CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 2(1), 1–26 (MARCH 1990)

Experiments with multicomputer LU-decomposition

ERIC F. VAN DE VELDE
Applied Mathematics 217-50
Caltech
Pasadena, CA 91125, USA

SUMMARY

We present a new concurrent LU-decomposition algorithm based on implicit pivoting of both rows and columns. This algorithm is, to a large extent, independent of the distribution of the matrix over the concurrent processes. As a result, it can be used in programs with dynamically varying data distributions. Another advantage is that most pivoting strategies are easily incorporated. We also introduce two new, intrinsically concurrent, pivoting strategies: multirow and multicolumn pivoting.

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 Symult 2010 and the iPSC/2 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 [2], Geist and Heath [3], Geist and Romine [4], and Moler [5] have examined varying combinations of row- and column-oriented distributions with row and column pivoting. Fox *et al.* [6], 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. For example, 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 matrix redistribution is then necessary between the initialization and LU-decomposition step.

It is also possible that the optimal data distribution changes as the computation proceeds. In this case, a significant performance gain may be achieved by changing the distribution dynamically. If the subroutines used by the program are valid for a range of data distributions, such dynamically changing data distributions are easily implemented.

Whether it is to avoid superfluous data redistributions or to make adaptive data distributions easily available, it is important to develop algorithms that are correct for a large class of distributions rather than for just one distribution. When a subroutine which is the implementation such an algorithm is called, the distribution is as much part of the data as the matrix itself, i.e., the distribution is user supplied and is not a property imposed by the algorithm. Our goal is to make possible optimizations of a

1040–3108/90/010001–26$13.00
Received 28 July 1989

global nature, i.e., optimizations affecting the total program and not just one component. We examine how this affects the construction of low level components, of which the LU-decomposition is one important example.

Data distribution is only one factor influencing performance. 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. The implementation of 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 will show that pivoting is also an effective randomizer of the computation and acts as a load balancer. Concurrent LU-decomposition is often faster with than without pivoting, i.e., the search cost is more than offset by the increased load balance. With pivoting, the performance of LU-decomposition is less sensitive to the particular matrix distribution, and 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 existing pivoting strategies. We shall also introduce two new strategies.

Most of our experiments were performed on both the Symult 2010 and the Intel iPSC/2. The results obtained 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, which is not the focus of this paper. We give full results only for the Symult 2010. In section 8, we give a selection of results on the iPSC/2.

Sections 2 and 3 describe the LU-decomposition and pivoting algorithms. Section 4 discusses an implementation detail. In section 5, we briefly describe the architecture of the Symult 2010 and its performance on elementary bench mark loops. In section 6, we give a performance analysis. In section 7, we report on our Symult 2010 experiments.

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* $\mathbf{A}$, *there exists an $M \times M$ permutation matrix* $\mathbf{R}$ *and an $N \times N$ permutation matrix* $\mathbf{C}$ *such that:*

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

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

For a proof, we refer to basic numerical analysis texts. In [8], the following theorem is obtained by permutation:

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:*

$$\mathbf{A} = \mathbf{L}\mathbf{C}^T\mathbf{I}_{N,M}\mathbf{R}\mathbf{U} \tag{1}$$

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

The matrix formed by the first M columns of $\mathbf{RLC}^T$ and the matrix $\mathbf{RUC}^T$ satisfy theorem 1 in the role of $\hat{\mathbf{L}}$ and $\hat{\mathbf{U}}$ respectively. Even though **L** and **U** have complicated structures, linear systems of the form $Lx = b$ or $Ux = b$ can be solved by a modified back-solve algorithm. For all practical purposes, the matrices **L** and **U** are as acceptable as the matrices $\hat{\mathbf{L}}$ and $\hat{\mathbf{U}}$.

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.

```
$\mathcal{M} := \{m : 0 \le m < M\}$ ;
$\mathcal{N} := \{n : 0 \le n < N\}$ ;
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

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.

A common way to implement concurrency on multicomputers is to use 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. For example, 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:

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

and the *scatter distribution* given by:

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

where $L = \lfloor \frac{M}{P} \rfloor$ 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 co-ordinate (p, q), where $0 \leq p < P$ and

Table 1. Four fold linear and scatter distribution

	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

$0 \leq q < Q$. 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, for example, 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 $\mathbf{A} = [a_{m,n}]$ distributed over a 2×2 process grid with a linear row and a scatter column distribution is stored as given by:

$$\begin{bmatrix} \mathbf{A}_{0,0} & \mathbf{A}_{0,1} \\ \mathbf{A}_{1,0} & \mathbf{A}_{1,1} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} a_{0,0} & a_{0,2} & a_{0,4} & a_{0,6} \\ a_{1,0} & a_{1,2} & a_{1,4} & a_{1,6} \\ a_{2,0} & a_{2,2} & a_{2,4} & a_{2,6} \end{bmatrix} & \begin{bmatrix} a_{0,1} & a_{0,3} & a_{0,5} \\ a_{1,1} & a_{1,3} & a_{1,5} \\ a_{2,1} & a_{2,3} & a_{2,5} \end{bmatrix} \\ \begin{bmatrix} a_{3,0} & a_{3,2} & a_{3,4} & a_{3,6} \\ a_{4,0} & a_{4,2} & a_{4,4} & a_{4,6} \end{bmatrix} & \begin{bmatrix} a_{3,1} & a_{3,3} & a_{3,5} \\ a_{4,1} & a_{4,3} & a_{4,5} \end{bmatrix} \end{bmatrix}$$

where $\mathbf{A}_{p,q}$ is the submatrix of $\mathbf{A}$ 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 in 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 [8]. 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. In contrast to explicit pivoting algorithms, no exchange of multiplier columns or pivot rows is necessary, and hence, some communication is eliminated.

```
$\mathcal{I} := \{i : 0 \le i < I\}$ ;
$\mathcal{J} := \{j : 0 \le j < J\}$ ;
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 $(r[0], c[0])$, $(r[1], c[1])$, ... , $(r[M-1], c[M-1])$ is thus given by $(0,0)$, $(1,1)$, ... , $(M-1, M-1)$. 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 once at most. 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 as to 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 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.*

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, the local search set contains all feasible entries of an arbitrary feasible column. A 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, e.g., -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 co-ordinate 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.

```
$a_{rc} := 0$ ; $r[k] := -1$ ; $c[k] := -1$ ;
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

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, for example, [9]. The complicated comparison is to ensure associativity in case two or more matrix entries are equal in absolute value.

4. INDEX SETS

The programs in Figures 1 and 2 almost immediately translate into implementations. The only complication is the management of arbitrary index sets. Many vector computers provide scatter-gather hardware and software to assist in this task. When such tools are not available, it is necessary to simulate them. Because it influences performance (only marginally on non-vector computers), we describe briefly our current implementation.

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. Only when the ranges become highly fragmented, does overhead become noticeable.

5. MULTICOMPUTER HARDWARE

Most experiments reported in this paper were performed on the Symult 2010. In this section, we give a brief description of its hardware, and we list performance information relevant to the interpretation of our computational experiments. For details we refer to [10].

The Symult 2010 is a multicomputer, and hence, it consists of a collection of independent computers connected by a communication network. The CPU of the Symult 2010 nodes is the Motorolla 68020. Each node has a 4 Mbyte memory. At the time of writing this paper, Caltech's Symult 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 [11] helped us to improve significantly the accuracy of these timings. From Table 2, we may conclude that the nodes should execute at a sustained performance level between 50 and 100 kFLOPS. We used the optimizing GNU C-compiler for all Symult 2010 experiments.

The topology of the communication network is a rectangular mesh. It features *worm hole routing*: a message header reserves all necessary channels between source and destination, and subsequently, the message is transmitted. The principal advantage is

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

Operation	Operations per second
i*=j	5.24659e+05
x+=y	4.64468e+05
x*=y	3.29924e+05
x=x+y*z	3.88350e+05
x[i]=x[i]+y*z[i]	7.71426e+04
x[i]=x[i]+y[i]*z[i]	6.95846e+04

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 node features a microprogrammed second processor to manage send and receive queues. Figure 4 displays the communication time as a function of the length of the message. These times were obtained by sending a message 1000 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 primitives provided by the Reactive Kernel/Cosmic Environment [12] (solid curve) and with our communication routines layered on top of the reactive primitives (dashed curve). This extra layer is necessary, because the reactive primitives do not provide any message selectivity. This shows that the overhead is negligible for all but the shortest messages.

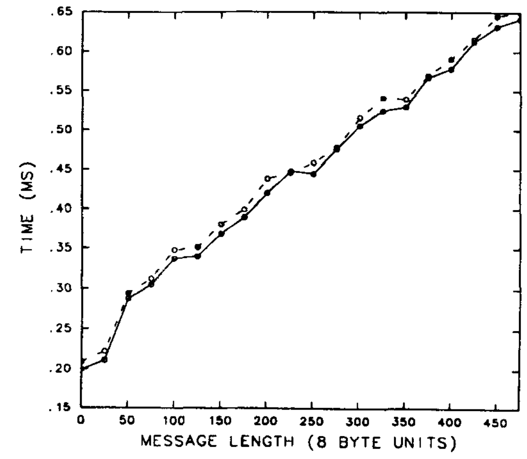


Figure 4. Symult 2010 communication time as a function of message length

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 4 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.

6. 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 section 7 establish the limits of the validity of this analysis.

We only deal with square matrices. 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, and 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 operating system.

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 terms 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. Summing the above for $k = 0, \dots, M-1$, and using equation (4), we obtain the following execution time for LU-decomposition:

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

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.

As noted by Fox *et al.* [6], equation (5) implies that the efficiency η of the calculation,

$$\eta = \frac{T_1}{PQT_{PQ}}$$

is—to leading order—only a function of the communication to computation ratio $\tau_C(L)/\tau_A$ (L is a characteristic message length) and the granularity $n = MN/PQ$ of the computation. The communication to computation ratio is determined solely by hardware characteristics. Thus, within the same family of multicomputers, the important parameter is the granularity n. For this reason, the efficiency of small problems on a small number of nodes accurately predicts the efficiency of larger problems on a larger number of nodes provided the granularity is kept equal. Typically, the efficiency increases as the granularity is increased.

In performance considerations for the LU-decomposition, the main effect of pivoting is load (im)balance. In the above best case analysis load balance was assumed a priori, and for this reason, the cost of pivoting was not included in equation (5). For row, column or diagonal pivoting, the pivot search cost is a low order load imbalance similar to that of the multiplier calculation. Complete, multirow or multicolumn pivoting is as load balanced as the elimination step, i.e., the amount of work in the pivot search is proportional to the sizes of the local feasible row and column sets. Hence, the pivot search cost cannot destroy the load balance. All pivoting strategies need $\log_2(PQ)$ communication steps for the recursive doubling procedure. Because the message length involved is $O(1)$, this adds only a low order term to the communication cost. The recursive doubling procedure forms a synchronization point between elimination steps, which prevents pipelining the elimination steps (this was tacitly assumed in the summation of the T^k_{PQ}). For specified data distributions, Geist and Romine [4] show that pipelining elimination steps is possible for row or column pivoting. On machines with large communication to computation ratio, this increases the efficiency.

7. SYMULT 2010 EXPERIMENTS

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. For example, 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. 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 restrictions. Because we did not want to introduce any artificial times (obtained through extrapolation of timings of small problems), we used the largest matrix that would fit on one node: a 300 by 300 double precision matrix. The efficiencies for problems of this size are easily surpassed by larger problems on comparable machines. As noted in section 6, the important parameter that determines efficiency to leading order is the granularity. As will be borne out by our experiments, excellent efficiencies are obtained with a granularity $n = 300^2/16 = 5625$. Similar efficiencies can be expected for problems of the same granularity on larger multicomputer configurations. Given the high efficiencies obtained with a modest value of the granularity, these experiments show great promise for massively concurrent computations.

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. Both our sequential and concurrent programs were optimized at an equal level. Neither implementation was geared for highest absolute performance, but rather for generality (i.e., data distribution and pivoting strategies). Table 3 displays the execution times and floating-point performance for two implementations of LU-decomposition with row pivoting. The first program is a fully optimized C-version of the LINPACK benchmark [13]. This implementation uses only a single index to access matrix entries, and it uses optimized BLAS-like procedures for the elementary vector operations. The second is an almost literal implementation of the program displayed in Figure 1. LU-decomposition 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. Because of the detailed optimizations of LINPACK, its sequential performance is close to peak performance, as follows from comparison with Table 2. Our implementation, without such detailed optimizations, runs at 78 per cent of LINPACK.

Because speed-up graphs depend on irrelevant implementation details of the sequential and concurrent programs, we give the observed execution times as a function of the

Table 3. Execution times and floating-point operations per second for two sequential LU-decomposition programs

Program	Execution time (ms)	FLOPS
LINPACK	1.21835e+05	7.387e+04
Implicit	1.56708e+05	5.743e+04

number of nodes. Speed-up graphs do have the advantage that ideal concurrency corresponds to a linear speed-up, i.e., $T_{PQ} = T_1/PQ$. Following a suggestion of Seitz [14], 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, the ideal speed-up curve is displayed as a solid straight line. For some computations, we list separately the speed-up and efficiency obtained between equally optimized sequential and concurrent programs.

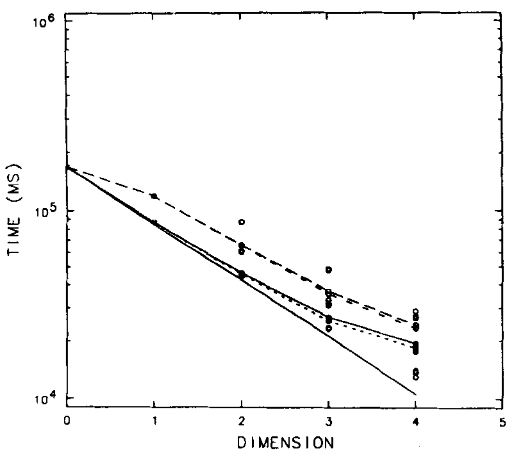
7.1. No pivoting

In Figure 5, 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 above, the solid straight line corresponds to ideal speed-up. The dashed curves in this and following figures connect times for one-dimensional distributions, i.e., $P \times 1$ or $1 \times Q$ process grids with either linear or scatter distribution. Points that correspond to two-dimensional distributions are not connected. The table in Figure 5 summarizes the characteristics of the least and most effective 16 node timings.

LU-decomposition without pivoting does not satisfy the assumptions of section 6. 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.

7.2. Row pivoting

The performance of LU-decomposition with row pivoting is displayed in Figure 6.



	No pivoting Minimum	Maximum
Speed-up	5.9	13.0
Efficiency	36.6%	81.3%
Process grid	4 × 4	4 × 4
Row distribution	linear	linear
Column distribution	linear	linear

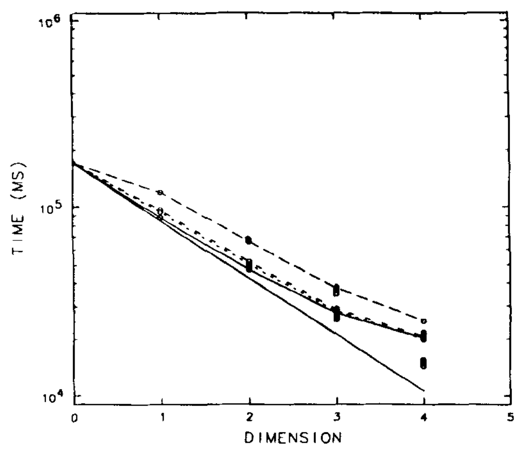
Figure 5. LU-decomposition without pivoting

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 per cent 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 per cent. 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.

7.3. Column pivoting

Figure 7 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 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



	Row pivoting Minimum	Maximum
Speed-up	6.8	11.8
Efficiency	42.4%	73.8%
Process grid	1 × 16	4 × 4
Row distribution		scatter
Column distribution	linear	scatter

Figure 6. LU-decomposition with row pivoting

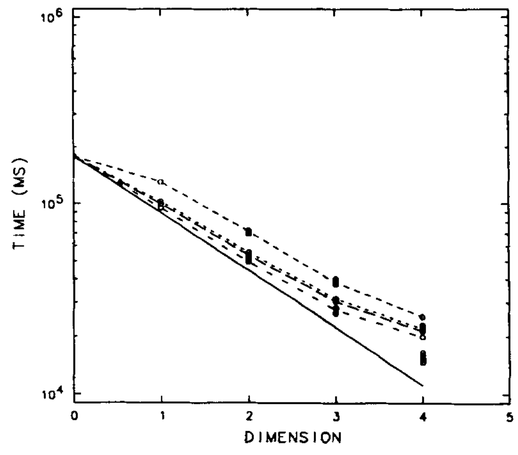
times in Figure 7 are obtained without carrying out this loop reversal because we wanted to vary as few parameters as possible during the experiment. To keep the speed-ups and efficiencies of the tables in Figures 6 and 7 comparable, we used the sequential row pivoting time as a standard. The minor difference in performance between column and row pivoting in favor of the latter is similarly due to the different sensitivity of the program to fragmentation of row and column index sets.

7.4. Diagonal pivoting

As in section 7.3, 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 (see Figure 8).

7.5. Complete pivoting

Complete pivoting guarantees numerical stability at a considerable expense. From Figure 9, 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



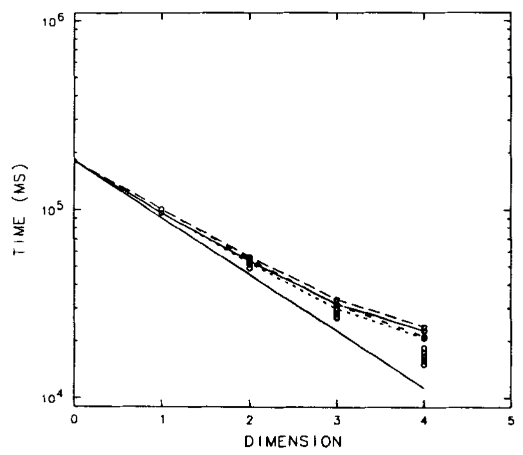
Column pivoting	Minimum	Maximum
Speed-up	6.7	11.5
Efficiency	42.0%	71.8%
Process grid	16 × 1	4 × 4
Row distribution	linear	scatter
Column distribution		scatter

Figure 7. LU-decomposition with column pivoting

range is much narrower than the previous two cases, 22.8 per cent. 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.

7.6. 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 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 10 displays the results. The efficiency range of 27.2 per cent 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 $Q = 1$.



Diagonal pivoting	Minimum	Maximum
Speed-up	7.1	11.3
Efficiency	44.6%	70.4%
Process grid	1 × 16	4 × 4
Row distribution		scatter
Column distribution	linear	scatter

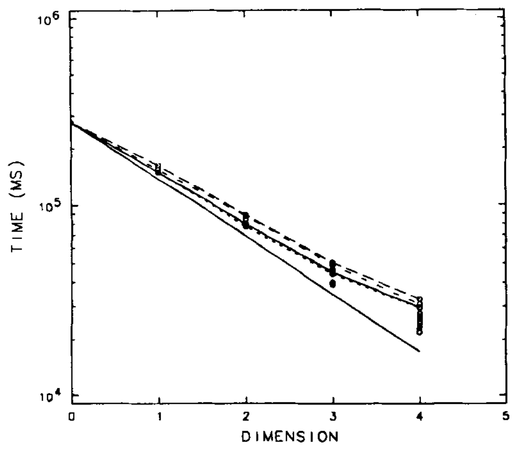
Figure 8. LU-decomposition with diagonal pivoting

7.7. Multicolumn pivoting

As in section 7.3, we use partial row pivoting as sequential standard. We obtain Figure 11. For both multirow and multicolumn pivoting, the one-dimensional process grids are the worst performers. A two-dimensional process grid is thus preferred.

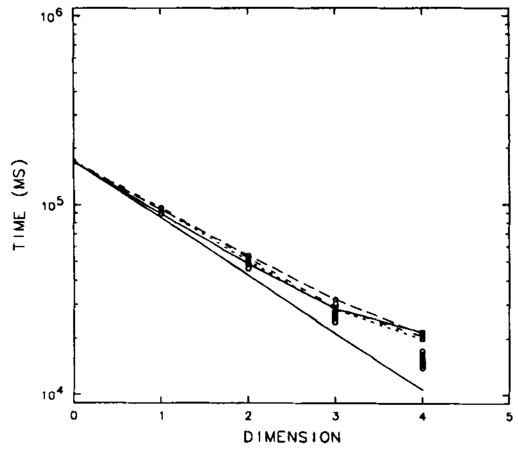
7.8. Randomized distributions

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. For example, 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.



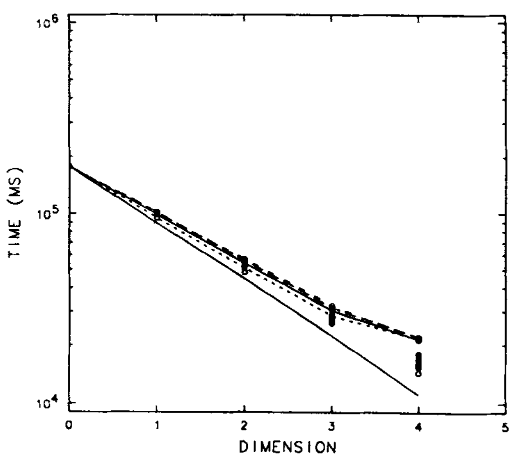
Complete pivoting	Minimum	Maximum
Speed-up	9.0	12.6
Efficiency	56.0%	78.8%
Process grid	1 × 16	4 × 4
Row distribution		scatter
Column distribution	linear	scatter

Figure 9. LU-decomposition with complete pivoting



Multirow pivoting	Minimum	Maximum
Speed-up	7.9	12.2
Efficiency	49.1%	76.3%
Process grid	1 × 16	4 × 4
Row distribution		scatter
Column distribution	linear	scatter

Figure 10. LU-decomposition with multirow pivoting



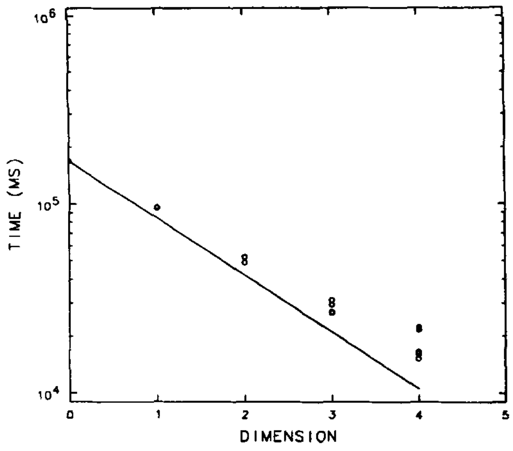
Multicolumn pivoting	Minimum	Maximum
Speed-up	7.7	11.8
Efficiency	48.3%	73.7%
Process grid	1 × 16	4 × 4
Row distribution		scatter
Column distribution	linear	scatter

Figure 11. LU-decomposition with multicolumn pivoting

Table 4. Generation of a pseudo-random distribution

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

In Figure 12, 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.



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

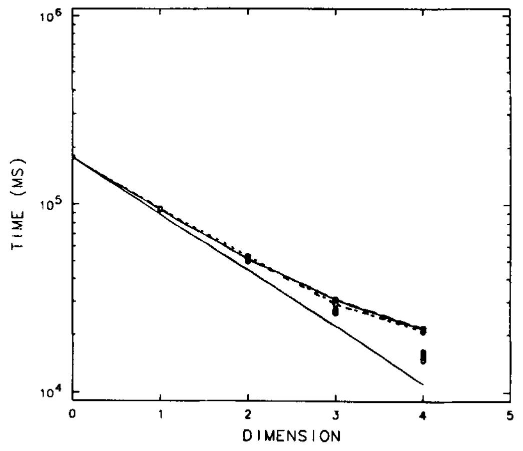
Figure 12. LU-decomposition without pivoting for pseudo-randomly distributed matrices

7.9. Preset random pivoting

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 13 summarizes the results.

7.10. 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 non-optimal, 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.



	Preset random pivoting Minimum	Maximum
Speed-up	8.2	12.0
Efficiency	51.1%	75.3%
Process grid	1 × 16	4 × 4
Row distribution	linear	linear
Column distribution	linear	linear

Figure 13. LU-decomposition with preset randomized pivoting

Table 5. Symult 2010 execution times in seconds for 16 Node LU-decomposition. In the first column, L stands for linear and S for scatter distribution

Distribution	No	Diagonal	Multirow	Multicolumn	Random
16 × 1 L-	23.309	21.152	20.590	21.251	21.336
16 × 1 S-	18.384	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

8. IPSC/2 EXPERIMENTS

The CPU of the iPSC/2 node is an Intel 80386 chip. Local memory is 4 Mbyte. Table 6 displays node performance in operations per second. By comparison with Table 2, the node performance for vector operations is higher on the iPSC/2 than on the Symult 2010. The communication network has a hypercube topology with worm hole routing. Figure 14, which displays the communication time as a function of the length of the message, can be compared with Figure 4. The Symult 2010 has better communication performance because of higher channel width and a more advanced implementation of node to node communication. The exchange of a message of 25 double precision numbers takes about 6.5 times longer on the iPSC/2 than on the Symult 2010.

Table 6. Number of operations per second for the iPSC/2 processor. The variables x, y, z, x[i], y[i], and z[i] represent double precision values, i,j, and k are integers

Operation	Operations per second
i*=j	1.06383e+07
x+=y	1.20250e+05
x*=y	1.05352e+05
x=x+y*z	1.11832e+05
x[i]=x[i]+y*z[i]	9.28074e+04
x[i]=x[i]+y[i]*z[i]	9.10913e+04

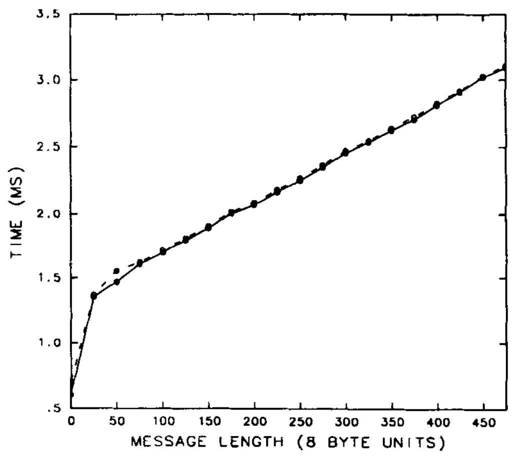


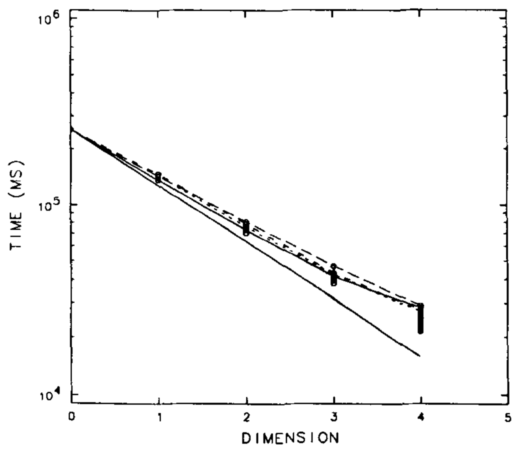
Figure 14. iPSC/2 communication time as a function of message length

This iPSC/2 advantage in node speed all but disappears in our application because the GNU optimizing compiler outperforms the Intel compiler. Table 7 gives the performance of the C-version of the LINPACK benchmark and the sequential implicit pivoting algorithm. LINPACK runs at about 76 per cent of peak (compared with 96 per cent on the Symult 2010). The non-optimized sequential algorithm works at about 50 per cent of LINPACK performance (compared to the 78 per cent on the Symult 2010).

Table 7. iPSC/2 execution times and floating-point operations per second for two sequential LU-decomposition programs

Program	Execution time (ms)	FLOPS
LINPACK	1.26817e+05	7.0968e+04
Implicit	2.53967e+05	3.5438e+04

In Figure 15, the performance of multirow pivoting on the iPSC/2 is displayed. In spite of the significant difference in computation to communication ratio, the speed-ups obtained on the two machines are nearly equal. This allows us to conclude that, for this problem of moderate granularity, communication is not a major issue.



Multirow pivoting	Minimum	Maximum
Speed-up	8.76	11.88
Efficiency	54.7%	74.3%
Process grid	1 × 16	4 × 4
Row distribution	linear	scatter
Column distribution	linear	scatter

Figure 15. LU-decomposition with multirow pivoting, iPSC/2 experiment

Table 8 compares no, multirow and random pivoting on the iPSC/2. As for the Symult 2010, multirow pivoting is always close to optimal or optimal.

9. 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

Table 8. iPSC/2 execution times in seconds for 16 node LU-decomposition. In the first column, L stands for linear and S for scatter distribution

Distribution	No	Multirow	Random
16 × 1 L-	31.468	27.534	28.706
16 × 1 S-	23.947	26.895	27.941
8 × 2 L-L	40.054	24.128	24.073
8 × 2 L-S	28.395	23.519	24.538
8 × 2 S-L	26.501	23.655	23.714
8 × 2 S-S	20.378	21.749	24.185
4 × 4 L-L	43.891	26.388	23.175
4 × 4 L-S	27.816	22.833	23.054
4 × 4 S-L	27.224	23.401	23.127
4 × 4 S-S	19.817	21.370	22.992
2 × 8 L-L	40.613	25.898	24.151
2 × 8 L-S	27.381	24.269	23.924
2 × 8 S-L	28.323	24.926	23.934
2 × 8 S-S	20.693	22.298	23.696
1 × 16 -L	32.209	28.997	28.373
1 × 16 -S	24.799	28.265	28.463

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. For all distributions, multirow pivoting always performs at or near optimal load balance.

In a practical context, our LU-decomposition program can be used in two ways. Either, the matrix distribution is considered as given and the LU-decomposition is applied to it irrespective of performance considerations, or it can be used to find the best possible distribution for a particular program. Thus, a global optimization of the data distribution is feasible if data distribution independent components are used. Even global optimization strategies that include dynamically changing distributions may be considered.

ACKNOWLEDGEMENTS

The lectures of K. Mani Chandy, based on [15], helped the author to look at concurrent programs from a fresh viewpoint. The help of Wen-King Su and Jakov Seizovic with the details of the Cosmic Environment and Reactive Kernel is also much appreciated.

This research is supported in part by Department of Energy Grant 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 U.S. government has certain rights in this material.

REFERENCES

1. R.M. Chamberlain, 'An alternative view of LU factorization with partial pivoting on a hypercube multiprocessor', in *Hypercube Processors 1987*, M.T. Heath, ed., SIAM Publications, Philadelphia, PA, 1987, pp. 569–575.

2. E. Chu and J.A. George, 'Gaussian elimination with partial pivoting and load balancing on a multiprocessor', *Parallel Computing*, 5, 65–74 (1987).
3. G.A. Geist and M.T. Heath, 'Matrix factorization on a hypercube multiprocessor', in *Hypercube Processors 1986*, M.T. Heath, ed., SIAM Publications, Philadelphia, PA, 1986, pp. 161–180.
4. G.A. Geist and C.H. Romine, 'LU factorization algorithms on distributed-memory multiprocessor architectures', *SIAM Journal on Scientific and Statistical Computing*, 9(4), 639–649 (1988).
5. C.B. Moler, 'Matrix computation on a hypercube multiprocessor, in *Hypercube Proces sors 1986*, M.T. Heath, ed., SIAM Publications, Philadelphia, PA, 1986, pp. 181–195.
6. 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.
7. P.G. Hipes and A. Kupperman, 'Gauss–Jordan inversion with pivoting on the Caltech Mark II hypercube', in *Hypercube Concurrent Computers and Applications*, G.C. Fox, ed., ACM Press, New York, NY, 1988, pp. 1621–1634.
8. E.F. Van de Velde, 'The formal correctness of an LU-decomposition algorithm', *Technical report C3P-625*, Caltech Concurrent Computation Project, 1988.
9. H.S. Stone, *High Performance Computer Architecture*, Addison-Wesley, 1987.
10. 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 *Hypercube Concurrent Computers and Applications*, G.C. Fox, ed., ACM Press, New York, NY, 1988, pp. 33–36.
11. K. Kennedy, Private communication, Rice University.
12. C.L. Seitz, J. Seizovic and W.-K. Su, 'The C programmer's abbreviated guide to multicomputer programming', *Technical report CS 5252:TR:87*, California Institute of Technology, 1987.
13. B. Toy, Private communication, SUN Microsystems.
14. C.L. Seitz, Private communication, California Institute of Technology.
15. K.M. Chandy and J. Misra, *Parallel Program Design, A Foundation*, Addison Wesley, 1988.