Multicomputer Matrix Computations:
Theory and Practice

Eric F. Van de Velde

CRPC-TR89023
March, 1989

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

Multicomputer Matrix Computations: Theory and Practice *

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

March 3, 1989

Abstract

We explore a strategy for concurrent program development. It is significant that just two basic techniques, data distribution and recursive doubling, are sufficient for the development of many linear algebra computations.

When constructing a concurrent library, it is counterproductive to develop "optimal algorithms" for each component. The problem is that optimality is not stable under program composition. A user program based on individually optimized components may be very inefficient due to interfaces between components. Invariably, these interfaces must perform some type of data redistribution. We implement our algorithms such that their correctness is, to a large extent, independent of data distribution. This allows the user to chose the most convenient and/or the most efficient distribution for a particular application and/or hardware. Another benefit is that the need for interfaces is minimized.

We applied the above ideas in the development of a multicomputer library for linear algebra computations. The library includes most basic operations and a representative selection of advanced algorithms for full and sparse matrices. We present some timing results for the Ametek multicomputer.

1 Introduction

In this paper, we summarize the results presented more extensively in [13] and [14]. Our goal is to develop an efficient linear algebra library for multicomputers. Specific design criteria are that the library components must be easy to integrate into a larger user program. For this reason, it is crucial to keep the correctness of our algorithms independent of specific data distributions. We also do not accept any artificial restrictions. An typical example of such restrictions is that the dimension of a matrix be a multiple of the number of processors. An advanced application of this software is described in [9]. In Section 2, we transform an easy, sequential sample program into multicomputer form. This same technique can be applied to more complicated algorithms like LU- and QR-decomposition. In Sections 3 and 4, we describe the results of such transformations for these more complicated algorithms. We shall also give some performance results on the Ametek 2010 multicomputer.

We shall restrict our attention to multicomputer programs. Because the latest multicomputer operating systems, like the Cosmic Environment/Reactive Kernel of Seitz et al. [11], allow running several processes per node, we shall refer to a computation with P processes rather than to a computation with P nodes. Whether the processes are run on P different nodes is not important as far as the correctness of the multicomputer program is concerned. It plays an important role, however, in the efficiency of the program execution. For timing results, we restrict ourselves to the case where each process is run on a dedicated processor.

2 The Inner Product

The inner product operation provides an easy example program, in which the relevant ideas can be expressed without technical complications. More complicated algorithms are transformed into multicomputer form with the same technique.

The inner product operation is easily specified by:

$$\sigma := \langle + m \ : \ 0 \leq m < M \ :: \ x[m]y[m] \rangle$$

In this section, we use the UNITY notation of Chandy and Misra [3], which provides an unambiguous framework to express algorithms. The above *quantification* specifies that the variable σ is the sum of all products $x[m]y[m]$ for $0 \leq m < M$.

This specification is transformed trivially into a concurrent program by duplicating all variables and all work in every process:

$$\begin{array}{l} \langle \parallel \ p \ : \ 0 \leq p < P \ :: \\ \quad \sigma[p] \ := \langle + m \ : \ 0 \leq m < M \ :: \ x[p][m]y[p][m] \rangle \\ \rangle \end{array}$$

Ignoring technical complications that need not concern us here, the $\|$–separator signifies the concurrency of the operations inside the quantification. The inner quantification can be considered a program for process p.

Obviously, the duplication of effort must be removed. We assign the responsibility of summing a subset of the quantities to each process. For this reason, the index set

*This research is supported in part by Department of Energy Grants Nos. DE-FG03-85ER25009 and DE-AS03-76ER72012 and by the NSF Center for Research on Parallel Computing at Caltech.

$\mathcal{M} = \{m : 0 \leq m < M\}$ is partitioned into P nonintersecting subsets $\mathcal{M}_p$, where $0 \leq p < P$. Process p sums all products $x[m]y[m]$, where $m \in \mathcal{M}_p$. In all $\sigma[p]$, we must obtain the sum of all such partial sums. It is well known that this is best achieved by ***recursive doubling***, see e.g. [6] and [12]. This leads to the program:

$$
\begin{array}{l}
\langle \parallel p : 0 \leq p < P :: \\
\quad \sigma[p] := \langle + m : m \in \mathcal{M}_p :: x[p][m]y[p][m] \rangle ; \\
\quad \langle ; d : 0 \leq d < \log_2 P :: \\
\quad\quad \textbf{send } \sigma[p] \textbf{ to } p\bar{\vee}2^d ; \\
\quad\quad \textbf{receive } t[p] \textbf{ from } p\bar{\vee}2^d ; \\
\quad\quad \sigma[p] := \sigma[p] + t[p] \\
\quad \rangle \\
\rangle
\end{array}
$$

The expression $p\bar{\vee}2^d$ is the integer obtained by performing an exclusive or operation on the binary expansions of p and 2^d. As a result, $p\bar{\vee}2^d$ flips bit number d of p.

The components of the arrays $x[p]$ and $y[p]$ with indices not in $\mathcal{M}_p$ are never accessed in process p. For this reason, the vectors are compacted. Instead of the global index m, we use a local index i. The correspondence between m and i is made with a distribution function μ that maps a global index m into a process number p and a local index i, i.e.:

$$\mu(m) = (p, i).$$

The local indices in process p are elements of a local index set, say $\mathcal{I}_p$. The multicomputer form of the inner product program is then obtained by changing the first assignment into:

$$\sigma[p] := \langle + i : i \in \mathcal{I}_p :: x[p][i]y[p][i] \rangle ;$$

The resulting program for process p is given in a more conventional notation by:

$$
\begin{array}{l}
\sigma := 0.0 ; \\
\textbf{for } i \in \mathcal{I}_p \textbf{ do } \sigma := \sigma + x[i]y[i] ; \\
\textbf{for } d = 0, 1, \ldots, \log_2 P - 1 \textbf{ do begin} \\
\quad \textbf{send } \sigma \textbf{ to } p\bar{\vee}2^d ; \\
\quad \textbf{receive } t \textbf{ from } p\bar{\vee}2^d ; \\
\quad \sigma := \sigma + t \\
\textbf{end}
\end{array}
$$

3 LU-Decomposition

Our concurrent LU-decomposition program is a component of a library. As such, it should be easy to integrate into a user program. Several criteria have to be met. First, it should be correct for as large a class of matrix distributions as possible. This is important because the particular matrix distribution may have a major impact on the convenience and/or efficiency of initializing the matrix. The latter component must be written by the user and may be a complex computation in its own right. Second, for as wide a range of distributions as possible, the LU-decomposition should be efficient. Ideally, the efficiency of the computation is independent of the data distribution. While the latter goal is unrealistic, we wish to come as near to it as possible.

A major component of LU-decomposition is the pivoting strategy. For full matrix computations, pivoting is used for numerical stability. The strategy of choice has long been row pivoting. For sparse matrices, pivoting is also used to reduce the creation of fill. In [13], we show that pivoting can also be used to randomize the computation and increase the concurrent efficiency. Because pivoting plays such an important role, we allow maximum flexibility in the choice of pivots. Our algorithm combines the strategies of Chamberlain [2], Chu and George [5], Geist and Heath [7], and Moler [10], who have examined varying combinations of row- and column-oriented distributions with row and column pivoting. It also includes rectangular decompositions, such as those used by Fox et al. [6], Hipes and Kupperman [8]. The latter are based on the observation that rectangular distributions minimize the communication cost.

Our LU-decomposition algorithm is based on implicit row and column pivoting. If in the course of the LU-decomposition no zero pivot is encountered, an implicit pivoting strategy factors an $M \times M$ matrix A in the form:

$$A = LC^T RU,$$

where R is a row permutation, C a column permutation, RLC^T is unit lower triangular and RUC^T is upper triangular. This is a variant of explicit pivoting strategies that factor a matrix in the form:

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

In the k–th elimination step of the implicit pivoting LU-decomposition algorithm, a pivot is chosen among the feasible entries of the matrix. Its row and column are made infeasible. The multipliers are calculated in the feasible rows of the multiplier column. The appropriate product of a multiplier and a pivot row entry is subtracted from the remaining feasible entries. The LU-decomposition program is given by:

$$
\begin{array}{l}
\mathcal{M} := \{m : 0 \leq m < M\} ; \\
\mathcal{N} := \{n : 0 \leq n < N\} ; \\
\textbf{for } k = 0, 1, \ldots, \min(M, N) - 1 \textbf{ do begin} \\
\quad \{\texttt{Pivot Strategy and Bookkeeping.}\} \\
\quad \text{do pivot search and find } a_{rc}, r[k], c[k] ; \\
\quad \mathcal{M} := \mathcal{M} \setminus \{r[k]\} ; \\
\quad \mathcal{N} := \mathcal{N} \setminus \{c[k]\} ; \\
\quad \textbf{if } a_{rc} = 0.0 \textbf{ then terminate} ; \\
\\
\quad \{\texttt{Calculation of the Multiplier Column.}\} \\
\quad \textbf{for all } m \in \mathcal{M} \textbf{ do} \\
\quad\quad a[m, c[k]] := a[m, c[k]]/a_{rc} ; \\
\\
\quad \{\texttt{Elimination.}\} \\
\quad \textbf{for all } (m, n) \in \mathcal{M} \times \mathcal{N} \textbf{ do} \\
\quad\quad a[m, n] := a[m, n] - a[m, c[k]]a[r[k], n] \\
\textbf{end}
\end{array}
$$

In this program, the index sets $\mathcal{M}$ and $\mathcal{N}$ keep track of the feasible rows and columns. The variable a_{rc} has the current pivot value and $r[k]$ and $c[k]$ are the row and column index of the k–th pivot.

By applying the technique discussed in Section 2, the above program is transformed into a multicomputer program. The processes in the resulting program are configured as a $P \times Q$ grid. The rows and columns of the matrix are distributed over the process rows and columns according to distribution functions μ and ν. The concurrent LU-decomposition program for process (p, q) has the form:

$\mathcal{I} := \{i : 0 \leq i < I\}$;
$\mathcal{J} := \{j : 0 \leq j < J\}$;
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**
      $a_c[i] := a[i, \hat{\jmath}] := a[i, \hat{\jmath}]/a_{rc}$;
    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

The major changes of the concurrent program with respect to the sequential one are: the conversion from global to local indexation, the broadcast of the pivot row and the broadcast of the multiplier column. The broadcasts are nothing more than recursive doubling procedures, although this is hidden by the notation.

The above program must be completed with a pivoting strategy. Because this LU-decomposition does not put any a priori restrictions on the pivot choice, most classical pivoting strategies are easily incorporated. We have used complete, row, column, and diagonal pivoting. In addition, we have used two intrinsically concurrent strategies, which we call *multirow* and *multicolumn* pivoting.

Multirow pivoting is based on the observation that with partial row pivoting only one process column is active during the pivot search. Without any overhead, every process column can search through one additional arbitrary matrix column. This increases the extent of the pivot search and usually yields increased numerical stability. In [13], we show that multirow pivoting also leads to increased performance. The extra degree of freedom in the computation, i.e., the column index of the pivot, results in an increased randomization of the distribution of the feasible

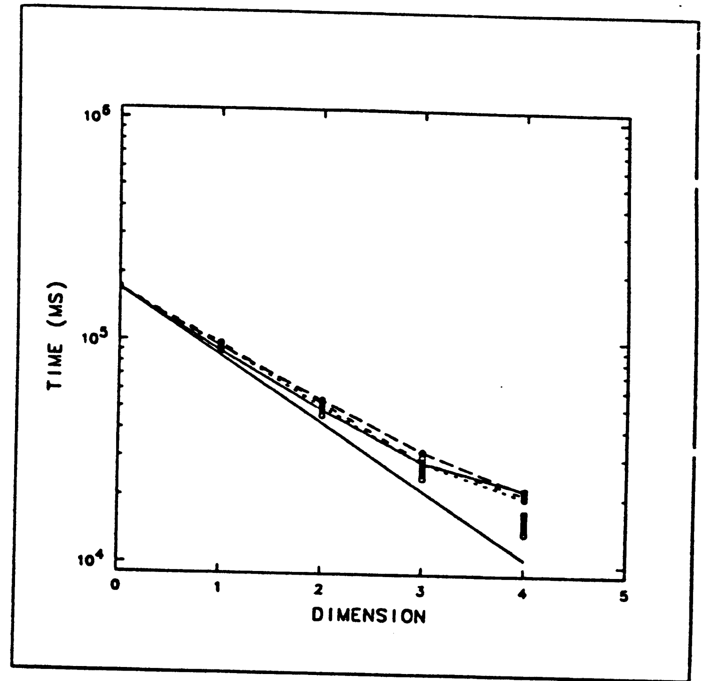

Figure 1: LU-Decomposition With Multirow Pivoting.

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

Table 1: Least and the Most Effective 16 Node LU-Decomposition with Multirow Pivoting.

entries over the processes. This in turn results in a higher load balance of the computation.

We refer to [13] for a complete experimental study of the performance of this program and of its sparse matrix versions. Here, we display the performance on the Ametek 2010 for multirow pivoting. The execution times for the LU-decomposition of a 300×300 matrix as a function of the number of independent processes is displayed in Figure 1. Because we use a log-log plot, the ideal speed up curve is a straight line. For each machine size, several matrix distributions are timed. In Table 1, we summarize the best and worst 16-node performance. Even for this small problem, a speed up of 12.2 is obtained on a 16 node system. For many distributions multirow pivoting is not only numerically more stable, it is also faster than no pivoting, i.e., the search cost is more than offset by the increased load balance.

4 QR-Decomposition

QR-decomposition should satisfy the same criteria as LU-decomposition: it should be correct for a range of data distributions and it should be efficient for as large a set of

distributions as possible. From a numerical perspective, pivoting is not as crucial for QR-decomposition as for LU-decomposition. Only column pivoting is used and then only in the case of underdetermined systems, see [1]. Considering the positive effect of pivoting on the performance of LU-decomposition, it is reasonable to pursue more general pivoting strategies. Here, we shall restrict ourselves to the algorithm without any pivoting. The major point that we wish to make is that data distribution and recursive doubling are the only concepts needed to implement this algorithm. A multicomputer QR-decomposition was developed by Chu [4]. We shall show that the complicated logic and extensive bookkeeping of this program can be avoided. The concept of recursive doubling is sufficient. Our program is valid for matrix distributions as discussed in Sections 2 and 3.

The QR-decomposition may be based on either Householder reflections or Givens rotations. It is well known that for full matrices, the former are more efficient. The Givens algorithm, however, can be generalized to sparse matrices. We consider only the latter and restrict our discussion to full matrices. For a more general discussion, see [14].

We assume the theory of Givens rotations known. Let Givens$(i, j, \vec{x})$ be the Givens rotation that, when applied to $\vec{x}$, replaces component i of $\vec{x}$ by $\sqrt{\xi_i^2 + \xi_j^2}$ and annihilates component j. In the k-th step of the QR-decomposition of an $M \times N$ matrix A, components $k+1$ through $M-1$ of the k-th column vector are annihilated. To do this, the matrix A is pre-multiplied with Givens rotations of the form: Givens$(i, j, \vec{a}_k)$, where $j > i \geq k$. Such Givens rotations replace rows i and j by linear combinations of rows i and j. Because $j > i \geq k$, these linear combinations do not affect previously annihilated entries.

The multicomputer implementation of the QR-decomposition algorithm exploits a number of key properties. The applicability of the recursive doubling technique follows from the associativity of the Givens operator. In contrast to the addition operator, it is not commutative. We shall also use that two Givens rotations Givens$(i, j, \vec{a}_k)$ and Givens$(i', j', \vec{a}_k)$ may be computed and applied concurrently provided $i \neq i'$ and $j \neq j'$. The QR-decomposition is essentially unique (only sign changes occur). The order in which entries of the k-th column are annihilated is not important, although different orderings use different elementary rotations.

Consider an $M \times N$ matrix A distributed over a $P \times Q$ process grid with row and column distributions μ and ν. As in the LU-decomposition algorithm without pivoting, row k and column k are made infeasible in step k of the QR-decomposition algorithm. The k-th step annihilates the entries in the feasible rows of column k. The distribution functions μ and ν determine the pivot process $(\hat{p}, \hat{q})$ and the local indices $(\hat{\imath}, \hat{\jmath})$ of the current pivot entry (k, k). In each process (p, q) the local feasible row and column sets $\mathcal{I}_p$ and $\mathcal{J}_q$ are also known.

All Givens rotations are determined by the values of the entries of the k-th column, the pivot column. Hence it is necessary to broadcast the pivot column from process column $\hat{q}$ to all other process columns. This is the only

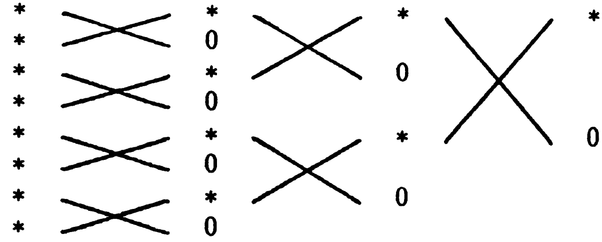

Figure 2: Recursive Doubling Procedure to Annihilate the Pivot Column.

complication of a general rectangular distribution. For the remainder of the discussion, we shall limit our comments to row distributed matrices, i.e., we shall refer to process p instead of to process row p. The final algorithm is formulated for general rectangular distributions, however.

The k-th step is started by choosing a local pivot row, say local row ℓ, in every process p. In process $\hat{p}$, the local pivot row is identical to the global pivot row, i.e., $\ell = \hat{\imath}$. In a computation that is entirely local to each process, the remaining local entries in the pivot column, i.e., those entries with a local row index $i \in \mathcal{I}_p \setminus \{\ell\}$ are annihilated. This can always be done with Givens rotations between rows ℓ and i. If blocks of consecutive rows are allocated to the same process (linear distribution), more efficient Householder transformations can be used. We do not consider this possibility further.

Cooperation between processes is needed to annihilate the leading entries of local pivot rows. We shall show that this may be achieved by a recursive doubling procedure. For now let us assume that every process p has a local pivot row (this is not the case if the set $\mathcal{I}_p$ is empty) and that $\hat{p} = 0$. We shall deal with complications that arise when these assumptions are not satisfied in subsequent paragraphs. In Figure 2, we display graphically a recursive doubling procedure to annihilate all leading entries but one among the local pivot rows. In the figure, an $*$ stands for a noneliminated row and a 0 for a row with zero entries in columns 0 through k. In this procedure, half of the processes become inactive at every step or, alternatively, the Givens rotations they compute and apply are identity operators. [1] The multicomputer program that implements the recursive doubling procedure displayed in Figure 2 is readily obtained. With $a[\ell]$ the local pivot row of the matrix A in process p and t a temporary array, the program for process p is given by:

```
for d = 0, 1, ..., log₂ P − 1 do begin
    send {a[ℓ, n] : 0 ≤ n < N} to p∨̄2^d ;
    receive {t[n] : 0 ≤ n < N} from p∨̄2^d ;
    if p∧2^d = 0 then
        modify row a[ℓ] ;
    else
        eliminate row a[ℓ] ;
end
```

In this program, *eliminate row* $a[\ell]$ means that the leading entry of row $a[\ell]$ is annihilated and *modify row* $a[\ell]$ means

[1] When $Q = 1$ this can be avoided by starting the annihilation of the next column.

that the leading entry of row t is annihilated by the Givens rotation. (Of course, only computations that modify row $a[\ell]$ are carried out in this process.)

A problem with the proposed procedure is that the noneliminated row always ends up in process 0. To make this algorithm practical, we must have the capability to get the final noneliminated row to process $\hat{p}$. A key difference between this recursive doubling procedure and the one in Section 2 is that the elementary operator, the Givens rotation, is not commutative. This is exhibited in the program by the test:

if $p \wedge 2^d = 0$ **then**

For the final noneliminated row to end up in process $\hat{p}$, this test must be modified such that process $\hat{p}$ behaves like process 0. This is achieved by replacing p in this test by a permutation PERM(p), such that PERM($\hat{p}$) = 0. Any permutation of the process numbers $0, 1, \ldots, P-1$ will do. A handy choice is: $p \bar{\vee} \hat{p}$ (the exclusive or of the binary representation of the two integers p and $\hat{p}$). Thus, we only need to replace the test by:

if $(p \bar{\vee} \hat{p}) \wedge 2^d = 0$ **then**

If the local feasible row set $\mathcal{I}_p$ for some process p was empty, this process cannot participate in the recursive doubling procedure. This is easily remedied by introducing a dummy row of zeroes for this process. While solving one problem, it creates another. It is easily seen that the only possible Givens rotations between a zero and a nonzero row are the identity and the permutation operator. The latter may have the effect that real data ends up in a dummy row. Such an occurrence is easily detected, however. This happens if and only if the following three conditions hold:

1. there is a rotation between a dummy and a real row
2. the real row has a leading nonzero entry
3. annihilation of the leading nonzero entry is required.

It suffices to detect such permutations and undo them in the reverse order in which they occurred. We note that this undoing is itself a recursive doubling procedure. Our experience is that the occurrence of such dummy exchanges is very rare and do not have any impact on the performance of the program.

Combining the key components discussed so far, the following multicomputer QR-decomposition program is obtained. Process (p, q) is driven by the program:

$\mathcal{I} := \{i : 0 \le i < I\}$;
$\mathcal{J} := \{j : 0 \le j < J\}$;
for $d = 0, 1, \ldots, \log_2 P - 1$ **do** undo[d] := NO ;

for $k = 0, 1, \ldots, \min(M, N) - 1$ **do begin**
  `{ Set up Local and Global Pivot Indices }`
  $\hat{p}, \hat{\imath} := \mu(k)$;
  $\hat{q}, \hat{\jmath} := \nu(k)$;
  if $p = \hat{p}$ **then begin**
    $\mathcal{I} := \mathcal{I} \setminus \{\hat{\imath}\}$; $\ell := \hat{\imath}$
  end else $\ell :=$ any element from $\mathcal{I}$;
  if $q = \hat{q}$ **then** $\mathcal{J} := \mathcal{J} \setminus \{\hat{\jmath}\}$;

  `{ Broadcast Pivot Column }`
  if $q = \hat{q}$ **then begin**
    for all $i \in \mathcal{I}$ **do** $a_c[m] := a[m, k]$;
    send $a_c[i : i \in \mathcal{I}]$ **to** $(p, \bullet)$
  end else receive $a_c[i : i \in \mathcal{I}]$ **from** $(p, \hat{q})$;

  `{ Local Elimination }`
  for $i \in \mathcal{I} \setminus \{\ell\}$ **do begin**
    compute the Givens rotation ;
    modify row $a[\ell]$;
    eliminate row $a[i]$
  end ;

  `{ Global Elimination }`
  `{ Initialize Local Pivot Rows }`
  null0 := $(\mathcal{M} = \emptyset)$;
  if null0 **then begin**
    $a0_{rc} := 0.0$; **for** $j \in \mathcal{J}$ **do** $a0_r[j] := 0.0$
  end else begin
    $a0_{rc} := a_c[\ell]$; **for** $j \in \mathcal{J}$ **do** $a0_r[j] := a[\ell, j]$
  end ;
  `{ Recursive Doubling }`
  for $d = 0, 1, \ldots, \log_2 P - 1$ **do begin**
    send null0, $a0_{rc}, a0_r[j : j \in \mathcal{J}]$ **to** $(p \bar{\vee} 2^d, q)$;
    receive null1, $a1_{rc}, a1_r[j : j \in \mathcal{J}]$ **from** $(p \bar{\vee} 2^d, q)$;
    compute the Givens rotation ;
    if $(p \bar{\vee} \hat{p}) \wedge 2^d = 0$ **then begin**
      undo[d] := (null0 **and** $a1_{rc} \neq 0.0$) ;
      modify row $a0$
    end else begin
      undo[d] := (null1 **and** $a0_{rc} \neq 0.0$) ;
      eliminate row $a0$
    end
  end ;
  `{ Undo the Dummy Exchanges }`
  for $d = \log_2 P - 1, \log_2 P - 2, \ldots, 1, 0$ **do**
    if undo[d] **then begin**
      undo[d] := NO ;
      send $a0_{rc}, a0_r[j : j \in \mathcal{J}]$ **to** $(p \bar{\vee} 2^d, q)$;
      receive $a0_{rc}, a0_r[j : j \in \mathcal{J}]$ **from** $(p \bar{\vee} 2^d, q)$
    end ;
  `{ Return Local Pivot Rows to Matrix }`
  if not null0 **then begin**
    for $j \in \mathcal{J}$ **do** $a[\ell, j] := a0_r[j]$;
    if $p = \hat{p}$ **and** $q = \hat{q}$ **then** $a[\ell, \hat{\jmath}] := a0_{rc}$
  end
end

In Figure 3, we display the times obtained for the QR-decomposition of 300 × 300 matrix on the Ametek 2010 multicomputer. The best sequential time (i.e., dimension=0) is obtained with Householder QR-decomposition. The other two sequential times are for the Givens algorithm: one with a straight sequential code and one with the multicomputer code run sequentially. As in Section 3, the times are displayed in a log-log plot such that the ideal speed up curve is a straight line. Table 2 summarizes the best and worst performance on the 16 node system. Speed ups and efficiencies are computed with respect to the se-

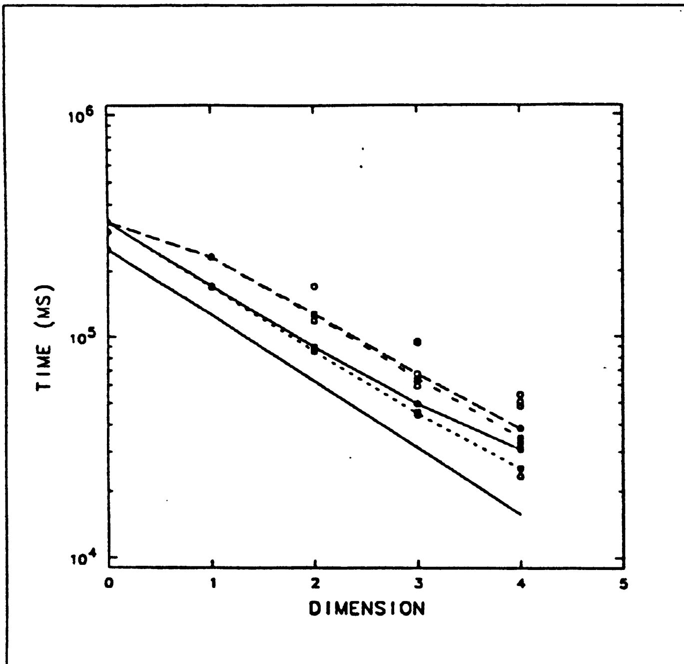

Figure 3: QR-Decomposition of a 300 × 300 Full Matrix on the Ametek 2010.

Givens QR-Decomposition		
	Minimum	Maximum
Speed Up	5.5	12.9
Efficiency	34.5%	80.8%
Process Grid	4 × 4	8 × 2
Row Distribution	linear	scatter
Column Distribution	linear	scatter

Table 2: Least and the Most Effective 16 Node QR-Decomposition.

quential Givens algorithm.

5 Conclusions

Many linear algebra algorithms on multicomputers are based on two straightforward concepts: data distribution and recursive doubling. The former determines to a large extent whether a particular algorithm is easily incorporated into a user program. For this reason, we have developed our linear algebra library such that it is as much as possible independent of particular data distributions. Recursive doubling is a key component algorithm to obtain efficient implementations.

References

[1] P. A. Businger and G. H. Golub. Linear least squares solutions by Householder transformations. *Numerische Mathematik*, 7:269–276, 1965.

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

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

[4] E. Chu. *Orthogonal Decomposition of Dense and Sparse Matrices on Multiprocessors.* PhD thesis, University of Waterloo, Waterloo, Ontario, 1988.

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

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

[8] 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.

[9] J. Lorenz and E. F. Van de Velde. Concurrent computations of invariant manifolds. March 1989. Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications.

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

[11] C. L. Seitz, J. Seizovic, and W.-K. Su. *The C Programmer's Abbreviated Guide to Multicomputer Programming.* report CS-TR-88-1, California Institute of Technology, 1987.

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

[13] E. F. Van de Velde. *Experiments with Multicomputer LU-Decomposition.* report C3P-725, California Institute of Technology, 1989.

[14] E. F. Van de Velde. Implementation of linear algebra computations on multicomputers. In Preparation.