Blocked LU Factorization on a Multiprocessor Computer

A. Gaber Mohamed, Geoffrey C. Fox and Gregor von Laszewski
agm@npac.syr.edu

Contents

Blocked LU Factorization on a Multiprocessor Computer

A. Gaber Mohamed, Geoffrey C. Fox and Gregor von Laszewski

agm@npac.syr.edu

Abstract

This paper presents implementations of new methods solving **LU** factorization used in engineering applications. The implementations are done on the Alliant FX/80 minisupercomputer and use Level 3 Basic Linear Algebra Subprograms. Three ways of expressing the **LU** factorization in terms of blocked algorithms are considered.
The performance of the blocked algorithms, using the parallel vector facilities, are compared to a noblock algorithm using only subprograms of level 1 and 2 BLAS.

Keywords: LU factorization, blocked LU factorization, Alliant FX/80, BLAS 3.

1 Introduction

The importance of parabolic, elliptic partial differential equations (PDE) and ordinary differential equations (ODE) in engineering problems is well known. Most physical phenomena are modeled either by a system of PDEs or ODEs. Using a discretization technique like finite differences or finite elements a system of linear algebraic equations can be obtained. Even in nonlinear phenomena, one might solve a nonlinear system by iterating over the solution of a sequence of linear systems [6, 10, 11, 12].
Consider the solution of the dense system of linear equations,

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

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

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

then solving for y and x in two consecutive substitution steps:

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

Experimental results show that in programs for applications of the above described type, more than 50% of the CPU time is usually spent in matrix factorization. This occurs because

1. the computational effort to factorize the matrix $\mathbf{A}$ is higher than for the two substitution steps and the rest of the program.

2. most standard programming practices used in Fortran to factorize the matrix **A** result in more memory accesses than floating point operations. This cause the processor to be idle during the time data is being transferred from the 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 worth to optimize a factorization algorithm in such a way 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, a central processing unit (CPU) is much faster than the time used to move the data for a computation in the CPU. The process of moving the data is called *fetching* and the time used 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 of the data used for computation as small as possible. Unfortunately, it is too expensive to build very fast memories of sufficient capacity as it is necessary for scientific applications with thousands or even millions of variables. Therefore, a *memory hierarchy* is used to decrease the cost of the memory such that a cost efficient memory access time can be achieved. Figure 1 shows a typical memory hierarchy. The closer the memory level is to the registers of the processor the faster is the access.

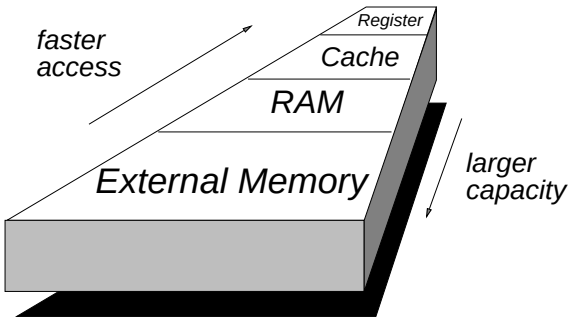

Figure 1: Typical memory hierarchy in a computer

For example, to use data stored in the external memory it has to pass through all components 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 referred. 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 and vectorization.

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 can be obtained by figuring out how the data is stored in the memory. One has to know that most memory organizations use specific strategies to reduce the memory access time. One rule which is common on many machines is to fetch not only one datum at a time but a block of data. In most cases the block is organized as a vector. The distance between two elements of a vector in the memory is called *stride*. It is best to organize the data in the memory in such a way that the algorithm access the data in unit stride (stride = 1).

Figure 2 and 3 show how data (a matrix) is stored in a memory using the programming languages C and FORTRAN. Having this in mind it is obvious why C is also called a row oriented programming language and 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, the above FORTRAN statement can be rewritten as

$$\begin{aligned} c_{i,j} \leftarrow c_{i,j} &+ a_{i,k} * b_{k,j} \\ &+ a_{i,k+1} * b_{k+1,j} \\ &\vdots \\ &+ a_{i,k+\alpha-1} * b_{k+\alpha-1,j} \end{aligned} \tag{4}$$

This leads to 2α floating point operations, 2 memory accesses for storing and fetching $c_{i,j}$, α for fetching the $a_{i,k}$'s and one for all $b_{k,j}$'s. The ratio is $r = \frac{2\alpha}{3+\alpha}$.

Storing the matrix $\mathbf{A}$ as its transpose one can rewrite the multiplication as

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

where $a^t_{k,i}$ specifies the element in the k-th row and i-th column of the transpose A^t.

Now there is only one memory access necessary to fetch even the vector a. Therefore, the ratio is $r = \frac{\alpha}{2}$. The prediction of a maximal vector length α depends on many factors: the memory hierarchy, the actually used machine, and its fetching algorithm. Algorithms which updates not

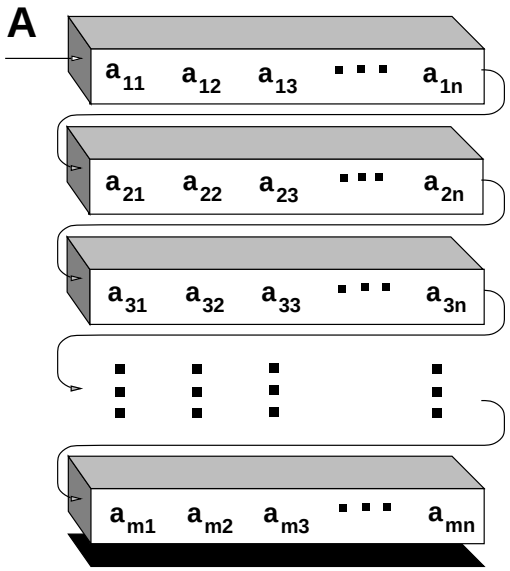

Figure 2: Storage of a two dimensional array in row oriented programming languages like C

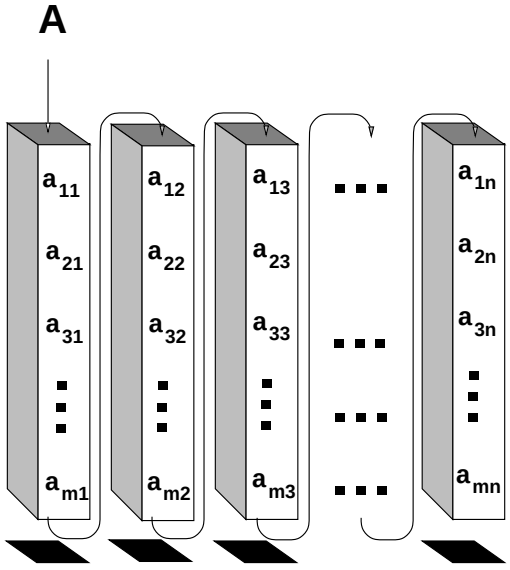

Figure 3: Storage of a two dimensional array in column oriented programming languages like FORTRAN

only one vector at a time but a block of contiguous vectors are known as *blocked algorithm*. This way the work is done *locally* on a block of data.

Previous numerical experiments [9] showed that traditional linear algebra algorithms do not achieve high performance on shared-memory multiprocessors because of lack of data locality. Therefore, data locality is the fundamental problem in parallel computing and has great influence on the performance of such machines. Use of block-based algorithms is one of the most efficient ways to improve the performance of shared memory machines.

Dongarra, Gustavson, and Karp [5] discussed six ways of implementing the **LU** factorization obtained by reordering the three nested loops that constitute the algorithm for these cases. Algorithm 1 explains the generic Gaussian elimination.

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

Algorithm 1: Gaussian Algorithm

The loop indices are i, j and k. Each of the six ways has a different order of the index variables. The three ways where loop i is lying inside loop j are the ways that access data in unit stride. Only these three unit stride ways are applicable to the column oriented FORTRAN. These three algorithms will be introduced later as block based algorithms with pivoting. Furthermore, the pivoting operation can be resembled by 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 the defined as $-\frac{a_{kj}}{a_{kk}}$.

2 Basic Linear Algebra Subprograms

In many applications, as seen above, vector and matrix operations can be used to formulate algorithmic solution to scientific problems. Programming languages like FORTRAN77 do not provide

this kind of operations. To make programming easier for a scientific software engineer it is desirable to use a library supporting this kind of routines.

A public domain set of **B**asic **L**inear **A**lgebra **S**ubprograms, called BLAS, has been very successful for scientific applications. Many algorithms and software packages make use of these programs [4]. At the end of this section it is shown which library routines are best for building an optimized algorithm on a shared memory multiprocessor.

Different levels of BLAS are distinguished by the amount of its *arithmetic complexity*. The complexity of programs in one level of BLAS is the same. 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 operations (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

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

These abbreviations are at first confusing, but the nomenclature of the BLAS programs is in fact very simple and give 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.

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. Therefore, it is obvious that the strategy is to maximize the use of level 3 BLAS.

3 Noblock Factorization using BLAS

Now a necessary basis has been established to formulate the factorization algorithms.

Abbreviation	stands for
M	**M**atrix
V	**V**ector
GE	**GE**neral
TR	**TR**iangular

Table 2: Abbreviations used in BLAS

Of the six ways of implementing **LU** factorization, using partial pivoting with row interchanges, that were discussed by Dongarra, Gustavson, and Karp [5], we describe the three column-oriented variants. These algorithms are suitable for an implementation in FORTRAN since the array structure is column-oriented.

The noblock factorization algorithms are the building blocks for the blocked 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 algorithms compared here are the jki-noblock algorithm and the jik-noblock algorithm. Since the number of memory touches for the kji noblock algorithm is twice as high as for the others[5], the running time for this algorithm is slower. Therefore, only the two algorithms with the same number of memory touches are compared. The abbreviation jik means that j is the loop index for the outermost loop and k for the inner most loop (compare to Algorithm 1).

Comparison between the two algorithms implemented on the Alliant showed that the jik version is faster than the jki version. Therefore, all blocked algorithms described in later sections are using the jik version. This version is also used in LAPACK [2]. LAPACK is a collection of public domain programs using level 3 BLAS to solve basic linear algebra problems. The name stands for **L**inear **A**lgebra **PACK**age.

3.1 jki-Noblock Algorithm

Before the algorithm is described in detail it is useful to visualize the data dependencies of the matrix elements in the $n \times n$ matrix between the computational steps. The dependencies are shown in figure 4. If a datum in the picture is higher than another then, this datum has to be calculated first. Following the data dependencies first $u^{(j)}$ is updated with the help of $L^{(j)}$. Next, $l^{(j)}$ is calculated with the help of $u^{(j)}$ and $A^{(j)}$. Then the matrix element l_{jj} is determined and $l^{(j)}$ is updated. Therefore at time step j only the j-th column is updated. This steps are executed over

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

all columns in the matrix once.

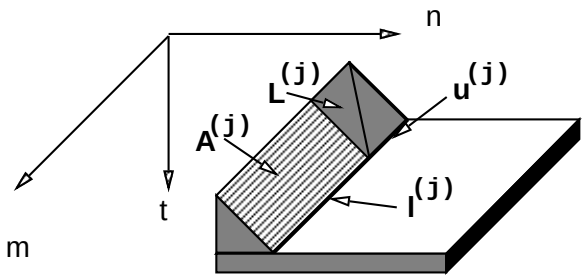

Figure 4: jki-noblock

The mathematical description of the algorithm is as follows:
For simplicity the algorithms are described for $n \times n$ matrices. Let $\mathbf{A}$ be the original $n \times n$ matrix and $\mathbf{L}$, the unit lower triangular matrix and $\mathbf{U}$ the upper triangular matrices after the factorization. Let $l^{(j)}$ and $u^{(j)}$ represent the j-th *column* vector of the matrix $\mathbf{L}$ and $\mathbf{U}$. The matrix $\mathbf{A}^{(j)}$ specifies a submatrix of $\mathbf{A}$. It includes all elements from the first column to the column $j-1$ and from the rows $j+1$ to n.
In each iteration one *column* of $\mathbf{U}$ and $\mathbf{L}$ is updated at each iteration step. The decomposition is generated in the following way:

Initialization:

$j \leftarrow 1$

Compute column $\mathbf{u^{(j)}}$:

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

Update $\mathbf{l^{(j)}}$:

$l^{(j)} \leftarrow l^{(j)} - \mathbf{A}^{(j)} u^{(j)}$

Select pivot and exchange:

$$\mathrm{p} \leftarrow j + \min\left\{ k \left| |l_k| = \max_{1 \le i \le n-j+1} \{|l_i^{(j)}|\} \right. \right\} - 1$$

exchange row j and row p

Scaling:

$$l_i^{(j)} \leftarrow l_i^{(j)} / a_{p,j} \qquad \forall 1 \le i \le n-j+1$$

Iterate:

$j \leftarrow j + 1$

goto **Compute column $\mathbf{u^{(j)}}$**

Using the BLAS level 2 routines the algorithm 2 can be obtained for the jki-noblock version.

subroutine jki-noblock $(m, n, A, ipiv, lda)$
do $j = 1, n$
1. Apply previous interchanges to j-th column.
2. Compute elements $1 : j-1$ *of j-th column.*
TRSV('Lower','No transpose','Unit',$j-1, a, lda, a_{(1,j)}, 1$)
3. Update elements $j : m$ *of j-th column.*
GEMV('No transpose',$m-j+1, j-1, -1, a_{(j,1)}, lda, a_{(1,j)}, 1, 1, a_{(j,j)}, 1$)
4. Find pivot
$p = j - 1 +$ IAMAX$(m-j+1, a_{(j,j)}, 1)$
$ipiv(j) = p$
5. Apply interchange to columns 1:j.
SWAP$(j, a_{(j,1)}, lda, a_{(p,1)}, lda)$
if $j = m$ then stop
6. Compute elements j+1:m of j-th column.
SCAL$(m-j, 1/a_{(j,j)}, a_{(j+1,j)}, 1)$
end do

Algorithm 2: jki-noblock

3.2 jik-Noblock Algorithm

In contrast to the jki-noblock algorithm the jik-noblock algorithm updates one *column* of **L** and one *row* of **U**. This noblock algorithm is also used in LAPACK to decompose a matrix and is called there GETF2. The jik-noblock algorithm is also known as Crout's Algorithm.
The data dependencies are shown in the figure 5. First the vector $l^{(j)}$ is updated with the help of $A^{(j)}$ and the vector $u^{(j)}$. The element of l_{jj} is computed and $u^{(j)}$ is updated with the help of $U^{(j)}$ and $l_r^{(j)}$.
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**. The matrix $\mathbf{A}^{(j)}$ specifies a submatrix of **A**. It 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**. It includes all elements from the first row to the row $k-1$ and from the columns $j+1$ to n.

Initialization:

$j \leftarrow 1$

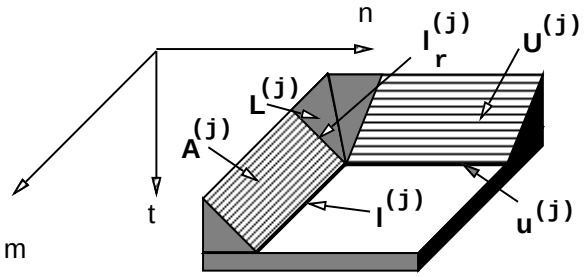

Figure 5: jik-noblock

Update $\mathbf{l^{(j)}}$:

$$l^{(j)} \leftarrow l^{(j)} - \mathbf{A}^{(j)} u^{(j)}$$

Select pivot and exchange:

$$\mathrm{p} \leftarrow j + \min\left\{ j \middle| |l_j| = \max_{1 \le i \le n-j+1} \{|l_i^{(j)}|\} \right\} - 1$$

exchange row j and row p

Scaling:

$$l_i^{(j)} \leftarrow l_i^{(j)} / a_{p,j}$$

if $(j = n)$ then stop

Compute row $\mathbf{u^{(j)}}$:

$$u^{(j)} \leftarrow u^{(j)} - (U^{(j)})^{-1} l_r^{(j)}$$

Iterate:

$$k \leftarrow j + 1$$

goto **update $\mathbf{l^{(j)}}$**

The code using level 2 BLAS subprograms is shown in algorithm 3.
Our numerical experiments on the Alliant showed that the jik-noblock's performance is superior to the jki-noblock's performance. Therefore, the jik-noblock subroutine from LAPACK is used to implement the block factorizing algorithms shown in a later section.

3.3 kji-Noblock Algorithm

The kji-noblock algorithm updates one *column* of **L**, one *row* of **U**, and the *remainder* of the matrix **A** as shown in figure 6 and 7.
The data dependencies are shown in the figure 6. For the updating of the rest of the matrix **A** only the vectors shown in figure 7 are needed. First the vector $l^{(k)}$ is factorized. Then vector $u^{(k)}$

subroutine jik-noblock$(m, n, A, ipiv, lda)$
do $j = 1, n$
1. update diagonal and subdiagonal elements
in column j
GEMV ('no transpose',$m - j + 1, j - 1, -1, a_{(j,1)}, lda, a_{(1,j)}, 1, 1, a_{(j,j)}, 1)$
2. find pivot
$jp = j - 1$+IDAMAX $(m - j + 1, a_{(j,j)}, 1)$
$ipiv(j) = jp$
3. apply interchange to columns 1:j
SWAP $(m, a_{(j,1)}, lda, a_{(jp,1)}, lda)$
if(j.lt.n)
4. compute elements j+1:m of j-th column
SCAL $(m - j, 1/a_{(j,j)}, a_{(j+1,j)}, 1)$
5. compute block row of u
GEMV ('transpose',$j - 1, m - j, -1, a_{(1,j+1)}, lda, a_{(j,1)}, lda, 1, a_{(j,j+1)}, lda)$
end if
end do

Algorithm 3: jik-noblock

is computed with the help of the factorized vector $l^{(k)}$. Now the Matrix remainder is updated with th help of $l^{(k)}$ and $u^{(k)}$.
Let $l^{(k)}$ represent the k-th column vector of the matrix $\mathbf{L}$ and $u^{(k)}$ the k-th row vector starting from a_{kk} of the matrix $\mathbf{A}$. The matrix $\mathbf{R}^{(k)}$ specifies a submatrix of $\mathbf{A}$. It includes all elements from the first column k+1 to n and from the rows $k + 1$ to n.

Initialization:

$k \leftarrow 1$

Select pivot and exchange:

$$\text{p} \leftarrow k + \min\left\{k \middle| |l_k| = \max_{1 \le i \le n-k+1} \{|l_i^{(k)}|\}\right\} - 1$$

exchange row k and row p

Scaling:

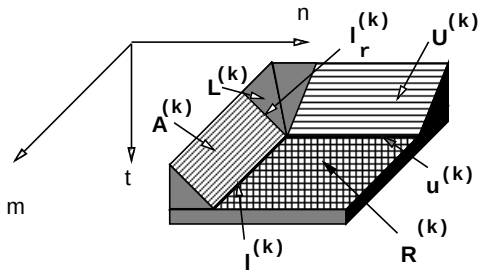

Figure 6: kji-noblock

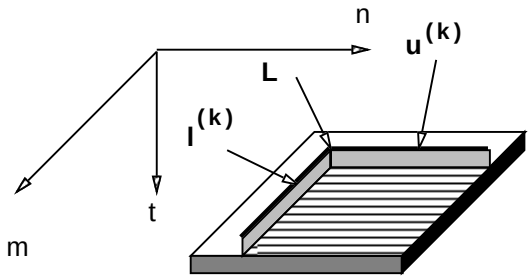

Figure 7: kji-noblock

$l_i^{(k)} \leftarrow l_i^{(k)}/a_{p,k}$
if $(k = n)$ then stop

Compute row $\mathbf{u^{(k)}}$:

$u^{(k)} \leftarrow u^{(k)} - (U^{(k)})^{-1} l_r^{(k)}$

Update $\mathbf{R^{(k)}}$:

$R^{(k)} \leftarrow R^{(k)} - \mathbf{L}u^{(k)}$

Iterate:

$k \leftarrow k + 1$
goto **update $\mathbf{l^{(k)}}$**

The code using level 2 BLAS subprograms is shown in algorithm 4.

subroutine kji-noblock $(m, n, A, ipiv, lda)$

do $k = 1, n$

1. find pivot

$jp = j - 1$+IDAMAX $(m - j + 1, a_{(j,j)}, 1)$

$ipiv(j) = jp$

2. compute elements $j + 1 : m$ of j-th column

SCAL $(m - j, 1/a_{(j,j)}, a_{(j+1,j)}, 1)$

3. apply interchange to columns $j + 1 : m$

SWAP $(n, a_{(j+1,j)}, lda, a_{(jp,j)}, lda)$

4. compute block row of u

GEMV ('transpose',$j - 1, m - j, -1, a_{(1,j+1)}, lda, a_{(j,1)}, lda, 1, a_{(j,j+1)}, lda)$

5. compute remainder matrix

GEMM ('notranspose','notranspose',$m - j, n - j, 1, 1, a_{(j+1,j)}$,
$lda, a_{(j,j+1)}, lda, 1, a_{(j+1,j+1)}, lda)$

end do

Algorithm 4: kji-noblock

4 Naming convention

In literature the algorithms are often named by the basic operations inherent in the last levels of the execution of the algorithms. 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 *S*calar multiplied by the vector *X P*lus the vector *Y* is called *SAXPY*. The basic operation of the jki algorithm is based on the following operation:

$$\vec{z} \leftarrow \mathbf{A}\vec{x} + \vec{y},$$

where $\mathbf{A}$ 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. The name *GAXPY* is often used. The basic operation of the jik algorithm is based on the dot product:

$$\vec{z} \leftarrow \vec{x}^T\vec{y},$$

where $\vec{x}, \vec{y}, \vec{z}$ are vectors. Therefore, it is called SDOT.

5 Block Factorization using BLAS

The algorithms described above are modified in such a way that not only one row vector or column is updated but a block of vectors.
In all cases work is done on blocks with β columns using a matrix-vector based elimination scheme to reduce each block column in turn. In each of the three block-column variants pivoting is performed only within a noblock algorithm. Any permutation resulting from this pivoting must be applied to the remainder of the matrix.

5.1 Block jik-SDOT

At the jth iteration step of the elimination process the jik-SDOT algorithm computes one block column of **L** and one block row of **U**. These computations require the operations shown in figure 8[3]:

0. Initialize: start with the first block

$j \leftarrow 1$

1. Update $C^{(j)}$:

the jth diagonal and subdiagonal blocks of C are computed using GEMM.
$C^{(j)} \leftarrow C^{(j)} - A^{(j)}B^{(j)}$

2. Factorize $C^{(j)}$ and Pivot:

the jth block column is factorized into **LU** factors using the jik-noblock algorithm
apply previous interchanges to the previous blocks $A_1^{(j)}$ and $U_2^{(j)}$.

3. Update $U_2^{(j)}$:

a) jth block row of **U** is updated using GEMM.

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

b) j-th block row of **U** is calculated using TRSM.

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

4. Iterate:

if no more blocks then stop
else goto [Update $C^{(j)}$:]

5.2 Block jki-GAXPY

In the jki-GAXPY algorithm, at the jth step of the elimination, a block column of both matrices **L** and **U** is computed. These computations require the following operations:

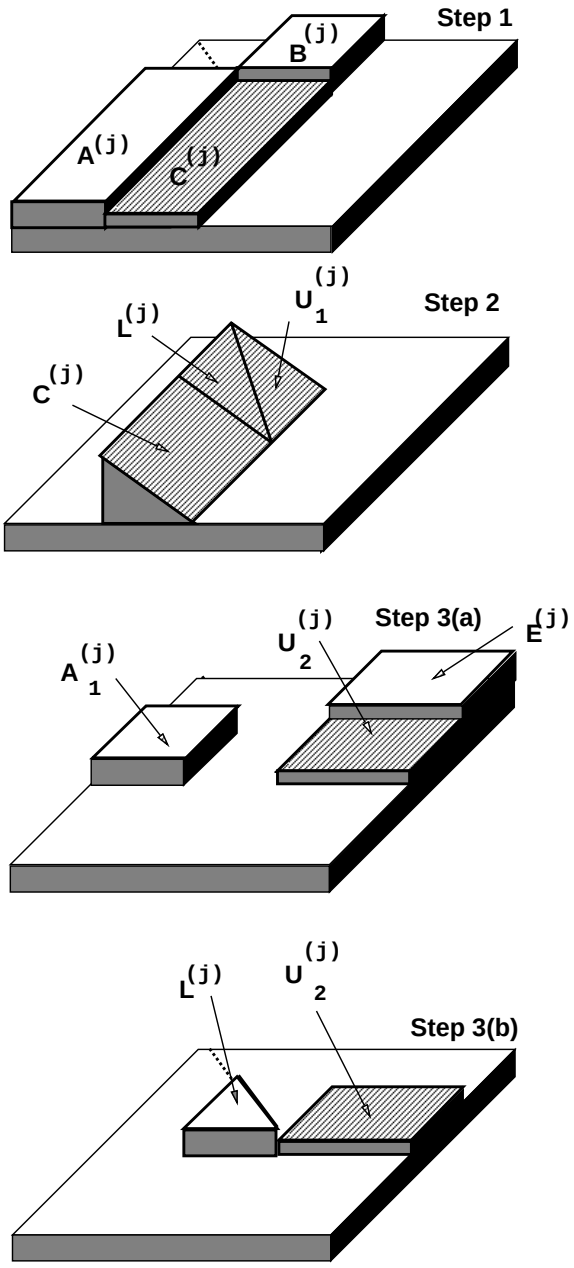

Figure 8: *jik*-SDOT

subroutine jik-block$(m, n, A, lda, ipiv, \beta)$
 if $\beta = 1$ then
  call jik-noblock$(m, n, A, lda, ipiv)$
 else
  do $j = 1, min(m, n), beta$
   $jb = min(min(m, n) - j + 1, \beta)$
    ▷ *update diagonal and subdiagonal blocks.*
     call gemm('no transpose','no transpose',$m - j + 1, jb, j - 1,$
      $-1, a_{(j,1)}, lda, a_{(1,j)}, lda, 1,$
      $a_{(j,j)}, lda)$
    ▷ *factorize diagonal and subdiagonal blocks and test for singularity.*
     call jik-noblock$(m - j + 1, jb, a_{(j,j)}, lda, ipiv(j))$
    ▷ *update pivot indices and apply the interchanges to the columns on either side of the current block.*
     do $i = j, min(m, j + jb - 1)$
      $ipiv(i) = j - 1 + ipiv(i)$
     end do
     call laswp$(j - 1, a, lda, j, j + jb - 1, ipiv, 1)$
     call laswp$(n - j - jb + 1, a_{(1,j+jb)}, lda, j, j + jb - 1, ipiv, 1)$
   if $j + jb \leq n$ then
    ▷ *compute block row of u.*
     call gemm('no transpose','no transpose',$jb, n - j - jb + 1,$
      $j - 1, -1, a_{(j,1)}, lda, a_{(1,j+jb)}, lda,$
      $1, a_{(j,j+jb)}, lda)$
     call trsm('left','lower','no transpose','unit',$jb,$
      $n - j - jb + 1, 1, a_{(j,j)}, lda, a_{(j,j+jb)}, lda)$
   end if
  end do
end if

Algorithm 5: jik-SDOT

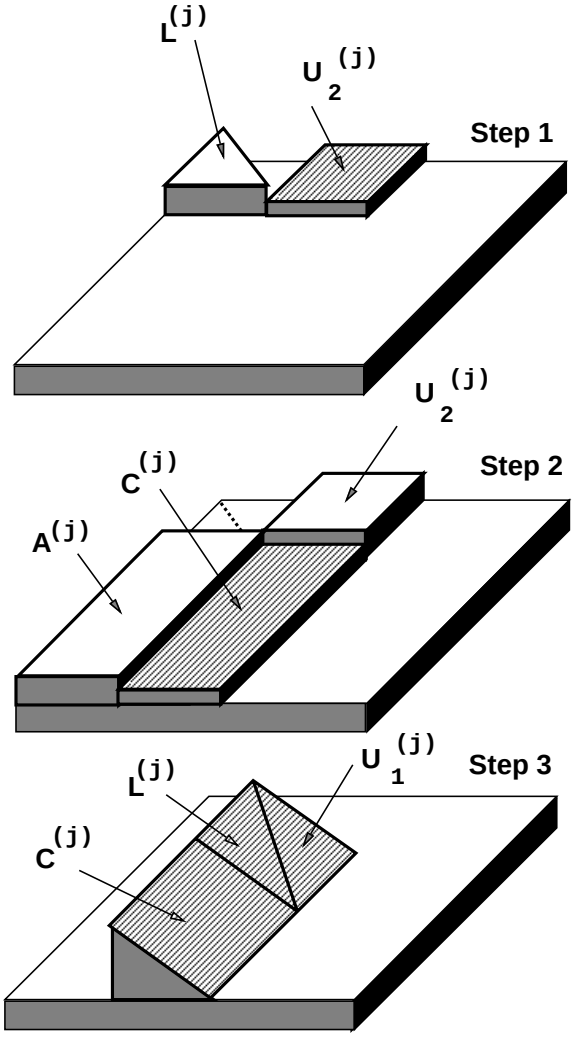

Figure 9: *jki*-GAXPY

subroutine jki-block$(m, n, A, ipiv, \beta, lda)$
 if $\beta = 1$ then
  call jik-noblock(m,n,A,ipiv,lda)
 else
 do $j = 1, n, \beta$
  $jb = \min(n - j + 1, \beta)$
   ▷ *Apply previous interchanges to current block.*
   call laswp$(jb, a_{(1j)}, lda, 1, j - 1, ipiv, 1)$
   ▷ *Apply previous transformations to current block of U.*
   do $k = 1, j - 1, \beta$
    $kb = \min(\beta, j - k)$
    call trsm('left','lower','no transpose','unit',kb,
     $jb, 1, a_{(kk)}, lda, a_{(kj)}, lda)$
    call gemm('no transpose','no transpose',$j - k - kb, jb$,
     $kb, -1, a_{(k+kb,k)}, lda, a_{(kj)}, lda$,
     $1, a_{(k+kb,j)}, lda)$
   end do
   ▷ *Update diagonal and subdiagonal blocks.*
   call gemm('no transpose','no transpose',$m - j + 1, jb, j - 1$,
    $-1, a_{(j1)}, lda, a_{(1j)}, lda, 1$,
    $a_{(jj)}, lda)$
   ▷ *factorize diagonal and subdiagonal blocks*
   call jik-noblock $(m - j + 1, jb, a_{(jj)}, ipiv(j), lda)$
   ▷ *update pivot indices and apply the interchanges to the*
   columns on either side of the current block.
   do $i = j, min(m, j + jb - 1)$
    $ipiv(i) = j - 1 + ipiv(i)$
   end do
   call slaswp$(j - 1, a, lda, j, j + jb - 1, ipiv, 1)$
 end do
end if

Algorithm 6: jki-GAXPY

0. Initialize:

start with first block.

$j \leftarrow 1$

1. Pivot and Update $U_2^{(j)}$:

apply previous interchanges to the current block $U_2^{(j)}$.

the j-th suberdiagonal block of U is computed using TRSM. $U_2^{(j)} \leftarrow (L^{(j)})^{-1} U_2^{(j)}$

2. Update $C^{(j)}$:

the jth 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 jth block column is factorized into **LU** factors using a noblocked algorithm.

4. Iterate:

if no more blocks then stop
else goto [Pivot and Update $U_2^{(j)}$:]

5.3 Block kji-SAXPY

At the kth step of the factorization process a block column of **L** and a block row of **U** are computed and the corresponding transformations are applied to the remaining part of the matrix. This algorithm requires the following operations:

0. Initialize:

start with first block.

$k \leftarrow 1$

1. Factorize $C^{(k)}$:

the kth block column is factorized into **LU** factors using the noblock algorithm.

apply previous interchanges to the previous blocks $A_1^{(k)}$ and $U_2^{(k)}$.

2. Update $U_2^{(k)}$:

Computing the kth block row of **U** using TRSM.

$U_2^{(k)} = L^{(k)} U_2^{(k)}$

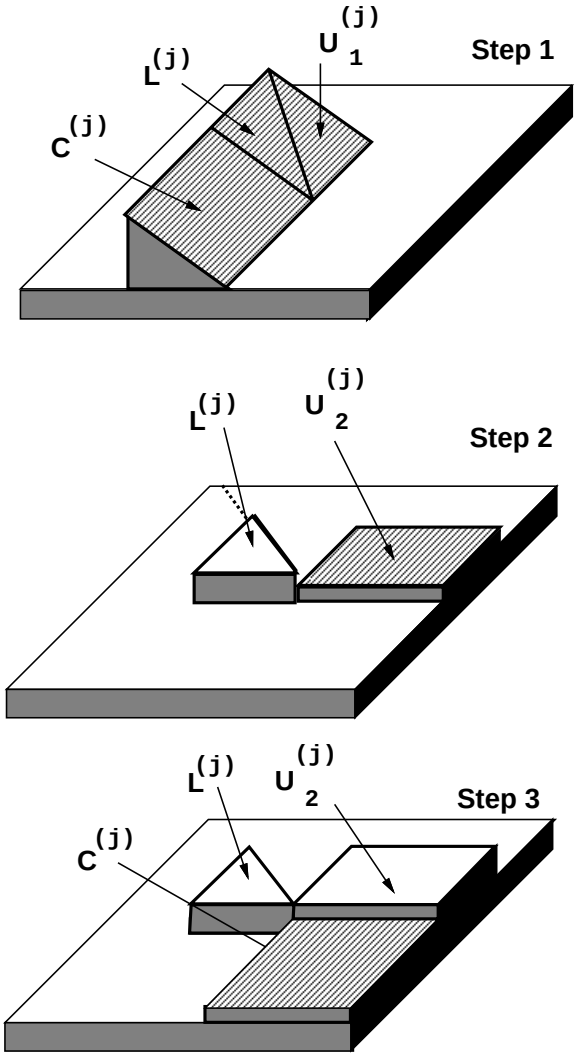

Figure 10: kji-SAXPY

subroutine kji-block$(m, n, A, lda, ipiv, \beta)$
if $\beta = 1$ then
  call jik-noblock$(m, n, A, lda, ipiv)$
else
  do $k = 1, \min(m, n), \beta$
    $kb = \min(\min(m, n) - k + 1, \beta)$
    ▷ *factorize diagonal and subdiagonal blocks*
      call jik-noblock$(m - k + 1, kb, a_{(k,k)}, lda, ipiv(k), info)$
    ▷ *update pivot indices and apply the interchanges to*
    previous and following blocks of the current block.
      do $i = k, min(m, k + kb - 1)$
        $ipiv(i) = k - 1 + ipiv(wi)$
      end do
      call laswp$(k - 1, a, lda, k, k + kb - 1, ipiv, 1)$
      call laswp$(n - k - kb + 1, a_{(1,k+kb)}, lda, k, k + kb - 1, ipiv, 1)$
    ▷ *Compute block row of U*
      call trsm('left','lower','no transpose','unit',kb,
            $n - k - kb + 1, one, a_{(k,k)}, lda, a_{(k,k+kb)}, lda)$
    ▷ *Update right-hand bottom,(n-k-kb+1) X (n-k-kb+1) block*
      call gemm('no transpose','no transpose',$n - k - kb + 1$,
            $n - k - kb + 1, kb, -one, a_{(k+kb,k)}, lda,$
            $a_{(k,k+kb)}, lda, one, a_{(k+kb,k+kb)}, lda)$
  end do
end if
return

Algorithm 7: kji-SAXPY

3. Update $C^{(k)}$:

Updating the remaining matrix using a block outer product using GEMM.

$C^{(k)} = C^{(k)} - L^{(k)}U_2^{(k)}$

4. Iterate:

if no more blocks then stop

else goto **Factorize** $C^{(k)}$

6 Arithmetic Complexity

All three blocked algorithms have the same computational complexity of approximately

$$\frac{2n^3}{3}$$

floating point operations. The difference in the execution time of the algorithms is therefore caused by the different data access patterns and data locality.

7 Experimental Results

The Experiments are done on the Alliant FX/80. The FX/80 is a shared memory parallel computer with six interactive processors (IPs) and eight pipelined advanced computational elements, called *ACE*s. Each ACE contains eight 64-bit floating-point registers 32 elements long. The total number of attached ACEs are also called a *complex*. A concurrency control bus connects the eight ACEs and acts as a synchronization facility. The ACEs share a 512k byte write-back cache. The cache is connected to memory by a memory bus [1].
In all cases, the assembly-coded BLAS routines of level 1, 2, and 3 from the Alliant scientific library are used, as well as some routines from LAPACK. All experiments are done in double precision (64 bit). The specific BLAS routines are shown in table 3.
The LAPACK routines include

- *jik*-noblock (GETF2) for partial pivoting with row interchanges,
- LASWP to perform a series of row interchanges on a block of a matrix **A** based on the SWAP routine from the Alliant scientific library,
- and GETRS for solving the system $\mathbf{A}x = b$ after the matrix **A** is **LU** factorized by any of the three block-**LU** factorization routines. This algorithm is known as back substitution.

The performance is measured in MFLOPS (**M**illion **FL**oating-point **O**perations **P**er **S**econd) for solving the whole system $\mathbf{A}x = b$, *i.e.*, factorizing by any of the three blocked algorithms and

solution by GETRS. The performance is measured on a stand-alone basis[1] for matrices of size $N \in \{100, 200, ..., 1000\}$ on 8 attached ACEs.
The programs are compiled with the optimization option of the fortran 77 compiler (-O -DAS).
The implemented algorithm uses the noblock algorithm if $\beta = 1$ and the blocked version if $\beta > 1$. Our numerical experiments of Crout's method from LAPACK, after necessary modifications to use the FX/BLAS, showed that the performance for $\beta = 1$ is higher than that for $\beta = 32$ if the matrix size is $N \leq 500$, as shown in figure 11.

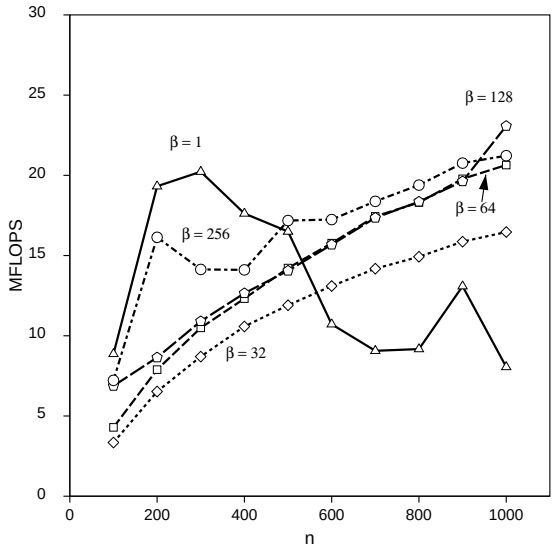

Figure 11: Performance of blocked jik-SDOT for different block sizes (β)

Using Level 2 BLAS (GEMV and TRSV) the ratio of operations to data movement is not very high which is, as shown before, important to achieve high performance when exploiting concurrency and vectorization. However, parallelization at this level can provide a reasonable performance improvement when efficient parallelization tools for fine granularity (especially low-cost synchronization) are available. The hardware-controlled microtasking provided by the Alliant enables this. It is the hardware concurrency control in the Alliant FX/80 that explains why the assembly-coded Level 2 BLAS can provide a reasonable performance on this machine.
On the other hand, Dayde and Duff [3] reported that their numerical experiments, on the CRAY-2, Cyber 205, and IBM 3090-200/VF showed that the jik-SDOT version was uniformly the worst. This is because much of the updating is done on a block row of **U** where the vector lengths are the same as that of the block size. Our results also show that jik-SDOT is penalized by the short vector

[1] No time sharing with other users.

length inherent in this algorithm. This is confirmed by our results shown in Figure 11. It is clear from Figure 11 that the performance improves as the block size increases. The best performance is obtained for the block size equivalent to the total number of computational elements in the hardware configuration of the Alliant system, called *complex*. Since the number of ACEs used is 8, a multiplicator of 256 elements is obtained.

Figures 12-15 show the performance comparison for the three column-oriented algorithms jik-SDOT, *JKI*-GAXPY, and kji-SAXPY for a block size of 32, 64, 128, and 256 on a complex size of 8 and stand-alone timing with the performance of the jik-noblock algorithm ($\beta = 1$ and using Level 1 and 2 BLAS from the Alliant library). kji-SAXPY achieves the best performance for all block sizes. The highest performance is obtained for a block size of 64, and this is in agreement with the experimental results reported by Gallivan, Jalby, Meier, and Sameh [8]. Although Dongarra, Gustavson, and Karp [5] stressed the different access patterns of these three algorithms; for example, kji-SAXPY requires about twice as many transfers to memory as jik-SDOT and jki-GAXPY, we do not see the effect of this in our results. This is because memory and cache management mechanisms mask such differences.

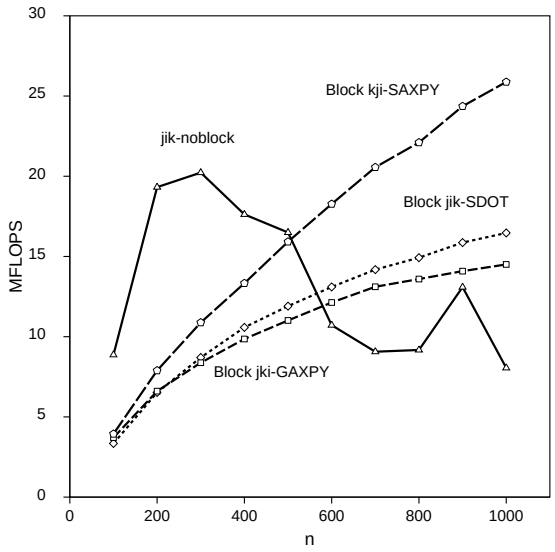

Figure 12: Performance of different blocked algorithm for $\beta = 32$ and jik-noblock

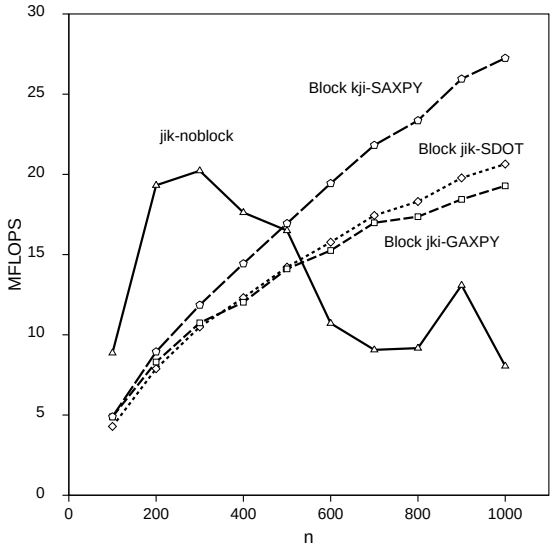

Figure 13: Performance of different blocked algorithm for $\beta = 64$ and jik-noblock

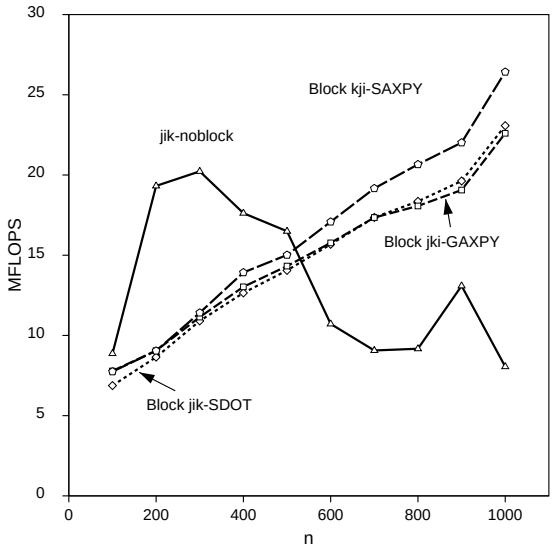

Figure 14: Performance of different blocked algorithm for $\beta = 128$ and jik-noblock

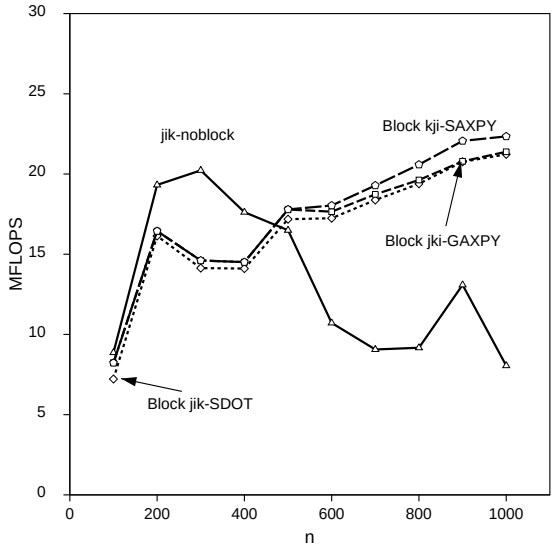

Figure 15: Performance of different blocked algorithm for $\beta = 256$ and jik-noblock

8 Conclusion

We have described the Fortran-oriented methods for block **LU** factorization on a shared memory parallel vector minisupercomputer. These methods are also ready for portable implementations on other shared memory parallel vector computers. Our numerical experiments and performance comparisons showed the following:

- The block jik-SDOT algorithm is very poor when block size is small due to the fact that most vector lengths in this algorithm are the same size as the block.
- The block kji-SAXPY algorithm's performance is superior to all other blocked algorithms for any block size.
- The jik-noblock algorithm's performance is superior to all blocked algorithms for a matrix of size $N \leq 500$. This is due to the built-in hardware concurrency control in the Alliant FX/80.

We recommend blocked **LU** factorization parallelism over the assembly-coded Level 3 BLAS for sufficiently large problems. Efficient utilization of the hardware and assembly-coded BLAS Level 1 and Level 2 should be used for small problems.

9 Future

Currently, we are testing the three blocked **LU** factorization algorithms in different FORTRAN dialects on all kinds of parallel machines. We already have implementations for SIMD and distributed memory MIMD machines as well.
As target machines the Intel iPSC/860 [7], nCube, Decmpp 12000, CM2, and the CM5 are used. This is being done as a part of our effort to develop a benchmark set for FORTRAND and the proposed HPF (High Performance Fortran).
To obtain a copy of all the software used in this study, send a one-line e-mail message "send index" to npaclib@minerva.npac.syr.edu or use anonymous ftp from minerva.npac.syr.edu. Npaclib 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@npac.syr.edu.

Acknowledgement

This work 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.

References

[1] *Architecture Manual.* Alliant Computer Systems Corporation, Littleton, MA, 1986.

[2] Anderson, E., Bai, Z., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. *Preliminary LAPACK Users' Guide.* Netlib, Oak Ridge National Laboratory, 1991.

[3] 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, Massachusetts Institute of Technology Vol. 3*, No. 2 (1989), pp. 40–70.

[4] 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.

[5] 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.

[6] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D. *Solving Problems on Concurrent Processors.* Prentice Hall, New Jersey, 1988.

[7] Fox, G. C., Mohamed, A. G., von Lasewski, G., Parashar, M., Lin, N.-T., and Yeh, N. High Performance Scalable Matrix Algebra Algorithms for Distributed Memory Architectures. Tech. Rep. SCCS-271, Northeast Parallel Architectures Center, Syracuse University, New York, April 1992. and CRPC-TR92210, Center for Research on Parallel Computation, Rice University, Houston, Texas.

[8] Gallivan, K., Jalby, W., Meier, U., and Sameh, A. Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design. *The International Journal of Supercomputer Applications 2*, 1 (Spring 1988), pp. 12–48.

[9] 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* (Columbus, Ohio, May 1991), H. Adeli and R. L. Sierakowski, Eds., ASCE, New York, pp. 19–22.

[10] 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* (Plenum, New York, NY, 1989), J. Kane, A. Carlson, and D. Cox, Eds., pp. 167–181.

[11] 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* (Boston, Massachusetts, Aug 1990), G. Kinzel and S. Rohde, Eds., vol. II, ASME, New York, pp. 471–479.

[12] 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.