High Performance Scalable Matrix Algebra Algorithms for Distributed Memory Architectures

Geoffrey C. Fox
A. Gaber Mohamed Nangkang Yeh
Gregory von Laszewski
Manish Parashar Neng-Tan Lin

CRPC-TR92210
April, 1992

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

High Performance Scalable Matrix Algebra Algorithms for Distributed Memory Architectures

Geoffrey C. Fox, A. Gaber Mohamed
Gregor von Laszewski, Manish Parashar
Neng-Tan Lin, Nangkang Yeh
agm@npac.syr.edu

Contents

High Performance Scalable Matrix Algebra Algorithms for Distributed Memory Architectures

Geoffrey C. Fox, A. Gaber Mohamed
Gregor von Laszewski, Manish Parashar
Neng-Tan Lin, Nangkang Yeh
agm@npac.syr.edu

Abstract

Our experimental results showed that block based algorithms for numerically intensive applications are superior to their noblock counterpart[12]. It is desirable to parallelize block based algorithms on distributed memory MIMD architectures since many scientific and engineering applications make use of these algorithms. Our goal is to optimize sample applications from LAPACK, develop them in Fortran 77D and Fortran 90D, and have them available as a scalable compiler library. In the presented study, we show ways to parallelize sequential block algorithms for the LU factorization. The goal of this paper is twofold.

On one hand, since this algorithms are difficult to parallelize they will be included in a benchmarking suite for the Fortran 90D project [6]. We point out problems inherent in the sequential nature of the block based algorithms. We learn that it is not intuitively clear which algorithm might perform best on a distributed memory architecture. The problems described here will help to improve the design of a source to source code compiler applied to numerically intensive applications.

Beside this conclusions, experiments done on the iPSC Hypercube show which parallel block algorithm should be used depending on the number of available processors, the matrix size, and the block size. Three algorithms for the column oriented Fortran are compared.

Keywords: LU factorization, blocked LU factorization, iPSC Hypercube, BLAS 3.

1 Introduction

Solutions of a system of linear equations are required in many scientific applications. [13, 5, 14, 15]. Consider the solution of the dense system of linear equations,

$$A\vec{x} = \vec{b}, \tag{1}$$

where A is an n-by-n matrix and $\vec{b}$ is a vector of dimension n and $\vec{x}$ is the solution vector of dimension n. One method of solving this problem is to proceed by first factorizing A into a unit lower triangular matrix L and an upper triangular matrix U, *i.e.*,

$$A = LU, \tag{2}$$

and then solving for $\vec{y}$ and $\vec{x}$ in two consecutive substitution steps:

$$L\vec{y} = \vec{b} \quad \text{and} \quad U\vec{x} = \vec{y}. \tag{3}$$

Experimental results show that in programs which need to solve a linear equation, more than 50% of the CPU time is usually spent in matrix factorization. This occurs because

1. the computational effort to factorize the matrix A is higher than for the two substitution steps.

2. most standard programming practices used to factorize the matrix result in more memory accesses than floating point operations. This cause the processor to be idle during the time data is transferred from memory for the computation.

The first observation motivates *why* it is desirable to build a fast LU factorization algorithm. The second observation shows *where* optimization can be successful: It is worthwhile to optimize a factorization algorithm so that it makes efficient use of the way data is transferred to the computational unit.

To understand why the algorithms described in this paper are efficient (not only for multiprocessor computers but also for sequential machines), it is necessary to review the concept of a *memory hierarchy.*

Normally, the computation done in a central processing unit (CPU) is much faster than the time necessary to move the required data from the memory to the registers of the CPU. The process of moving the data is called *fetching* and the time required for transferring data from a part of the memory to the CPU is called *memory access time.* In order to use the processor efficiently it is important to keep the memory access time as small as possible. Unfortunately, it is too expensive to build very fast memories with sufficient capacity for scientific applications requiring huge amounts of data. Therefore, a *memory hierarchy* is used to decrease the cost of the memory system while retaining efficient memory access times. Figure 1 shows a typical memory hierarchy. The closer the memory level is to the registers of the processor the faster is the access.

For example, to use data stored in the external memory it has to pass through all levels of the memory hierarchy. Often, access time can be decreased if the usage of specific data can be predicted, so that data is transferred into a faster part of the hierarchy before it is actually referenced.

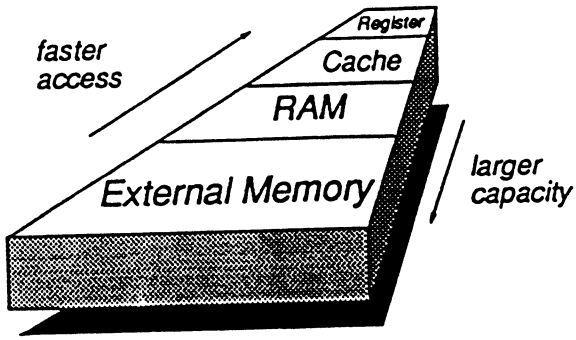

Figure 1: Typical memory hierarchy in a computer

One simple way to evaluate if a program can make use of the hierarchy in an efficient way is to keep the ratio of operations to data movement as large as possible. This ratio is important to achieve high performance when exploiting concurrency.
For example, the following statement inside a loop performing matrix multiplication,

$$c_{ij} \leftarrow c_{ij} + a_{ik} * b_{kj}$$

requires three memory accesses to obtain the data c_{ij}, a_{ik}, b_{kj}, and one to store the result in c_{ij}. Addition and multiplication count as one floating point operation each. The ratio of floating point operations to memory access time is $r = \frac{1}{2}$.
A simple programming trick to improve this ratio is to figure out how data is stored in the memory. One has to know that most memory organizations use specific strategies to reduce the memory access time. A common rule on many machines is to fetch a block of data instead of only one datum at a time. The distance between elements in the memory is called *stride*.
Therefore, it is best to formulate the algorithms in such a way that data elements used in consecutive computation steps are stored in contiguous addresses of the memory. Hence they are fetched in a block requiring fewer memory accesses. Figure 2 shows how data (a matrix) is stored in a memory using the Fortran programming language. Having this in mind it is obvious why Fortran is called a column oriented programming language.
Under the assumption, that a machine is able to fetch α contiguous data elements from the memory in one time step, some statements of the loop performing the matrix multiplication steps can be rewritten as

$$\begin{aligned} c_{ij} \leftarrow c_{ij} &+ a_{i,k} * b_{k,j} \\ &+ a_{i,k+1} * b_{k+1,j} \\ &\vdots \end{aligned}$$

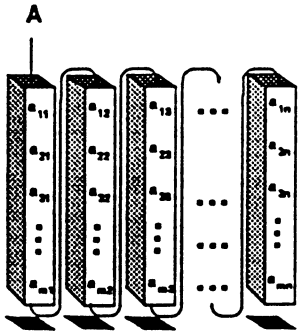

Figure 2: Storage of a two dimensional array in column oriented programming languages like Fortran

$$+ \quad a_{i,k+\alpha-1} * b_{k+\alpha-1,j} \tag{4}$$

This leads to 2α floating point operations, 2 memory accesses for storing and fetching c_{ij}, α memory accesses for fetching the a_{ik}'s and one memory access for all b_{kj}'s. The ratio is $r = \frac{2\alpha}{3+\alpha}$.
By storing the matrix A as its transpose, A^t, one can rewrite the multiplication as

$$\begin{aligned} c_{ij} \leftarrow c_{ij} \quad &+ \quad a^t_{k,i} * b_{k,j} \\ &+ \quad a^t_{k+1,i} * b_{k+1,j} \\ &\vdots \\ &+ \quad a^t_{k+\alpha-1,i} * b_{k+\alpha-1,j} \end{aligned} \tag{5}$$

where a^t_{ki} specifies the element in the k-th row and i-th column of A^t.
Now there is only one memory access necessary to fetch vector a^t $(a_{i,1}, \ldots, a_{i,k+\alpha-1})$. Therefore, the ratio is $r = \frac{\alpha}{2}$. The prediction of a maximal vector length α depends on many factors: the machine used, the memory hierarchy, and their fetching algorithm. Algorithms which update a block of contiguous vectors instead of only one vector at a time are known as *blocked algorithm*. This way the work is done *locally* on a block of data.
Previous numerical experiments [11] showed that traditional linear algebra algorithms do not achieve high performance on distributed-memory multiprocessors because of the lack of data locality. Therefore, data locality is the fundamental problem in parallel computing and has great influence on the performance on such machines. The use of block based algorithms is one of the most efficient ways to improve the performance of numerical algorithms on distributed memory machines.
Dongarra, Gustavson, and Karp [4] discussed six ways of implementing the LU factorization obtained by reordering the three nested loops that constitute the algorithm. Algorithm 1 explains

the generic Gaussian elimination. The loop indices are i, j and k.

```
do ————————
 do ————————
  do ————————
   a_ij ← a_ij − (a_ik * a_kj) / a_kk
  end do
 end do
end do
```

Algorithm 1: Gaussian Elimination Algorithm

Only three of these ways are applicable to the column oriented Fortran. These algorithms will be introduced later as block based algorithms with pivoting. Furthermore, the pivoting operation can be embedded in matrix multiplication if the elements a_{kj} are scaled by $-a_{kk}$ such that

$$a_{ij} \leftarrow a_{ij} + a_{ik} * a'_{kj}$$

where a'_{kj} is defined as $-\frac{a_{ij}}{a_{kk}}$.

1.1 Parallel Computational Model

In order to define a parallel algorithm to solve a system of linear equations it is necessary to define the computational model on which the implementation is based. A study on shared memory MIMD machines using blocked based algorithms for LU factorization can be found in [12].
The results presented in this paper concentrate on distributed memory MIMD machines. These machines have a natural bound on the number of available processors. At a time, each processor can execute different instructions in parallel on different data. Message passing allows interprocessor communication. To transfer a datum of specific length between two processors a startup time is needed to establish a communication path, and a transfer time which is proportional to the length of the message is needed to complete the transmission.
In order to incorporate a wide variety of architectures, the communication relation between the processors is based on a unidirectional ring.
Now it is clear that an efficient implementation has to find the trade off between communication time and computation time. If the algorithm sends too many messages and computes to less, the communication time dominates the computation time. Furthermore we assume that the memory capacity of each processing element is restricted.

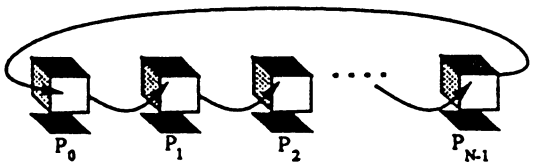

Figure 3: Parallel computing modell

2 Basic Linear Algebra Subprograms

In many applications, vector and matrix operations can be used to formulate algorithmic solution to scientific problems. Programming languages like Fortran77 do not provide these kinds of operations. To make programming easier for a scientists and engineers, it is desirable to have a library supporting such a class of routines.

A public domain set of Basic Linear Algebra Subprograms, called BLAS, has been very successful in scientific applications. Many algorithms and software packages make use of these programs [3]. Different levels of BLAS are distinguished by the *amount of data* used for an operation and its *arithmetic complexity.*

The complexity of programs at the same level of BLAS is equal. Computations on vectors of order n can be found in level 1 BLAS. For example the dot product of two vectors each with n elements is calculated in $2n$ arithmetic operations. Level 2 BLAS provides matrix-vector computations of order n^2, and level 3 BLAS provides matrix-matrix computations of order n^3 (table 1).

Level	Data type of operation	Arithmetic complexity
1	vectors	$O(n)$
2	matrix, vectors	$O(n^2)$
3	matrix, matrix	$O(n^3)$

Table 1: Arithmetic complexity of the different BLAS levels

Abbreviation	stands for
M	Matrix
V	Vector
GE	GEneral
TR	TRiangular

Table 2: Abbreviations used in BLAS

There are a number of important subprograms included in BLAS used for the algorithms presented in this paper. For example the matrix multiplication subprogram, called GEMM, and a subprogram for solving a triangular system, called TRSM.

The nomenclature of the BLAS programs is simple and gives information about the semantic of

the subprograms. Table 2 shows the abbreviations necessary to explain the algorithmic codings presented in this paper. Table 3 shows the BLAS subprograms used in the different implementations of the LU factorization algorithms.

BLAS Name	(Level)	Description as used in this paper	Arithmetic Complexity
IAMAX	(1)	finds the index of the element of a vector with the maximal absolute value	$O(n)$
SCALL	(1)	scale a vector by dividing with a constant	$O(n)$
SWAP	(1)	swap two vectors	$O(n)$
GEMV	(2)	multiply a general matrix with a vector	$O(n^2)$
TRSV	(2)	solve a triangular system where the result is a vector	$O(n^2)$
GEMM	(3)	multiply a general matrix with another general matrix	$O(n^3)$
TRSM	(3)	solve a triangular system where the result is a matrix	$O(n^3)$

Table 3: List of BLAS routines used for blocked factorization algorithms

Looking at the computational effort of the BLAS routines it is clear that the ratio between floating point operations and memory accesses for the level 1 and 2 BLAS is not as good as for the level 3 BLAS which consists of more computations per memory access. Therefore, it is obvious that the strategy is to maximize the use of level 3 BLAS.

3 Noblock algorithm

Now a necessary basis has been established to formulate the factorization algorithms. A noblock factorization algorithm forms the building block for the block based algorithms. To be most efficient a fast noblock algorithm has to be selected. The noblock algorithms are distinguished by the order of loops in which the factorization is done. The suitable loop orders for the column oriented FORTRAN are *jik*, *kji*, and *jki*. For example, the abbreviation *jik* points out that j is the loop

index for the outermost loop and k for the inner most loop (compare to algorithm 1).
Since the number of memory touches for the kji noblock algorithm is twice as high as for the jki-noblock and the jik-noblock algorithm [4], the running time for this algorithm is slower. Experiments show that the jki noblock algorithm performs better than the two others.

3.1 jik-Noblock Algorithm

Before the algorithm is described in detail it is useful to visualize the data dependencies of the $n \times n$ matrix elements between the computational steps (figure 4) of the jik-noblock algorithm. Data dependencies are expressed by the height of the matrix element in the figure. If a datum is higher than another then this datum has to be calculated first.
In figure 4 the state of the algorithm is shown at time step j. Following the data dependencies first the vector $l^{(j)}$ is updated with the help of $A^{(j)}$ and the vector $u_r^{(j)}$; next the element of l_1^j is computed and $u^{(j)}$ is updated with the help of $U^{(j)}$ and $l_r^{(j)}$.
Therefore, at time step j the jik-noblock algorithm updates one *column* of L and one *row* of U. This noblock algorithm is also also known as Crout's Algorithm.

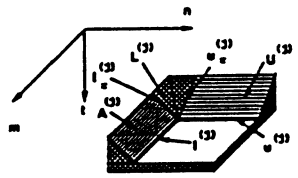

Figure 4: jik noblock

Let $l^{(j)}$ represent the j^{th} column vector of the matrix L and $u^{(j)}$ the j^{th} row vector of the matrix U beginning at a_{jj}. The matrix $\mathbf{A}^{(j)}$ specifies a submatrix of A which includes all elements from the first column to the column $j-1$ and from the rows $j+1$ to n. The matrix $\mathbf{U}^{(j)}$ specifies a submatrix of A which includes all elements from the first row to the row $j-1$ and from the columns $j+1$ to n.

0. Initialization: $j \leftarrow 1$

1. Update $\mathbf{l^{(j)}}$: $l^{(j)} \leftarrow l^{(j)} - \mathbf{A}^{(j)} u^{(j)}$

2. Select pivot and exchange: $p \leftarrow j + \min\left\{ j \middle| |l_j| = \max_{1 \le i \le m-j+1} \{|l_i^{(j)}|\} \right\} - 1$
Exchange row j and row p

3. Scaling: $l_i^{(j)} \leftarrow l_i^{(j)} / a_{p,j} \quad \forall i \in [1, m-j+1]$

IF $j = n$ THEN stop

4. Compute row u(j): $u^{(j)} \leftarrow u^{(j)} - U^{(j)} l_r^{(j)}$

5. Iterate: $j \leftarrow j + 1$

GOTO Step 1.

The detailed description of the algorithms using BLAS can be found in [12].

3.2 Naming convention

In literature, LU factorization algorithms are often named by the basic operations inherent in the algorithms[7]. The basic operation of the kji algorithm is based on the following operation: $\vec{z} \leftarrow a\vec{x} + \vec{y}$, where a is a scalar and $\vec{x}, \vec{y}, \vec{z}$ are vectors. This operation of a Scalar A multiplied by the vector X Plus the vector Y is called *SAXPY*.
The basic operation of the jki algorithm is based on the following operation: $\vec{z} \leftarrow \mathrm{A}\vec{x} + \vec{y}$, where A is a matrix and $\vec{x}, \vec{y}, \vec{z}$ are vectors. Therefore, it is a generalized SAXPY operation working on a matrix rather than on a vector. Hence the name *GAXPY* is used.
The basic operation of the jik algorithm is based on the dot product: $\vec{z} \leftarrow a\vec{x}^T\vec{y}$, where $\vec{x}, \vec{y}, \vec{z}$ are vectors and a is a scalar. Therefore, it is called SDOT. The names introduced above are used in the later sections.

4 Parallel Blocked Algorithms

The trade off between the communication time and the computation time, as introduced earlier, can be achieved by rewriting the LU decomposition with the help of matrix blocks, with properly defined block sizes.
To understand the parallel block factorization algorithm the corresponding sequential block based algorithms are also introduced. This way one is able to observe the data dependencies inherent in the algorithms. The parameter β specifies the block size which is the number of column vectors used by the noblock algorithm for factorization in each iteration of the block based algorithm.

4.1 Parallel Blocked *jki*-GAXPY

The sequential jki-GAXPY algorithm computes a block column of both matrices L and U at the j^{th} step of the elimination. The following operations are required (compare with figure 6):

0. Initialize: Start with first block.
$j \leftarrow 1$

1. Pivot and Update $U_2^{(j)}$: Apply previous interchanges to the block $U_2^{(j)}$.

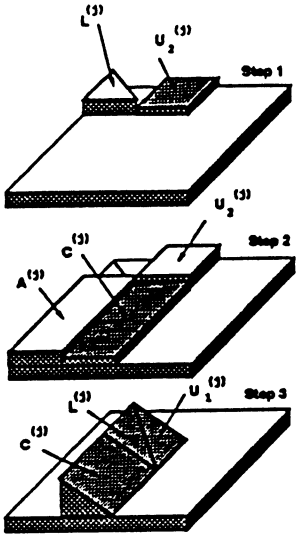

Figure 5: jki-GAXPY

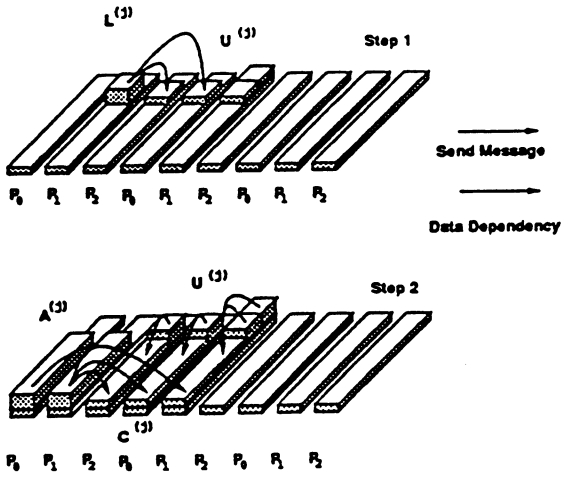

Figure 6: Parallel GAXPY a) update U_k, b) update C_k

The j^{th} superdiagonal block of U is computed using TRSM:

$U_2^{(j)} \leftarrow (L^{(j)})^{-1} U_2^{(j)}$

2. Update $C^{(j)}$: The j^{th} diagonal and subdiagonal blocks of C are computed using GEMM:

$C^{(j)} \leftarrow C^{(j)} - A^{(j)} U_2^{(j)}$

3. Factorize $C^{(j)}$: The j^{th} block column is factorized into LU factors using a noblocked algorithm (jik-noblock).

4. Iterate: IF no more blocks THEN stop

ELSE GOTO Step 1.

Because the sequential algorithm updates only one block at a time, a corresponding parallel version of this algorithm should be restricted to this block. Nevertheless, to be efficient the parallel algorithm has to work interleaved. Therefore, the data is spread in a scattered fashion over the processors as shown in figure 6. Information is exchanged between the processors after the factorization step is completed using the noblock algorithm. Looking at the data dependencies of the sequential algorithm it is clear that in order to update the matrix each processor has to know the data L computed so far and $A^{(j)}$ even if they are generated in another processor. Hence, each processor has to store a complete copy of the original matrix.

To simplify the description of the parallel GAXPY algorithm a submatrix A from row y_0 to y_1 and from column x_0 to x_1 is indicated by the Fortran 90 statement $A(y_0 : y_1, x_0 : x_1)$. The command $\stackrel{\text{def}}{=}$ indicates a macro statement. Its variables are expanded at runtime. The definition $L \stackrel{\text{def}}{=} A(1 : j, 1 : j)$ leads to the statement $A(1 : 4, 1 : 4)$ for L if the value of the variable j equals 4. The following definitions of submatricies are used in the parallel GAXPY algorithm, where Ly_0, Lx_0 denotes the first row and column of the current block to be factorized, and $Lx_1 = Lx_0 + \beta$, and $Ly_1 = Lx_1 + \beta$:

$L \stackrel{\text{def}}{=} A(1 : j, 1 : j)$	$\Leftrightarrow$ the lower triangular matrix calculated so far
$A^{(j)} \stackrel{\text{def}}{=} A(Ly_1 + 1 : n, 1 : j)$	$\Leftrightarrow$ the matrix used for updating $C^{(j)}$
Work block $\stackrel{\text{def}}{=} A(Ly_0 : n, Lx_0 : Lx_1)$	$\Leftrightarrow$ a submatrix of $U^{(j)}$ of the seq. alg.
$L^{(j)} \stackrel{\text{def}}{=} A(Ly_0 : Ly0 + \beta, Lx_0 : Lx_1)$	$\Leftrightarrow$ a submatrix of L
$C^{(j)} \stackrel{\text{def}}{=} A(Ly_1 + 1 : m, Ux_0 : Ux_0 + \beta - 1)$	$\Leftrightarrow$ a submatrix of $C^{(j)}$ of the seq. alg.

Different $U^{(j)}$'s must be distinguished. One for the sending processor

$$U_s^{(j)} \stackrel{\text{def}}{=} A(1 : Ly_0, Ux_0 : Ux_0 + \beta - 1)$$

and the others for the receiving processors

$$U_r^{(j)} \leftarrow A(Ly_0 : Ly_0 + \beta, Ux_0 : Ux_0 + \beta - 1)$$

The *Process ID* $\in [0, Processors - 1]$ is used to distinguish the processors from each other. Note that the variable Ux_0 is dependent on the *Process ID*.
With this semantical abbreviations the parallel algorithm can be formulated as shown in algorithm 2.
Figure 6 shows the algorithm at time step $j = 1$ (note that the time steps start from 0). Only one processor calculates the factorization at a time. The others are updating their matrix with the result of this computation. First the block $L^{(j)}$ is obtained with the help of the noblock factorization algorithm. The result of the computation is submitted to its neighbor processor. Then the part of $U_s^{(j)}$ assigned to this processor is calculated. The neighbor processor receives the factorized block and sends it itself to its neighbor. As soon as the block is sent, the processor updates its part of $U_r^{(j)}$. In addition each processor performs pivoting to all the columns smaller than $U^{(j)}$. With the help of $A^{(j)}$ it is now possible to update $C^{(j)}$.
In the example shown in figure 6 three processors are involved in the computation. Therefore, the update is split into three parts. The processor which does the factorization has to update its part of $C^{(j)}$ with all blocks of $U^{(j)}$ directly above $C^{(j)}$. In contrast, the other processors update their part only with the first block above their part of $C^{(j)}$. Looking at the next time steps it is easy to see that the update of $C^{(j)}$ is completed after three time steps (which is the number of processors).

The disadvantage of this algorithm is clearly the need for storing the matrix in each processor. This data redundancy limits the use of this algorithm to small matrices. The advantage of the algorithm is that it is fast due to the data redundancy and the interleaved execution.
To allow even bigger matrices to be calculated on a parallel machine with restricted memory capacity, other algorithms are needed. They are introduced in the next two sections.

4.2 Parallel Blocked *jik*-SDOT

In the blocked *jik*-SDOT or Crout's algorithm [2], one block column of L and one block row of U are computed in each iteration. The basic steps involved in the j^{th} iteration are shown figure 7 along with the data dependencies involved in each step. The steps are described below:

0. Initialize Start with the first block

$j \leftarrow 1$

1. Update $C^{(j)}$ The diagonal and subdiagonal blocks of the j^{th} block column are computed using GEMM:

$C^{(j)} \leftarrow C^{(j)} - A^{(j)}B^{(j)}$

2. Factorize $C^{(j)}$ The j^{th} block column, $C^{(j)}$ is factorized into LU factors using the *jik*-noblock algorithm.

The row interchanges are applied to blocks on both sides of the current block.

3. Update $U_2^{(j)}$ a) The j^{th} block row of U is updated using GEMM:

$U_2^{(j)} \leftarrow U_2^{(j)} - A^{(j)}E^{(j)}$

b) The j^{th} block row of U is computed using TRSM:

$U_2^{(j)} \leftarrow (L^{(j)})^{-1}U_2^{(j)}$

4. Iterate: IF no more blocks remaining THEN stop

ELSE goto Step 1.

In the j^{th} iteration, the j^{th} block depends on the all of the $j-1$ previously factorized blocks. With the matrix laid out onto the processor (as shown in figure 8) in a manner similar to that presented in the parallel GAXPY algorithm, this dependency requires that the factorized matrix is stored in each processor. As a consequence, the size of the matrix which can be factorized by this algorithm is limited by the available memory at each node. This limitation can be overcome to a certain extent by observing that in each iteration, the block to be factorized depends only on the portion of the factorized submatrix which includes and which is located below the current block row. In

SUBPROGRAM jki-GAXPY-parallel $(m, n, a, lda, ipiv)$

Factorization Counter ← Process ID

Ux_0 ← process ID $*\beta + 1$

DO $j = 0, n/\beta - 1$

$Lx_0 \leftarrow Ly_0 \leftarrow j * \beta + 1$; $Lx_1 \leftarrow Lx_0 + \beta - 1$

IF Factorization Counter = 0 THEN

Factorize the work block

SEND $L^{(j)}$ and the pivot vector to the right neighbor processor

Apply pivoting to the columns before $L^{(j)}$, and after $L^{(j)}$ up to
the end of the block where this processor will factorize the next time

$Ux_0 \leftarrow Lx_0$ + Processors $* \beta$

IF $Ux_0 \leq n$ THEN

Update $U_s^{(j)}$ using L computed so far

Update $C^{(j)}$ using $U_s^{(j)}$ and $A^{(j)}$

ENDIF

Factorization Counter ← Processors

ELSE

RECEIVE $L^{(j)}$ and pivot vector from the left neighbor processor

IF (Factorization Counter + 1) $\neq$ Processors THEN

SEND $L^{(j)}$ an pivot vector to the right neighbor processor

ENDIF

Store $L^{(j)}$ in A

Apply pivoting to the columns before $L^{(j)}$, and after $L^{(j)}$ up to
the end of the block where this processor will factorize the next time

$Ly_1 \leftarrow Ly_0 + \beta - 1$

IF $Ux_0 \leq n$ THEN

Update $U_r^{(j)}$ using $L^{(j)}$

Update $C^{(j)}$ using $U_r^{(j)}$ and $A^{(j)}$

ENDIF

ENDIF

Factorization Counter ← Factorization Counter - 1

END DO

Algorithm 2: jki-GAXPY-parallel

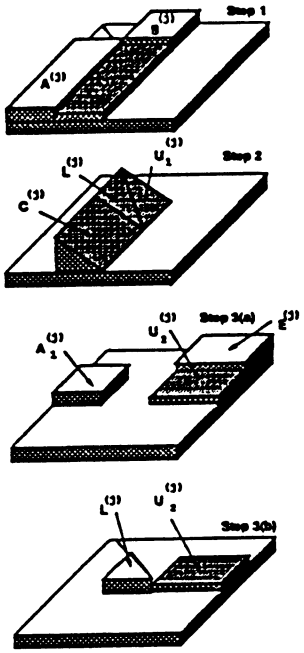

Figure 7: *jik*-SDOT

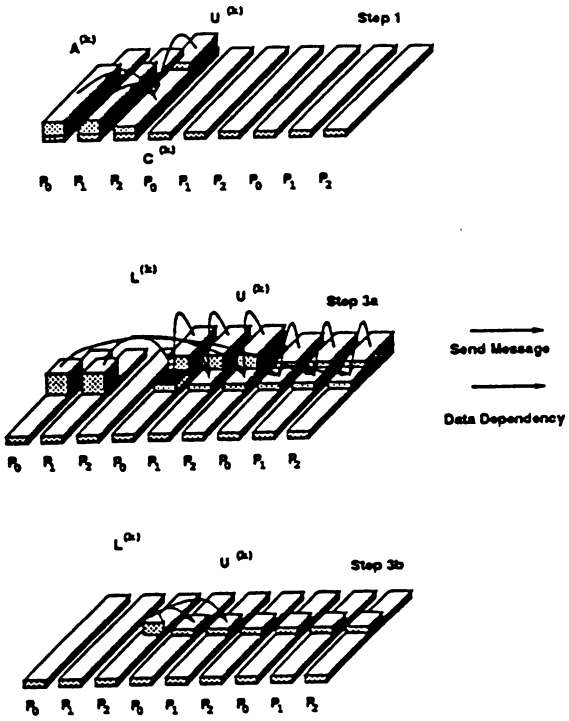

Figure 8: Parallel SDOT

the presented implementation, we store the factorized submatrix in a work array. At the beginning of each iteration, the work array is reshaped so as to retain only that portion of the factorized submatrix required for subsequent computations, thereby overcoming the memory limitation.
The structure of the algorithm requires that the blocks of the matrix are factorized in a sequential order. In order to offset this inherent bottleneck, a pipelined approach is used [9]. In this approach, the iterations of the algorithm are pipelined so as to overlap the factorization of the j^{th} block column (steps 1 and 2) with the update of the block row associated with the $j-1^{th}$ block column (step 3). The pseudo-code for the pipelined version of the algorithm is given in Algorithm 3. Here, n = number of columns of the matrix, m = number of rows of the matrix, lda = leading dimension of the matrix, $Processors$ = number of allocated processor, $proc$ = number of the processor currently performing the factorization and shipping of the factorized panel, and $myid$ = id of a processor. A is the matrix to be factorized. Note that SGETF2 refers to the noblock factorization routine used. (Since, in our experimentation with the algorithms we used single precision data, all BLAS routines used have a "S" prefix). Figure 8 shows the layout of the matrix onto the processor along with the operations in the third iteration. The activities of each of the processors in the pipelined algorithm are shown in figure 9 for a four processor system.

```
subprogram jik-SDOT-parallel (m, n, a, lda, ipiv)
  1. Processor 0 starts the pipeline
      proc = 0
     IF ( myid = proc ) THEN
       call noblock routine (SGETF2) to factorize first panel
       SEND factorized panel & pivots to all processors using BLACS routine (SGESD)
     ENDIF
  2. Repeat for each panel
     DO  i = 1, n, β
       IF ( i > 1)  THEN
         Reshape work matrix
       ENDIF
       IF ( proc = myid ) THEN
          copy factorized panel into work matrix
          apply pivoting permutations to work matrix and rest of my panels
       ELSE
          RECEIVE factorized panel & pivots from processor proc (SGERC)
          apply pivoting permutations to work matrix and rest of my panels
       ENDIF
       proc = mod(proc+1,Processors)
       IF ( I have panels left to factorize ) THEN
          IF ( proc = myid ) THEN
             call SGEMM to update i^th block row of next panel only
             call STRSM to compute i^th block row of next panel only
             call SGEMM to update the next panel
             call SGETF2 to factorize next panel
             SEND factorized panel & pivots to all processors (SGESD)
          ENDIF
          all processors update their remaining panels by calling SGEMM & STRSM
       ENDIF
     ENDDO
```

Algorithm 3: Pipelined Parallel *jik* SDOT

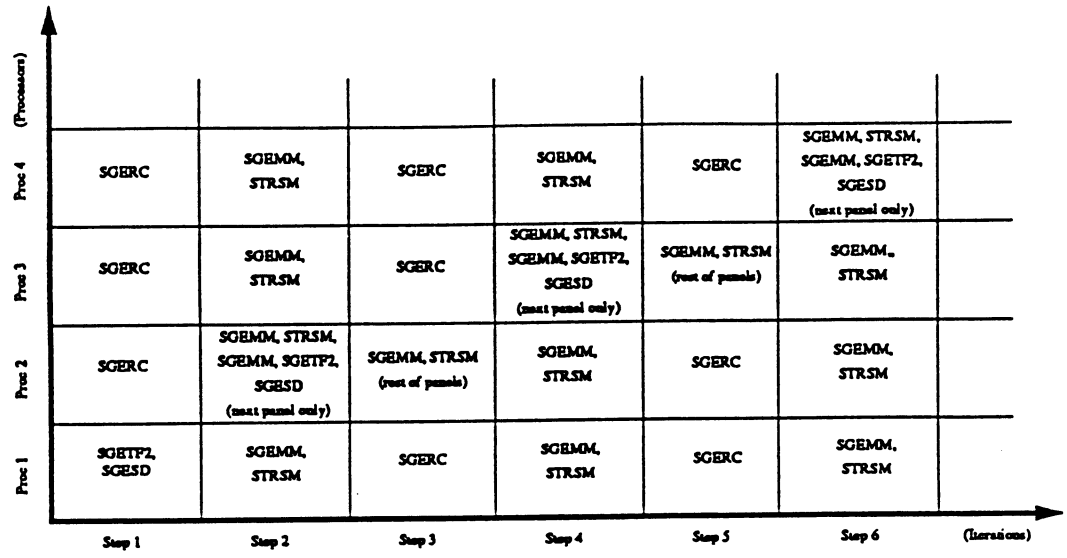

Figure 9: Spacetime diagram for the pipelined jik-SDOT algorithm

Although, using the pipelined implementation did provide some performance improvement for large matrices when compared with the non pipelined version, this improvement was very limited. The reason being that the amount of work involved in updating and computing the block row, i.e. step 3 (rest of the processors) is small compared to work required to update and factorize the subdiagonal block (current processor). This unbalance of work along with the overhead involved in reshaping the work matrix in each iteration prevents the pipeline from remaining full and limits the improvement in performance that can be obtained. Numerical results for the jik-SDOT algorithm for various matrix and blocks sizes are presented in a later section.

4.3 Parallel Blocked kji-SAXPY

In the j^{th} step of the kji-SAXPY algorithm, one block column of L and one block row of U are computed and the corresponding transformations are applied to the remaining submatrix. The basic steps involved in the j^{th} iteration are are shown in figure 10.

0. Initialize Start with the first block

$j \leftarrow 1$

1. Factorize $C^{(j)}$ The j^{th} block column is factorized into LU factors using the jik-noblock algorithm.

The row interchanges are applied to blocks on both sides of the current block.

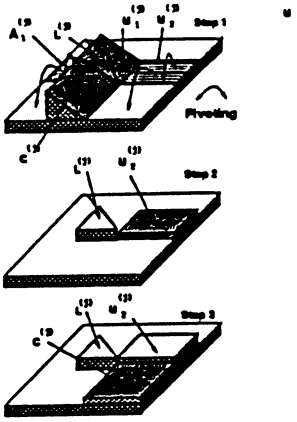

Figure 10: kji-SAXPY

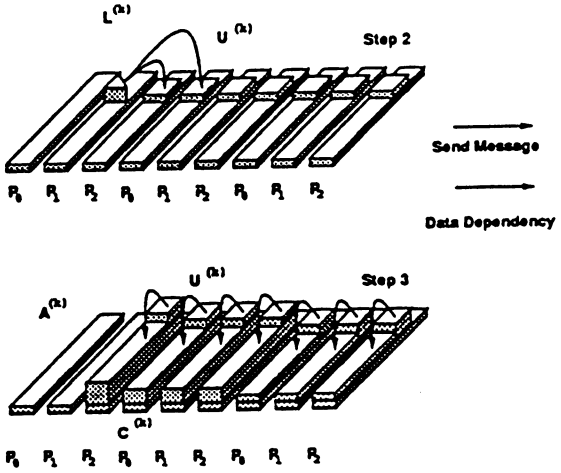

Figure 11: Parallel SAXPY

2. Update $U_2^{(j)}$ The j^{th} block row of U is computed using TRSM:

$$U_2^{(j)} \leftarrow (L^{(j)})^{-1} U_2^{(j)}$$

3. Update $C^{(j)}$ Updating the remaining matrix using a block outer product (GEMM):

$$C^{(j)} \leftarrow C^{(j)} - L^{(j)} U_2^{(j)}$$

4. Iterate: IF no more blocks remaining THEN stop

ELSE goto Step 1.

This algorithm is best suited, of the three algorithms presented, for distributed memory MIMD architectures. The reason for this is the data dependencies involved in the steps above (shown in figure 10.) In the j^{th} iteration, the j^{th} block depends only on the $j-1^{th}$ factorized block. Hence, each node has to store only the last factorized panels. As a result the memory limitations encountered in the parallel SDOT and GAXPY algorithms do not exist here. Furthermore, the amount of work involved in updating and computing the block row, i.e. steps 2 and 3 (rest of the processors) is comparable to work required to update and factorize the subdiagonal block (current processor). Hence, the pipelined version of this algorithm produces a significant improvement in performance. The pseudo-code [9] for the pipelined version of the algorithm is given in Algorithm 4. The notations are the same as those defined in the previous section for the parallel SDOT algorithm. Figure 11 shows the layout of the matrix onto the processor along with the operations in the second iteration. The activities of each of the processors in the pipelined algorithm are shown in figure 12 for a four processor system.

```
subprogram kji-SAXPY-parallel (m, n, a, lda, ipiv)
  1. Processor 0 starts the pipeline
     proc = 0
     IF ( myid = proc ) THEN
       call noblock routine (SGETF2) to factorize first panel
       SEND factorized panel & pivots to all processors using BLACS routines (SGESD)
     ENDIF
  2. Repeat for each panel
     DO i = 1, n, β
      IF ( proc = myid ) THEN
        copy factorized panel into work matrix
        apply pivoting permutations to work matrix and rest of my panels
      ELSE
        RECEIVE factorized panel & pivots from processor proc (SGERC)
        apply pivoting permutations to work matrix and rest of my panels
      ENDIF
      proc = mod(proc + 1,Processors)
      IF ( I have panels left to factorize ) THEN
        IF ( proc = myid ) THEN
          call STRSM to compute i^th block row of next panel only
          call SGEMM to update the next panel only
          call SGETF2 to factorize next panel
          SEND factorized panel & pivots to all processors (SGESD)
        ENDIF
        all processors update their remaining panels by calling STRSM & SGEMM
     ENDIF
  ENDDO
```

Algorithm 4: Pipelined Parallel *kji*-SAXPY

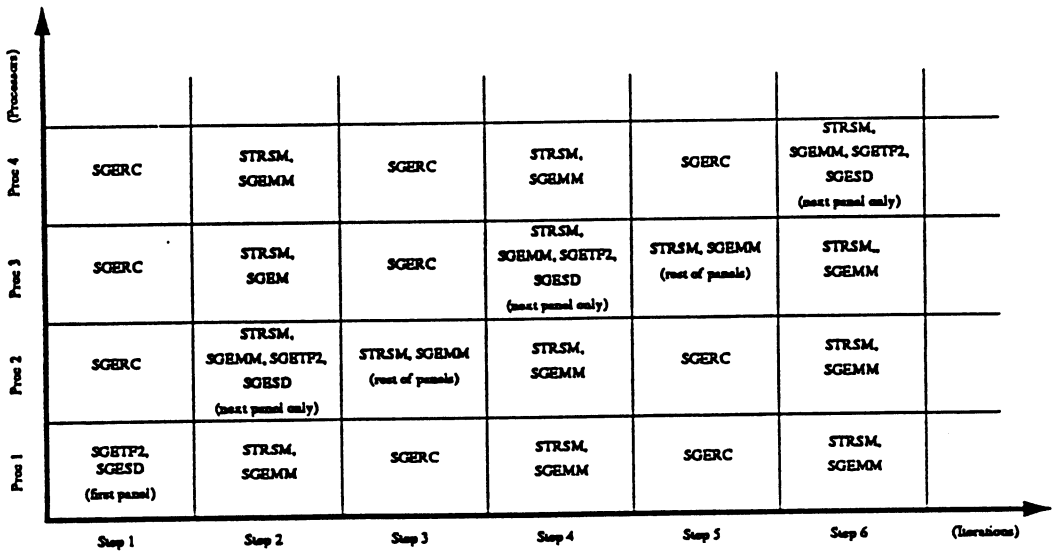

Figure 12: Spacetime diagram for the pipelined kji-SAXPY algorithm

5 Results

The presented study was conducted on a 32 node Intel iPSC/860 which is an Intel i860 processor based Hypercube. Each i860 node has an 8 KByte cache and 8 MBytes of main memory. The clock speed is 40 MHz, and each node has a theoretical peak performance of 80 MFLOPS for single precision and 40 MFLOPS for double precision. Communication is supported by direct-connect modules present at each node [16] which allow the nodes to be treated as though they are directly connected. The communication time for a message is a linear function of the size of the message. Hence, the time, t_m to transmit a message of length n from an arbitrary source node to an arbitrary destination node is given by:

$$t_m = t_s + t_b \times n$$

where t_s is the fixed startup overhead and t_b is the transmission time per byte.

PICL [8] (Portable Instrumented Communication Library) is used in the presented implementations and provides a simple and portable communication structure. PICL also provides tracing facilities which can be used in conjunction with ParaGraph [10], a tool for visualizing the behavior and performance of parallel programs.

In addition to the above, the presented implementations also make use of the BLACS [1] (Basic Linear Algebra Communication Subprograms) library for data movement. BLACS is a portable, high level communication library, developed for linear algebra applications. It is a part of the effort to implement LAPACK on distributed memory MIMD architectures. The matrices used in the experiments were dense matrices where each matrix element was a randomly generated real

number between 0 and 1.

In order to compare the parallel implementations of the three factorization algorithms we first compared the performance achieved for a constant matrix size with different block sizes and on different numbers of processors. Figure 13 shows the results for a 512×512 matrix computed on 4,8,16 and 32 processors and with different block sizes. The parallel GAXPY algorithm performs better than the other two algorithms for a smaller number of processors. The figure shows that the maximal performance is achieved by the parallel GAXPY algorithm using 16 processors and a block size of 8. The performance of the of the SAXPY algorithm is close to the GAXPY algorithm and achieves the best performance for block sizes in the range 8 ± 4. The SDOT algorithm achieves very little improvement over the the noblock algorithms because of its data dependencies as described earlier.

The performance obtained for very small block sizes is low for all the three algorithms. Although small block sizes provide better load balancing, this advantage is offset by the increased communication between the processors and the decrease in the amount of computation at each processor. Very large block sizes on the other hand, lead to low communication overheads but have poor load balancing. This leads to a fall in performance for the parallel GAXPY and SAXPY algorithms as the block size increases. The observation however, does not hold true for the parallel SDOT algorithm which shows a slight improvement in performance for very large block sizes. The reason for this is that large block sizes imply smaller number of reshapes which means a smaller penalty is paid by the SDOT algorithm.

Figures 14, 15 and16 show the graphs for SAXPY and SDOT algorithms for larger matrices. The parallel GAXPY algorithm could not be used for these matrices because of its inherent limitation on the the matrix size. The penalty paid for reshaping and work unbalance in the SDOT are obvious in each of these plots. The SAXPY algorithm continues to perform well and achieves high MFLOP's as the matrix size increases.

The peak performance in MFLOPS per processor for various matrix sizes is shown in figures 17 and18 for 4 and 16 processors respectively. An absolute peak performance of about 26 MFLOPS/Processor is achieved by the parallel SAXPY algorithm for a 1536×1536 matrix with β equal to 12 and on 4 processors. These figures (17 and 18) also show that the optimal number of processors is problem dependent. A larger number of processors does not necessarily imply better performance.

Figure 19 show the scalability of the algorithms for different block sizes for 4, 8, 16 and 32 processors. SAXPY scales almost linearly with the problem size for large number of processors as is clear from the figure.

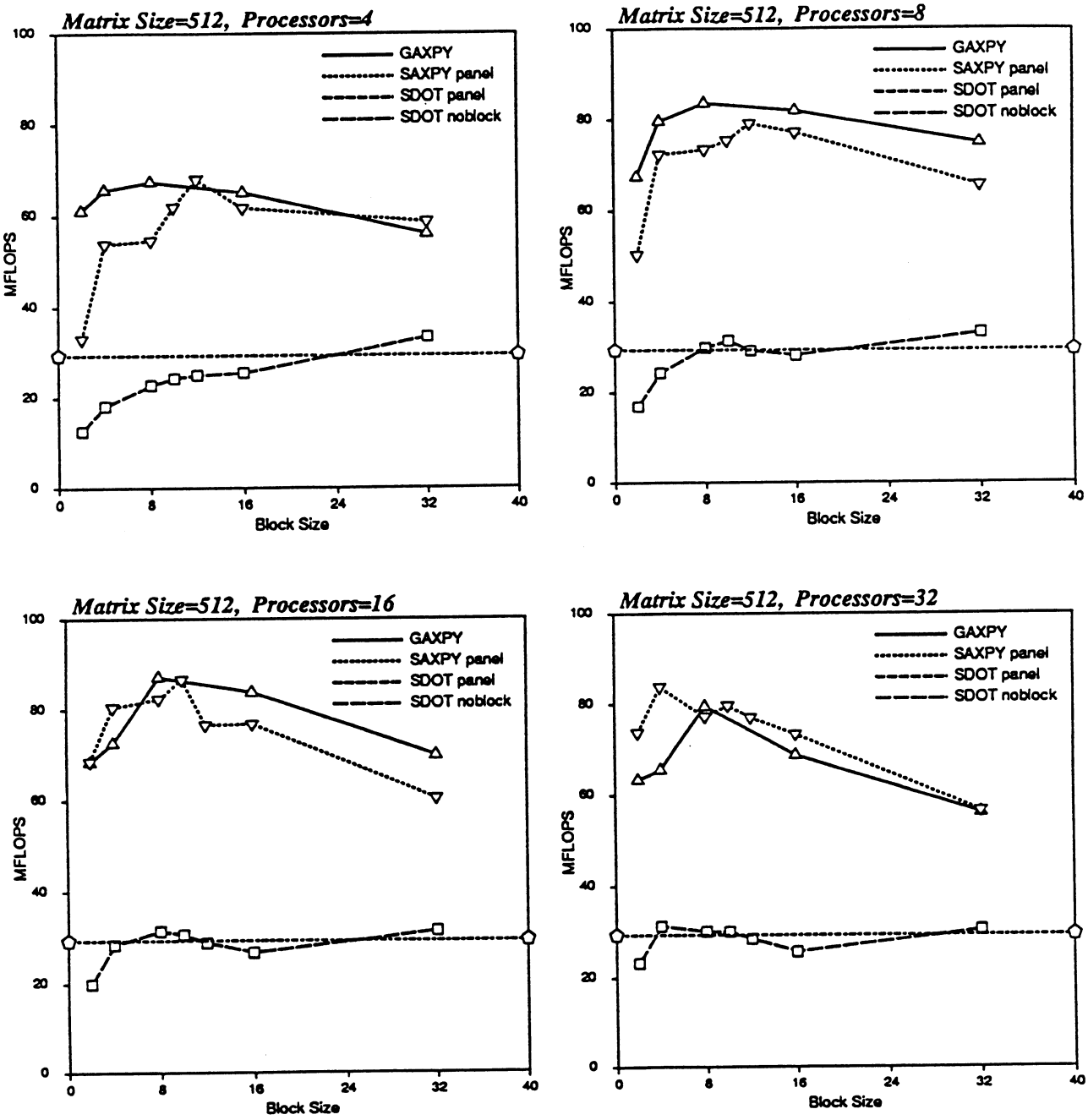

Figure 13: Performance of the different algorithms on a constant matrix size.

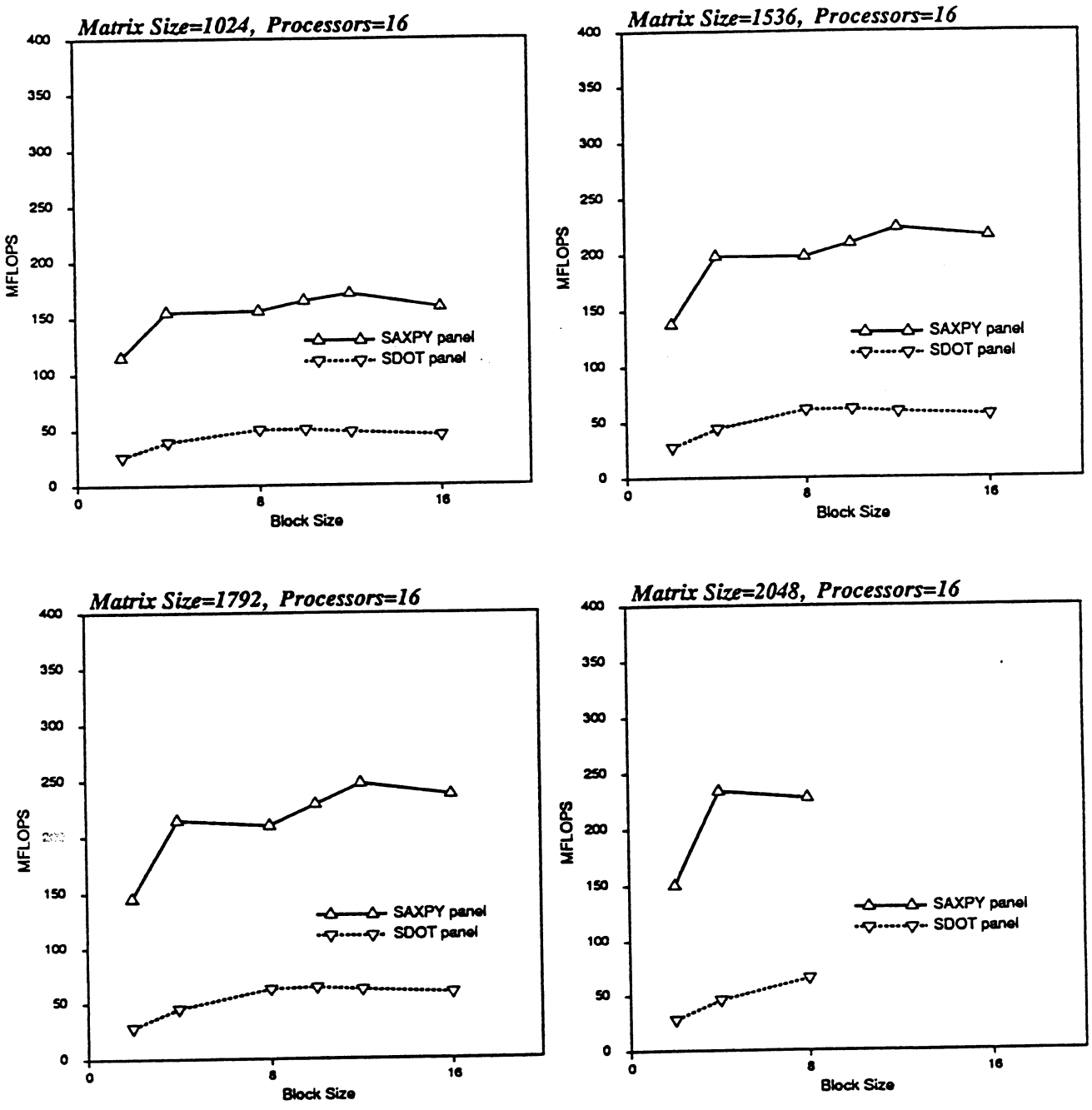

Figure 14: Performance of the different algorithms using constant number of processors.

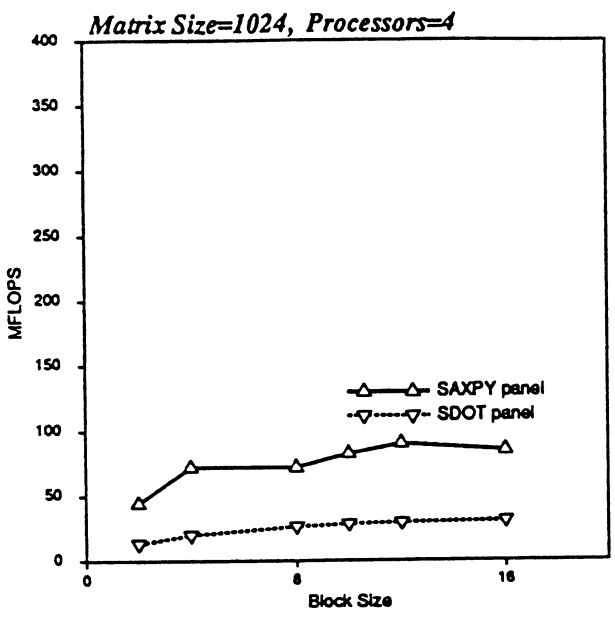

Figure 15: 1024 ×1024 Matrix, 4 Processors, Pipelined SDOT SAXPY.

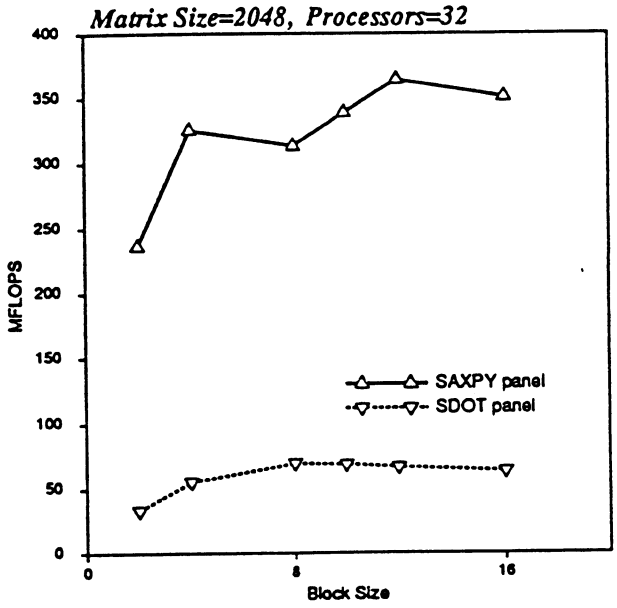

Figure 16: 2048 ×2048 Matrix, 32 Processors, Pipelined SDOT SAXPY.

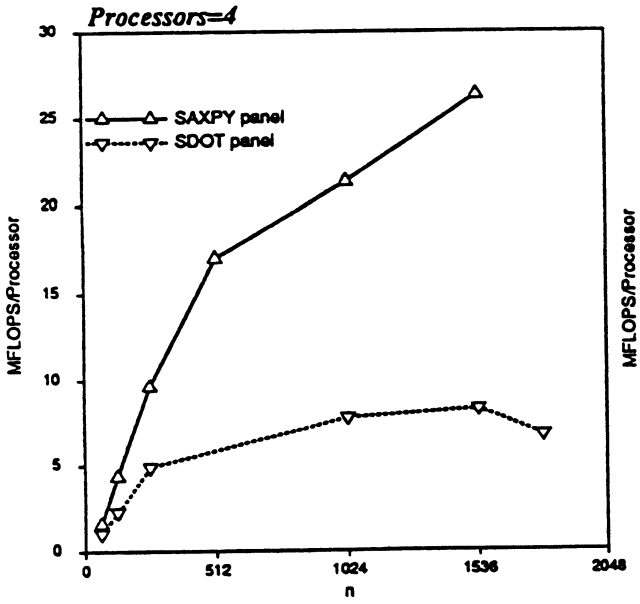

Figure 17: Best Ratio of MFLOPS per processor for different matrix sizes and variable block size (4 processors).

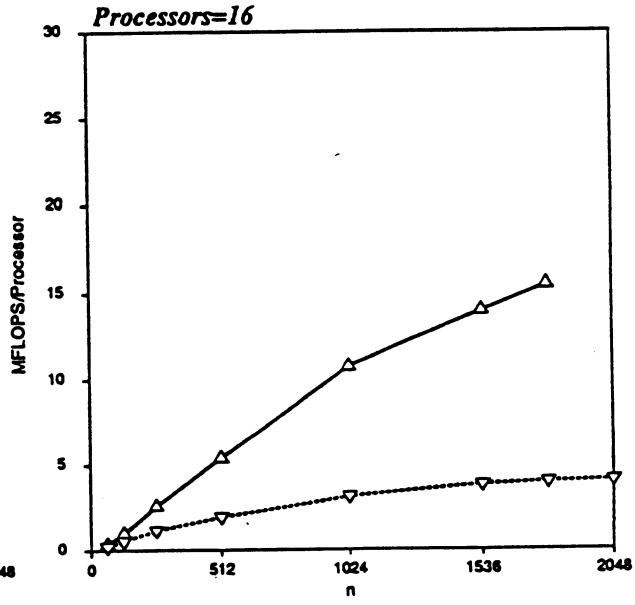

Figure 18: Best Ratio of MFLOPS per processor for different matrix sizes and variable block size (16 processors).

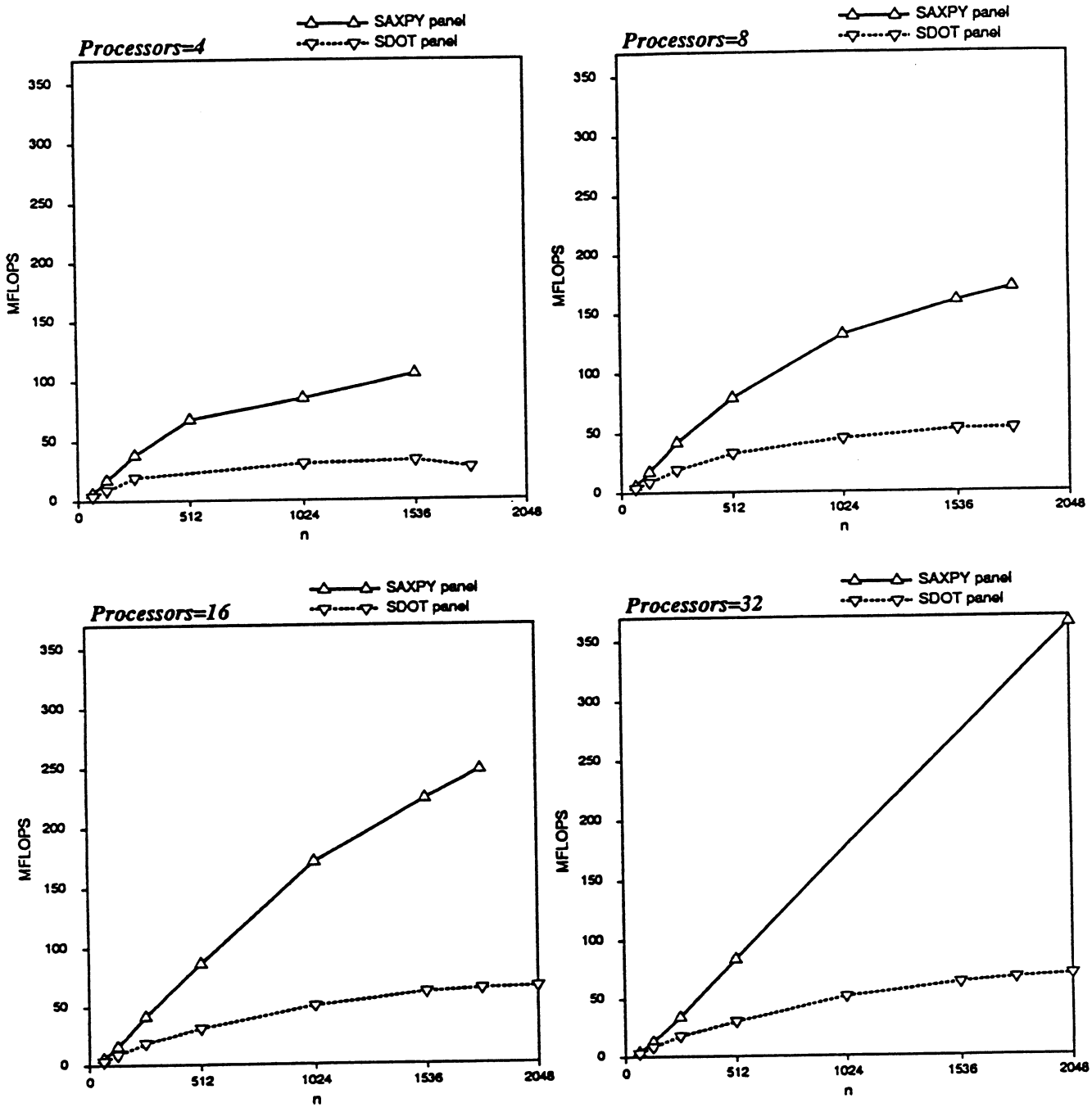

Figure 19: Best performance of the different algorithms on different numbers of processors using the optimal block sizes

6 Conclusion

This paper describes the Fortran-oriented methods for block LU factorization on distributed memory MIMD architectures. These methods are equally applicable shared memory parallel vector computers [12]. Our numerical results and performance comparisons show the following:

GAXPY

- The parallel GAXPY algorithm is very fast for small matricies. However, it does not scale up due the fact that the entire matrix needs to be stored in each nodal memory.
- Because of the need to store the matrix in each node, the memory capacity of the node limits the maximal problem size for this algorithm.
- Paneling was not necessary for this algorithm and even without paneling, this algorithm performs very well.

SDOT

- With the help of reshaping, the paneled version of the parallel SDOT algorithm is able to factorize large matricies, which could not be solved otherwise. Since the reshaping is time insensitive, this algorithm has the worst performance of the three studied. It is only slightly better than the sequential program for large enough block sizes since this implies that reshaping is done more seldom.

SAXPY

- The data dependencies inherent in the parallel SAXPY algorithm are most suited for distributed memory MIMD architectures.
- No reshaping of the matrix is necessary since only a small portion of the factorized matrix has to be stored in each processor.
- Parallel SAXPY provides an efficient algorithm which scales effectively with the matrix size and can be used with a wide of number of processor.

A further observation from our experimentation with LU factorization is that the best performance is achieved at block sizes where the computation at each node outweighs the tradeoff between high load balancing (small block sizes) and low communication overhead (large block sizes). This optimal block size is dependent on the algorithm used, the size of the matrix and the number of processors available.

In conclusion we make the following recommendations:

- The parallel GAXPY algorithm should be used in case of small matricies and few available processors.
- The parallel SAXPY algorithm should be used for larger matrices or if the matrix size varies over a wide range and the is the number of processors is variable.

- The block size should be chosen depending on the algorithm used, the size of the matrix and the number of processors used so as to maximize performance. Graph of the type provided in this paper my help in making this decision.

7 Future

Currently, we are testing the three blocked **LU** factorization algorithms in different FORTRAN dialects on a variety of parallel platforms machines. We have already implemented versions for SIMD and shared memory MIMD machines.
The target machines to be used in this research include the Intel iPSC/860,nCube, Alliant FX/80 [12], IBM 3090, Decmpp 12000, CM2, and the CM5.
This work is being done as a part of our effort to develop a benchmark suite for FORTRAND and the proposed HPFF (High Performance Fortran Forum).
To obtain a copy of the software used in this study, send a one-line e-mail message "send index" to allus@netlib.npac.syr.edu. Allus is a free software distribution electronic service. The index lists information on how to access all the programs used in this study. Users who have problems accessing these programs should send e-mail to the authors at agm@nova.npac.syr.edu.

Acknoledgement

The presented research is sponsored by DARPA under contract #DABT63-91-k-0005. The content of the information does not necessary reflect the position or the policy of the Government and no official endorsement should be inferred.
Use of the Intel iPSC/860 was provided by the Center for Research on Parallel Computation under NSF Cooperative Agreement Nos. CCR-8809615 and CDA-8619893 with support from the Keck foundation.
We would like to thank Jack Dongarra for making a preliminary version of LAPACK available to us and to Susan Ostrouchov for her help with **LU** factorization on the iPSC/860.

References

[1] ANDERSON, E., BENZONI, A., DONGARRA, J., MOULTON, S., OSTROUCHOV, S., TOURANCHEAU, B., , AND VAN DE GEIJN, R. Basic Linear Algebra Communication Subprograms. *Sixth Distributed Memory Computing Conference Proceedings, IEEE Computer Society Press* (1991), pp. 287–290.

[2] DAYDE, M. J., AND DUFF, I. S. Level 3 BLAS in LU Factorization on the CRAY-2, ETA-10P, and IBM 3090-200/VF. *The International Journal of Supercomputer Applications* (1989), pp. 40–70.

[3] DONGARRA, J., CROZ, J. D., HAMMARLIN, S., AND DUFF, I. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software 16*, 1 (Mar 1990), pp. 1–17.

[4] DONGARRA, J., GUSTAVSON, F. G., AND KARP, A. Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine. *SIAM Review 26*, 1 (Jan 1984), pp. 91–112.

[5] FOX, G., JOHNSON, M., LYZENGA, G., OTTO, S., SALMON, J., AND WALKER, D. *Solving Problems on Concurrent Processors.* Prentice Hall, New Jersey, 1988.

[6] FOX, G. C., KENNEDY, K., AND ET AL. Compiling Fortran 77D and 90D for MIMD Distributed Memory Machines. Tech. Rep. SCCS-251, CRCP#TR92203, Northeast Parallel Architectures Center at Syracuse University, Rice University, May 192.

[7] G. H. GOLUB, C. F. V. L. *Matrix Computations.* John Hopkins University Press, 1989.

[8] GEIST, G. A., HEATH, M. T., PEYTON, B. W., AND WORLEY, P. H. A User's Guide to PICL, A Portable Intrumented Communication Library. Tech. Rep. Tech. Rep. ORNL/TM-11616, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, Oct 1991.

[9] JACK DONGARRA, S. O. LAPACK Block Factorization Algorithms on the Intel iPSC/860. Tech. Rep. LAPACK Working Note 24, Department of Computer Science Technical Report, University of Tennessee, 1990.

[10] M. HEATH, J. A. E. Paragraph. Tech. rep., Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, Oct 1991.

[11] MOHAMED, A. G. Block-based Solvers for Engineering Applications. In *Mechanics Computing in the 1990's and Beyond, Proceedings of the ASCE Engineering Mechanics Speciality Conference* (May 1991), pp. 19–22.

[12] MOHAMED, A. G., FOX, G. C., AND VON LASZEWSKI, G. Blocked LU Factorization on a Multiprocessor Computer. Internal Report SCCS-94b, Northeast Parallel Architectures Center, Syracuse University, April 1992. appears in: Microcomputers in Civil Engineering.

[13] MOHAMED, A. G., AND VALENTINE, D. T. Taylor's Vortex Array: A New Test Problem for Navier-Stokes Solution Procedures. In *Solution of Superlarge Problems in Computational Mechanics* (1989), J. Kane, A. Carlson, and D. Cox, Eds., pp. 167–181.

[14] MOHAMED, A. G., AND VALENTINE, D. T. Numerical Predictions of Turbulent Flow in an Annular Pipe. In *Proceedings of the ASME International Computers in Engineering* (Aug 1990), vol. II, pp. 471–479.

[15] MOHAMED, A. G., VALENTINE, D. T., AND HESSEL, R. E. Numerical Study of Laminar Separation Over an Annular Backstep. *Computers & Fluids, Pergamon Press 20*, 2 (1991), pp. 121–143.

[16] NUGENT, S. F. The iPSC/2 Direct-Connect Technology. *Third Conference on Hypecube Concurrent Computers and Applications 1* (1988), pp. 51–60.