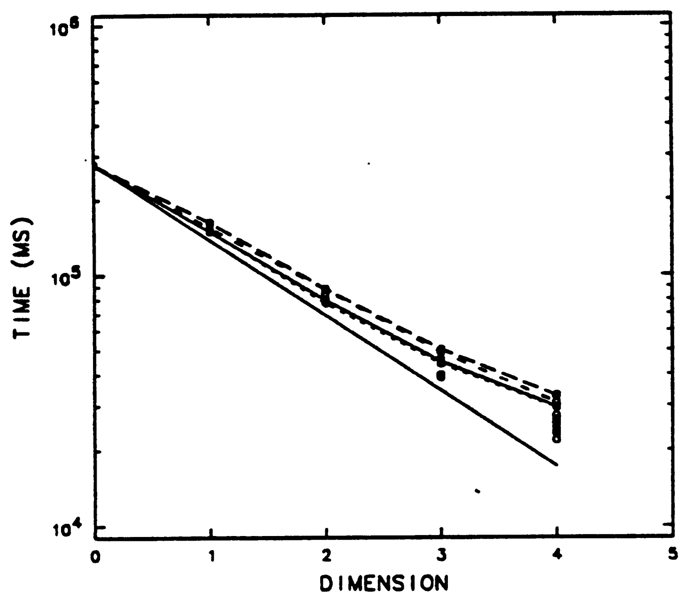
9.6 Multirow Pivoting

A logical question to pursue is whether the narrow efficiency range of complete pivoting can be obtained without the cost of a full search. The reason for a narrow range is that the pivot locations of successive decomposition



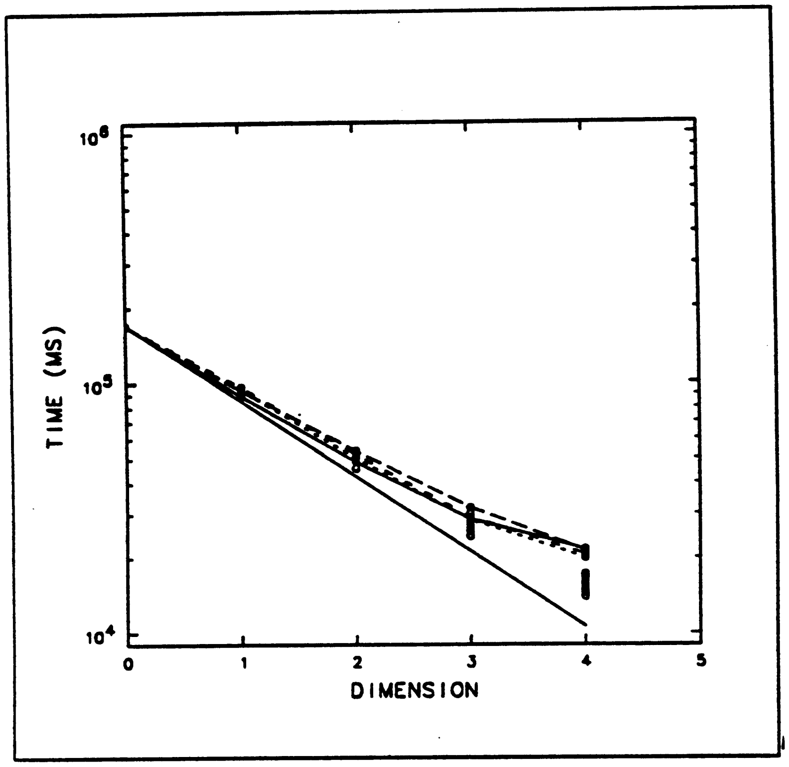
Diagonal Pivoting		
	Minimum	Maximum
Speed Up	7.1	11.3
Efficiency	44.6%	70.4%
Process Grid	1 x 16	4 x 4
Row Distribution		scatter
Column Distribution	linear	scatter

Figure 9: LU-Decomposition With Diagonal Pivoting.



Complete Pivoting		
	Minimum	Maximum
Speed Up	9.0	12.6
Efficiency	56.0%	78.8%
Process Grid	1 × 16	4 × 4
Row Distribution		scatter
Column Distribution	linear	scatter

Figure 10: LU-Decomposition With Complete Pivoting.



Multirow Pivoting		
	Minimum	Maximum
Speed Up	7.9	12.2
Efficiency	49.1%	76.3%
Process Grid	1 x 16	4 x 4
Row Distribution		scatter
Column Distribution	linear	scatter

Figure 11: LU-Decomposition With Multirow Pivoting.

steps are more random with respect to the process grid. Like complete pivoting, multirow pivoting introduces extra unpredictability into the algorithm: which process column contains the next infeasible column. However, it does this without increasing the pivot search cost. Figure 11 displays the results. The efficiency range of 27.2% is intermediate between row and complete pivoting. We note that, strictly speaking, the sequential and concurrent programs are not equivalent: the concurrent program has a more extended pivot search and, hence, performs a different calculation for different values of Q. Multirow and row pivoting are equivalent when $Q = 1$.

9.7 Multicolumn Pivoting

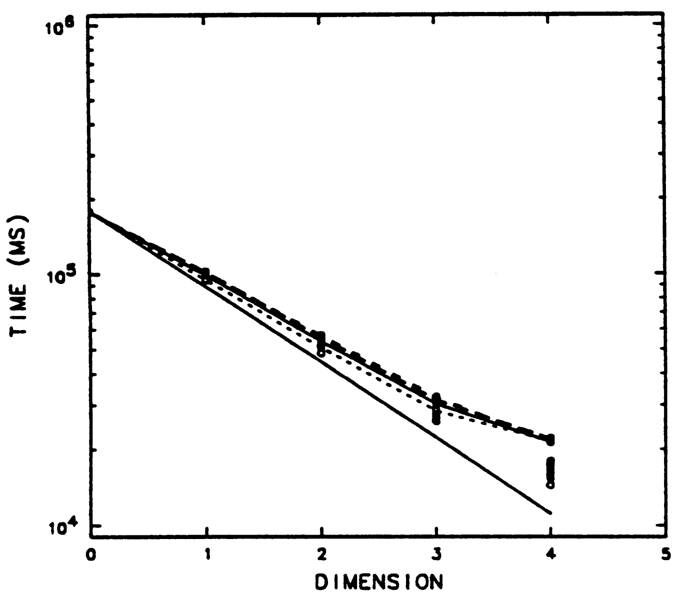
As in Section 9.3, we use partial row pivoting as sequential standard. We obtain Figure 12. For both multirow and multicolumn pivoting, the one dimensional process grids are the worst performers. A two dimensional process grid is thus preferred.

9.8 Randomized Distributions

To test the claim that, for performance considerations, pivoting is equivalent to randomizing the data distribution, we timed LU-decomposition without pivoting on a pseudo-randomly distributed matrix. By this we mean that the matrix rows are allocated to random process rows. Similarly, the matrix columns are allocated to random process columns. To generate the row distribution, we construct a permutation of the sequence of global row indices $0, 1, \ldots, M-1$ by applying $100 \times M$ random pairwise permutations to it. The latter are obtained with a pseudo-random number generator. This permutation is split up linearly among the process rows. E.g., for $M = 10$ and $P = 4$, assume the permutation of the row indices is given by the first column of Table 4. The second and third column give the corresponding process row and local index. Another permutation is used to find a similar pseudo-random distribution for the columns.

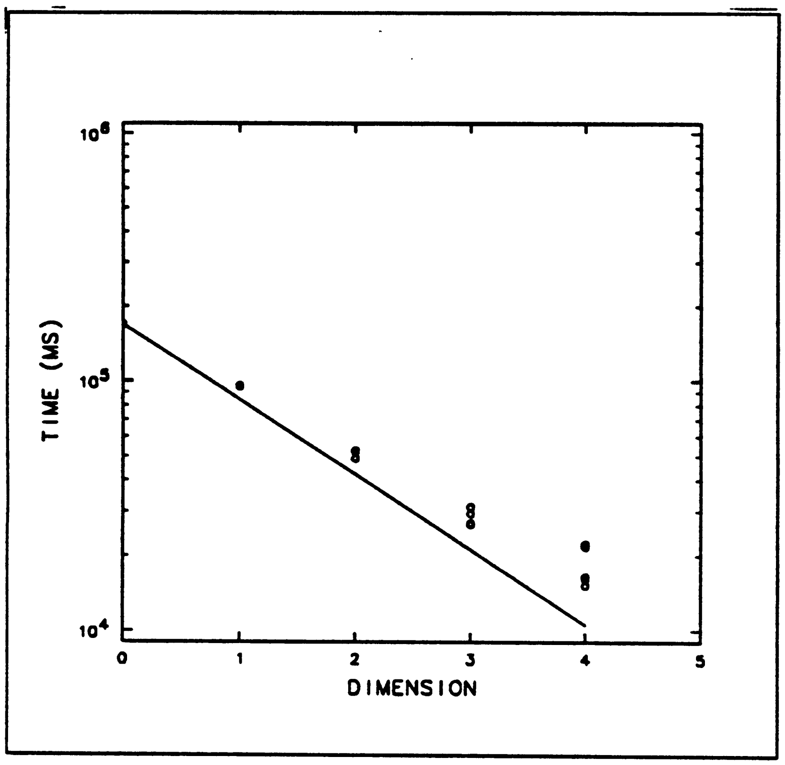
In Figure 13, we show the times obtained for several process grid configurations. The execution times coincide with the best times obtained thus far. In particular, it shows that the scatter distribution for all practical purposes is as good as a truly random distribution.

9.9 Preset Random Pivoting



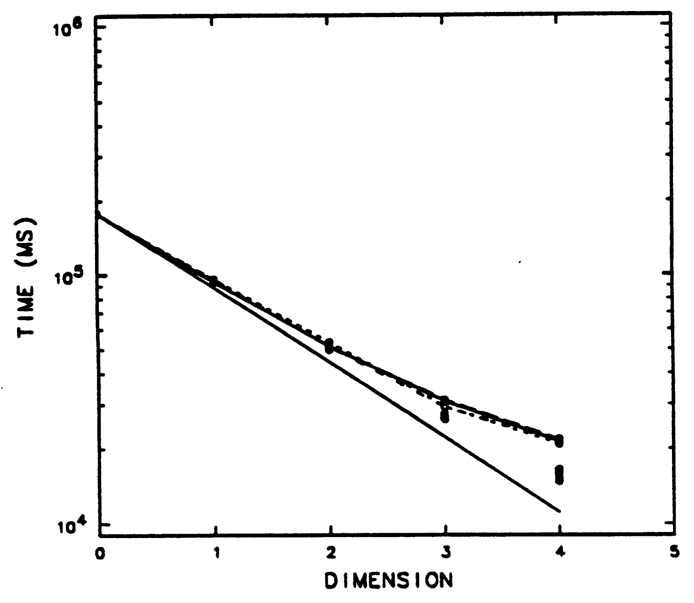
Multicolumn Pivoting		
	Minimum	Maximum
Speed Up	7.7	11.8
Efficiency	48.3%	73.7%
Process Grid	1 × 16	4 × 4
Row Distribution		scatter
Column Distribution	linear	scatter

Figure 12: LU-Decomposition With Multicolumn Pivoting.



Randomly Distributed, No Pivoting		
	Minimum	Maximum
Speed Up	7.6	11.1
Efficiency	47.3%	69.5%
Process Grid	1 × 16	4 × 4
Row Distribution	random	random
Column Distribution	random	random

Figure 13: LU-Decomposition Without Pivoting for Pseudo-Randomly Distributed Matrices.



Preset Random Pivoting		
	Minimum	Maximum
Speed Up	8.2	12.0
Efficiency	51.1%	75.3%
Process Grid	1 x 16	4 x 4
Row Distribution	linear	linear
Column Distribution	linear	linear

Figure 14: LU-Decomposition With Preset Randomized Pivoting.

Global	Process	Local
6	0	0
8	0	1
5	0	2
0	1	0
4	1	1
1	1	2
3	2	0
9	2	1
7	3	0
2	3	1

Table 4: Generation of a Pseudo-Random Distribution.

We also used a randomized pivoting strategy. This is easier to use because no difficult data distributions are involved. However, with preset random pivoting one loses all control over the numerical stability of the algorithm. (With a randomized distribution, pivoting strategies for numerical stability can still be incorporated.) Figure 14 summarizes the results.

9.10 A Performance Comparison

In Table 5, we compare the performance on the 16 node machine for the different matrix distributions and pivoting strategies. The underlined times are the minimal ones for the given distribution. No pivoting is the most efficient alternative in only five out of sixteen instances. For five other distributions, random pivoting wins. Multirow pivoting is fastest the other six times. Even when nonoptimal, multirow pivoting is a good choice as its extra cost is usually negligible. It is not fair, however, to compare the dynamic strategies with the static ones, which may be numerically unstable. Among the dynamic strategies, multirow pivoting is the fastest, except for a 1×16 scattered column distribution, where multicolumn pivoting wins out.

10 Experiments with Sparse Matrices

To test our sparse matrix programs, we used a matrix obtained from the discretization of the three dimensional Laplace operator on a regular grid. We chose this operator because it is representative of many sparse matrices in

Distribution	No	Diagonal	Multi-row	Multi-column	Random
16 × 1 L-	18.384	21.152	20.590	21.251	21.336
16 × 1 S-	23.309	20.709	19.933	21.410	20.815
8 × 2 L-L	26.457	17.615	15.898	17.957	15.811
8 × 2 L-S	18.858	16.826	15.667	17.334	15.717
8 × 2 S-L	17.551	16.741	15.742	16.664	15.934
8 × 2 S-S	13.708	16.070	14.428	15.551	15.837
4 × 4 L-L	28.784	16.787	17.096	17.668	14.753
4 × 4 L-S	18.179	15.816	14.893	16.013	14.799
4 × 4 S-L	17.878	15.777	15.221	15.362	15.301
4 × 4 S-S	12.990	15.079	13.914	14.407	15.360
2 × 8 L-L	27.076	18.519	17.182	17.297	16.442
2 × 8 L-S	18.302	17.402	16.206	16.995	16.251
2 × 8 S-L	19.047	17.469	16.566	16.648	16.448
2 × 8 S-S	14.037	16.479	14.966	15.452	16.267
1 × 16 -L	24.258	23.808	21.275	21.993	21.733
1 × 16 -S	19.369	22.652	21.603	21.319	21.378

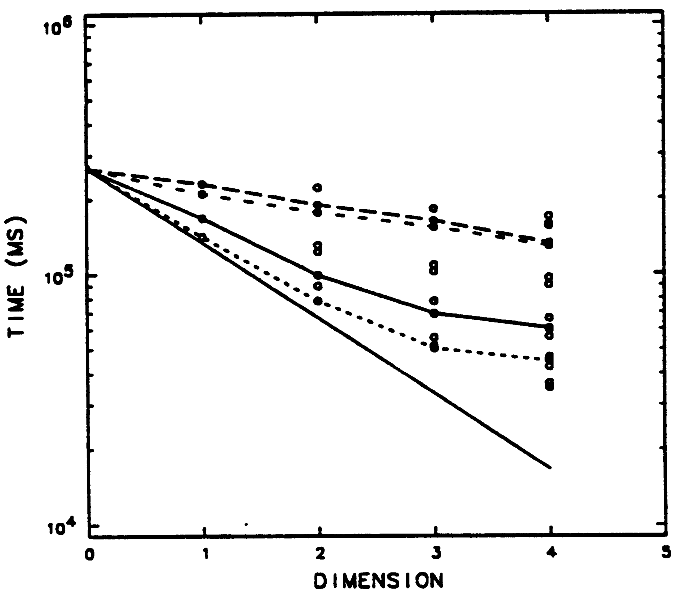
Table 5: Execution Times in Seconds for 16 Node LU-Decomposition. In the first column, L stands for linear and S for scatter distribution.

terms of amount of fill. The Laplace operator is reasonably well conditioned on coarse grids and it is the basis for a repeatable experiment. With lexicographic ordering of the grid points, its fill is highly structured. This is not not typical, but we do not exploit it in any of our computations. On the contrary, any fill structure is destroyed by the matrix distribution. The program initializing the Laplace matrix is general and allows Dirichlet, Neumann and Newton boundary conditions. For this reason, boundary points are treated as unknowns, even in the Dirichlet case. The equations corresponding to Dirichlet boundary points are of course trivial. They do influence timings, however, because of the resulting larger matrix dimensions.

The arguments given in Section 9 regarding the size of the problem, also apply here. Analogous tradeoffs led us to choose a sparse matrix resulting from discretizing Laplaces equation on a 10 by 10 by 10 grid. This problem has a total of 1331 unknowns of which 602 are trivial Dirichlet boundary unknowns (the ratio of boundary versus interior points goes down quickly as the problem size is increased).

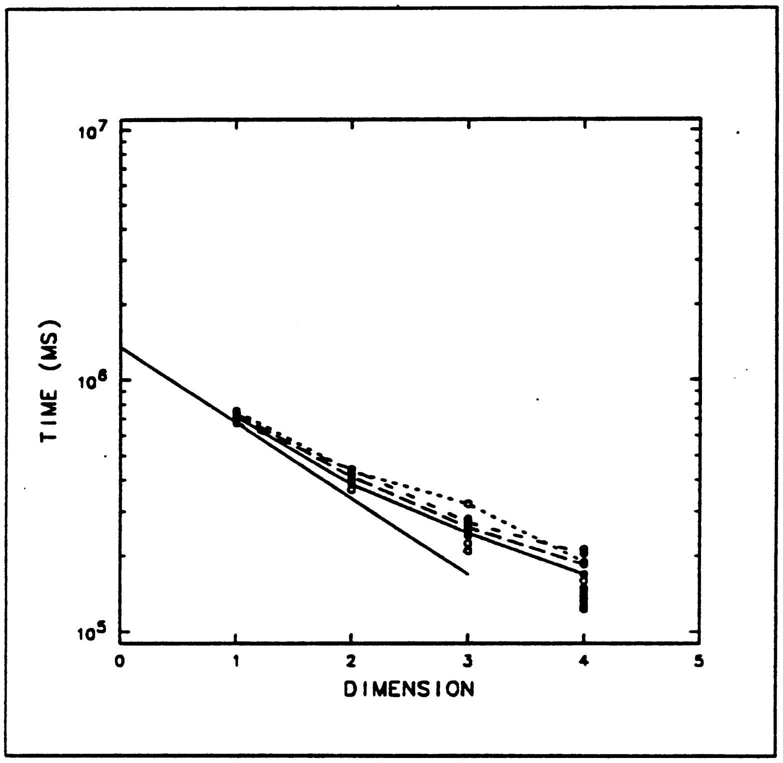
The execution time of the sequential sparse matrix program is dominated by the cost of accessing matrix entries. Because this is a complicated operation, it is difficult to verify objectively whether the sequential program executes at an expected performance level. We only note here that the LU-decomposition without pivoting of our sparse 1331×1331 matrix is done in about the same time as the full 300×300 case.

As is apparent from Figure 15, the matrix distribution is of overriding importance in sparse LU-decomposition without pivoting. We also tested our sparse solver with a pivoting strategy. Due to the near diagonal dominance of our test problem, row and column pivoting strategies choose the same sequence of pivots as no pivoting and complete pivoting is equivalent to diagonal pivoting. Hence, we only time the latter one. This pivoting strategy does not take into account any consideration of fill. Resulting memory limitations prohibited us from measuring one node times. The ideal speed up curve is drawn with the best 2 node timing as a standard. Any speed up for this algorithm is with respect to this time as well. Errors introduced by this are likely to be small. From Figure 16 it follows that pivoting has a major positive impact on the load imbalance. The range of efficiencies has narrowed considerably. It is also apparent, however, that the cost of simple pivoting strategies in sparse matrix calculations is substantial (note that the scale of Figure 16 has jumped an order of magnitude with respect to the other figures).



Sparse No Pivoting		
	Minimum	Maximum
Speed Up	1.6	7.6
Efficiency	9.8%	47.5%
Process Grid	4 x 4	4 x 4
Row Distribution	linear	scatter
Column Distribution	linear	scatter

Figure 15: Sparse LU-Decomposition Without Pivoting.



Sparse Diagonal Pivoting		
	Minimum	Maximum
Speed Up	6.3	11.0
Efficiency	39.5%	68.7%
Process Grid	2 x 8	4 x 4
Row Distribution	scattered	scatter
Column Distribution	scattered	linear

Figure 16: Sparse LU-Decomposition With Diagonal Pivoting.

11 Conclusion

The efficiency of LU-decomposition is determined mainly by load imbalance effects, which are determined by the matrix distribution and pivoting strategy. Efficiency generally increases if the pivoting strategy returns pivots in uniformly distributed locations and/or if the matrix entries are scattered uniformly over the processes. The scatter distribution in both rows and columns is an effective randomizer of the computation and generally results in a near optimal or optimal computation.

In a practical context, our LU-decomposition program can be used in two ways. Either, the matrix distribution is considered a given and the LU-decomposition is applied to it irrespective of performance considerations. It can also be used to find the best possible distribution for a particular problem.

Further research into pivoting strategies, especially for sparse matrices, is necessary. Our work shows that the ideal pivoting strategy not only maximizes numerical stability and minimizes fill, it also must randomize the computation. Whether all these criteria can be met in practice, remains an area of active research.

Acknowledgements

The lectures of K. Mani Chandy, based on [2], helped me to look at concurrent programs from a fresh viewpoint. Subsequently, we have had many fruitful discussions which have contributed significantly to this work. Wen-King Su and Jakov Seizovic helped the author with the details of the Cosmic Environment and Reactive Kernel.

References

[1] R.M. Chamberlain. An alternative view of LU factorization with partial pivoting on a hypercube multiprocessor. In M.T. Heath, editor, *Hypercube Processors 1987*, SIAM Publications, Philadelphia, PA, 1987.

[2] K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation.* Addison Wesley, 1988.

[3] E. Chu and J.A. George. Gaussian elimination with partial pivoting and load balancing on a multiprocessor. *Parallel Computing*, 5:65–74, 1987.

[4] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors.* Prentice Hall, 1988.

[5] G.A. Geist and M.T. Heath. Matrix factorization on a hypercube multiprocessor. In M.T. Heath, editor, *Hypercube Processors 1986*, SIAM Publications, Philadelphia, PA, 1986.

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

[7] P.G. Hipes and A. Kupperman. Gauss-Jordan inversion with pivoting on the Caltech Mark II hypercube. In G.C. Fox, editor, *Hypercube Concurrent Computers and Applications*, ACM Press, New York, NY, 1988.

[8] K. Kennedy. Private Communication.

[9] C.B. Moler. Matrix computation on a hypercube multiprocessor. In M.T. Heath, editor, *Hypercube Processors 1986*, SIAM Publications, Philadelphia, PA, 1986.

[10] C. L. Seitz, J. Seizovic, and W.-K. Su. *The C Programmer's Abbreviated Guide to Multicomputer Programming.* report CS 5252:TR:87, California Institute of Technology, 1987.

[11] C.L. Seitz. Private Communication.

[12] C.L. Seitz, W.C. Athas, C.M. Flaig, A.J. Martin, J. Seizovic, C.S. Steele, and W.-K. Su. The architecture and programming of the Ametek series 2010 multicomputer. In G.C. Fox, editor, *Hypercube Concurrent Computers and Applications*, ACM Press, New York, NY, 1988.

[13] H. S. Stone. *High Performance Computer Architecture.* Addison-Wesley, 1987.

[14] E. F. Van de Velde. *A Concurrent Direct Solver for Sparse Unstructured Systems.* report C3P-604, Caltech Concurrent Computation Project, 1988.

[15] E. F. Van de Velde. *The Formal Correctness of an LU-Decomposition Algorithm.* report C3P-625, Caltech Concurrent Computation Project, 1988.

[16] E. F. Van de Velde. *Introduction to Concurrent Scientific Computing.* Technical Report, Caltech, Applied Mathematics, 1989. Unpublished Lecture Notes.