Submitted to the proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing, editor D. C. Sorensen

# DESIGN AND EVALUATION OF PARALLEL BLOCK ALGORITHMS: LU FACTORIZATION ON AN IBM 3090 VF/600J

KRISTER DACKLAND, ERIK ELMROTH, BO KÅGSTRÖM, CHARLES VAN LOAN *
INSTITUTE OF INFORMATION PROCESSING
UNIVERSITY OF UMEÅ
S-901 87 UMEÅ, SWEDEN

**Abstract.** Explicitly parallel block algorithms for the LU factorization with partial pivoting which utilize optimized uniprocessor level-3 BLAS are compared with the corresponding routines from LAPACK (presently under development). Parallelism is mainly invoked implicitly in LAPACK by replacing calls to uniprocessor level-3 kernels by calls to parallel level-3 kernels, thereby maintaining portability. However, by parallelizing at the block level (with explicit parallelism) it is possible to overlap and pipeline different matrix-matrix operations and thereby gain some further performance. Load balancing of the explicitly parallel block algorithms is by either static or dynamic scheduling of work. Static scheduling utilizes cost functions based on the number of flops expressed as GEMM equivalents. The load balancing of the implicit approach is inherited from the parallel level-3 kernels. The implementations are done in IBM Parallel Fortran and performance results for an IBM 3090 VF/600J are presented. Theoretical models give upper bounds on the best possible speed-up of the explicitly and implicitly parallel block algorithms for the target machine.

**1. Introduction.** Different matrix factorizations are basic and important tools in most scientific, economic and engineering computational problems. Today it is well-known that block algorithms are required to exploit the full potential of hierarchical memory computers and multiprocessors. Our goal is to design parallel block algorithms with high parallel efficiency for different matrix factorizations and that still are transportable over a range of parallel architectures. In this paper we focus on the *LU* factorization with partial pivoting of a general matrix. Similar parallel block algorithms for the Cholesky and QR factorizations are presented and evaluated in [5].

Our contribution builds on the work of many other research groups (see e.g. [9], [10], [6], [18], [1]). The single most important source of inspiration is the LAPACK-project [1], [2], [3] whose goal is to design and implement a portable linear algebra library in Fortran 77 for efficient use on a variety of high-performance computers. LAPACK is based on block algorithms and therefore especially designed to utilize the level-3 BLAS [8] as the major computational kernels.

Although, we are using the vector multiprocessor IBM 3090 VF/600J [4] as our target architecture and IBM Parallel Fortran [20] as our implementation language the parallel algorithms described here could also be implemented on similar architectures like Alliant and Cray. The parallel language extensions of IBM Parallel Fortran [20] that we are using are included in the Parallel Computing Forum (PCF) proposal [17]. Furthermore, the data partitioning, the inherent overlapping and pipelining at the block level, and the description of the parallel block algorithms as node programs also make them suitable for implementa-

* DEPARTMENT OF COMPUTER SCIENCE, CORNELL UNIVERSITY, ITHACA, NEW YORK 14853-7501

tion as distributed ring-oriented block algorithms with column block-wrap mapping of the matrix to be factorized.

The outline of the rest of the paper is as follows. Section 2 describes block algorithms for the $LU$ factorization. In section 3 implicit and explicit parallelism is introduced and the parallel block algorithms implemented for $LU$ factorization are described. Explicit parallelization and static load balancing is the topic of section 4. Section 5 presents an explicitly parallel block algorithm with dynamic load balancing. Performance results of the parallel block algorithms discussed in sections 3-5 are presented and evaluated in section 6. Finally, in section 7 we present some conclusions.

**2. Block LU Algorithms.** The $LU$ factorization of an $m \times n$ matrix $A$ is given by

$$PA = LU$$

where the $m \times n$ matrix $L$ is unit lower trapezoidal, the $n \times n$ matrix $U$ is upper triangular and $P$ is a permutation matrix corresponding to row-interchanges of $A$ [12]. See also [6].

We have considered three column-oriented block algorithms (right looking, left looking and Crout) which easily can be derived from the following block-matrix equality

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

where we have ignored the permutations required by partial pivoting. They have in common that a block column of the matrix is $LU$ factorized by a level-2 routine. Then a part of $U$ is computed by solving a triangular system with multiple right hand sides (using the level-3 BLAS routine `DTRSM`). Finally, a part of the matrix is updated by a rank-$k$ update performed by the level-3 BLAS routine `DGEMM` (the size of $k$ is different for the algorithms). They differ in the order in which they perform the above operations, which also affects the way matrix elements are referenced. $L$ and $U$ overwrite $A$, i.e., no extra storage is required by the block algorithms.

Performance results on an IBM 3090 VF show that the block right looking algorithm performs best and it is described below. More detailed descriptions of all three variants can be found in [5].

**2.1. LU - Block Right Looking.** The block right looking algorithm, which also is called the block KJI or block SAXPY algorithm, refers mainly to data on the right hand side of the current block column. We illustrate the main operations of the algorithm by referring to the $3 \times 3$ block matrix above. First, the $LU$ factorization of the first block column of A is computed with a left looking level-2 routine, giving $L_{11}, L_{21}, L_{31}$ and $U_{11}$. Then $U_{12}$ and $U_{13}$ are obtained by solving $L_{11}[U_{12}, U_{13}] = [A_{12}, A_{13}]$. The remaining part of $A$ is updated with respect to the first block column of $L$ and block row of $U$:

$$\begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix} - \begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix} \cdot \begin{bmatrix} U_{12} & U_{13} \end{bmatrix}$$

This completes one step of the block algorithm and the same operations are now applied to the recently updated submatrix of $A$ until the factorization is completed.

**2.2. Pivotings.** It is well-known that, on the IBM 3090 VF, the row swaps on a block are most efficiently done if all permutations are applied to one column at a time (e.g. see [2]). This means that the level-1 BLAS routine `DSWAP` should *not* be used. We have also found it advantageous in the block right looking algorithm to delay the application of pivotings on the left hand side of the factorized block until the end of the computations. This results in the possibility of doing all swaps on each completed column in one iteration (implemented in the explicitly parallel algorithms, discussed in sections 3-5).

**3. Parallel Block Matrix Algorithms.** Parallelism in block algorithms can be invoked in several ways and we distinguish between *implicit* and *explicit* parallelism. The easiest way to introduce parallelism is to replace calls to uniprocessor level-3 kernels by calls to parallel level-3 kernels. Since the parallelism is invoked inside (within) the level-3 kernels, we refer to this type of block algorithms as *implicitly parallel.* Provided that efficient parallel level-3 kernels exist, this approach makes it easy to write portable programs for parallel architectures and this is the approach taken in the LAPACK project [1]. However, the implicit parallelism imposes a synchronization after each level-3 operation, which means that idle processors must wait until all processors have completed their share of work. Of course, this can lead to an unnecessary degradation of performance. The amount of degradation is determined by the efficiency of the parallel level-3 kernels and the data dependency of the block algorithm.

The synchronization discussed above is not algorithm dependent, and can therefore be overcome by invoking the parallelism at the block level in the algorithm. Processors can then work on independent block operations thus making it possible to overlap and pipeline different matrix-matrix operations. Only efficient uniprocessor level-3 kernels [14], [16] are required. We define this type of block algorithms as *explicitly parallel.* Here the parallelism must be invoked by using parallel language constructs. The number of synchronization points of an explicitly parallel block algorithm is in theory determined only by the data-dependency at the block level, and can therefore be minimized. The explicit approach makes it easy to balance the work over the available processors and it is possible to use either *static* or *dynamic* scheduling of the work.

**3.1. Parallel Block LU Algorithms.** The block right looking algorithm has been both implicitly and explicitly parallelized. The implicit version uses our parallel level-3 kernels `PDTRSM` and `PDGEMM` (see [5]). The explicit version is parallelized based on the data dependencies at the block level. It is obvious that the level-2 factorization of a block column $A_{s:m,s:e}$ must be completed before the update with respect to this block column can take place. In the following node program, there is only one synchronization needed in each iteration.

If $me = 0$
  1. $L_{1:m,1:nb}U_{1:nb,1:nb} = A_{1:m,1:nb}$ {Use level-2 routine}
For $i = 2 : min(n/nb, m/nb)$
  2. Wait for all processors {Synchronization}
  $s = (i-2) \cdot nb + 1$ {Start of last factorization}
  $e = (i-1) \cdot nb$ {End of last factorization}
  3. $[u, v]$ = My_block($me$, $P$, $n - e$, $nb$) {Get new block indices for $me$}
  If $u \leq n$
    4. Apply pivotings to $A_{s:e,u:v}$ {From all factorizations}
    5. $U_{s:e,u:v} = L^{-1}_{s:e,s:e}A_{s:e,u:v}$ {Use `DTRSM`}
    If $e < m$
      6. $A_{e+1:m,u:v} \leftarrow A_{e+1:m,u:v} - L_{e+1:m,s:e}U_{s:e,u:v}$ {Use `DGEMM`}
  If $me = 0$
    7. $L_{u:m,u:e+nb}U_{u:e+nb,u:e+nb} = A_{u:m,u:e+nb}$ {Use level-2 routine}
8. $[s, e]$ = Next_columns($nb$) {Get indices for next block column}
While $s < n - nb$
  9. Apply pivotings to $A_{e+1:m,s:e}$ {From all factorizations}
  10. $[s, e]$ = Next_columns($nb$) {Get indices for next block column}

The statement [u,v] = My_block(*me*, *P*, *n-e*, *nb*) determines the start position $u$ and the end position $v$ of the columns that processor *me* will work on at this step. In each step $i$ the number of columns of the remaining matrix ($n - e$ = current problem size) is partitioned evenly over the processors with the exception that processor 0 is always given a block with at least *nb* columns (the next block to factorize). The work is shared so that each processor computes the part of $U$ (operation 5) that is required for updating the remaining rows of its part of $A$ (operation 6). This excludes any sychronization between the triangular solve and the matrix update. Processor 0 can start the factorization of the next *nb* columns as soon as it has completed its part of the update of $A$.

The pivotings that so far have not been applied to the left hand side of the current block columns are performed after all block columns are factorized. This is done in parallel, so that each processor applies all row interchanges to a block consisting of *nb* columns. The routine *Next_columns* returns the start position $s$ and end position $e$ for the next column block where processor *me* will do the row interchanges.

**4. Explicit Parallelization and Static Load Balancing.** Since processor 0 performs all level-2 factorizations in the explicitly parallel block algorithm, it will in general do more work than processors $1 : P - 1$. To obtain a balanced load, processor 0 must have less work in the level-3 operations that perform the update of the remaining matrix in each step $i$. This imbalance can be corrected by having a cost function incorporated in the routine *My_Block* that determines the computational work that is performed in each step at the block level. The cost function, based on *the number of flops measured (expressed) in GEMM equivalents*, is described below.

Table 4.1 shows the percentage cost in flops and real execution time of different suboperations in the block right looking algorithm ($m = n$, $nb = 48$). Pivotings_R and Pivotings_L correspond to operations 4 and 8-10, respectively. Different suboperations perform with

| Matrix size | $m = n = 800$ | | $m = n = 1200$ | |
|---|---|---|---|---|
| Suboperation | *Flops* | *Time* | *Flops* | *Time* |
| Level-2 *LU* | 4.6 | 8.6 | 3.0 | 5.8 |
| Pivotings_R | 0.1 | 9.0 | 0.1 | 6.4 |
| DTRSM | 4.2 | 5.4 | 2.9 | 3.9 |
| DGEMM | 91.0 | 74.7 | 93.9 | 82.3 |
| Pivotings_L | 0.1 | 2.3 | 0.1 | 1.6 |

TABLE 4.1
**Percentage cost of suboperations in block LU algorithm**

different speed (performance) which means that the level-2 factorizations and permutations consume more time than their percentage share of the total number of flops. Further, the level-2 operations cost relatively more when the matrix dimension decreases.

| Suboperation | Operation counts | $i$ | $l_i$ |
|---|---|---|---|
| *LU_2* | $(m-e)nb^2 - \frac{1}{3}nb^3 - \frac{1}{2}nb^2 + \frac{5}{6}nb$ | 1 | 1.92 |
| *SWAP* | $(n-e)nb$ | 2 | 120. |
| *TRSM* | $(n-e)nb^2$ | 3 | 1.62 |
| *GEMM* | $2(m-e)(n-e)nb$ | 4 | 1.0 |

TABLE 4.2
**Operation counts and weights of *LU*-cost function**

In Table 4.2, *SWAP* corresponds to the permutation performed in each block iteration, i.e. Pivotings_R in Table 4.1. The operation counts (multiplies and adds) for the different

suboperations in each block iteration are shown together with weights $l_i \geq 1.0$ computed as

$$l_1 = \frac{Mflops\_GEMM}{Mflops\_LU\_2}; l_2 = \frac{Mflops\_GEMM}{Mflops\_SWAP}; l_3 = \frac{Mflops\_GEMM}{Mflops\_TRSM}; l_4 = 1.0;$$

The performance in *Mflops* of individual suboperations is measured on one processor. *Mflops_GEMM* expresses the performance of `DGEMM` for the problem size of the corresponding suboperation. The weights are experimentally chosen and represent a weighted average of the performance of the different block iterations. The cost function for the block *LU* algorithm is expressed as:

$$GEMM_{flops}(LU) = l_1 \cdot LU\_2 + l_2 \cdot SWAP + l_3 \cdot TRSM + l_4 \cdot GEMM$$

**5. Explicit Parallelization and Dynamic Load Balancing.** The static mapping of work on processors as described in section 4 will work well on a dedicated system where each scheduled task is guaranteed a physical processor. However, if the program is executing in a multi-user environment the parallel performance may degrade since other users will compete for the resources of the multiprocessor system (e.g. processors and shared memory). When a system cannot be dedicated *dynamic scheduling* of the work to processors is likely to be a better approach. The idea is to split the parallelizable work into a *queue of tasks* and an idle processor is given the next available task in the queue.

**5.1. Parallel Block LU Algorithm with Dynamic Scheduling.** The parallelizable work in the block right looking *LU* algorithm (i.e. the columns to be updated in each step) is partitioned into $2 \times P$ tasks ($P$ = number of processors to be used). A larger number of tasks (with fewer columns per block) will be more likely to keep processors from becoming idle. However, it would also imply a possible degradation of the performance of `DTRSM` and `DGEMM`. Our choice of $2 \times P$ tasks in the queue is based on benchmark results with different sizes of the task queue.

The dynamic algorithm does not require the same kind of synchronization as the statically scheduled algorithm. Here different processors can be working with operations belonging to two different block iterations $i$ at the same time. The synchronization required is to ensure that the submatrices to be used in the triangular solve and in the matrix update are computed and that no other processor is working on the same column block.

**6. Performance Results of Parallel Block LU factorization.** In this section we present performance results of the parallel block algorithms described in sections 3-5 and implemented in IBM Parallel Fortran (PFP) [20]. The parallel language extensions used are summarized below: `ORIGINATE` to create tasks, `TERMINATE` to terminate tasks, `SCHEDULE` or `DISPATCH` to assign work to a task - makes it possible to execute independent subroutines in parallel, `WAIT FOR` to detect when a task has completed its assigned work, `PARALLEL LOOP` to execute iterations of a loop concurrently, and routines from the parallel library that handle locks and events are used to implement different synchronization techniques. Notice that subroutine calls are not permitted within a `PARALLEL LOOP` in PFP. For more details of the parallel language extensions see [20]. Code that does not make use of parallel language constructs is compiled with the IBM compiler VS FORTRAN Version 2.4 which at present has better optimization for vectorization. All results presented are obtained using up to five physical processors of a quasi-dedicated IBM 3090 VF/600J (the IBM Fortran routine `CLOCKX` is used for wall-clock timing). The results for a specific parallel block algorithm are the best performance obtained from a series of tests. Performance results measured in Mflops and speed-up factors are presented for implicitly parallel and explicitly parallel block algorithms. The theoretical peak performance of IBM 3090 VF/600J for 64 bit arithmetic

is 828 Mflops (6 · 138) while its practical peak performance [5] is around 600 Mflops (500 Mflops on 5 processors).

Table 6.1 shows performance results of described block right looking algorithms and the corresponding ESSL routine [13].

| Block algorithmm | *Implicit* | *Implicit* | *Static* | *Static* | *Dynamic* | *Dynamic* | *ESSL* | *ESSL* |
|---|---|---|---|---|---|---|---|---|
| Matrix size | 500 | 1200 | 500 | 1200 | 500 | 1200 | 500 | 1200 |
| 1 proc | 83.7 | 95.8 | 83.5 | 98.4 | 84.9 | 98.1 | 88.7 | 99.0 |
| 2 proc | 104.4 | 167.4 | 147.5 | 192.4 | 148.3 | 192.9 | 162.7 | 195.4 |
| *Speed-up* | *1.25* | *1.75* | *1.77* | *1.96* | *1.75* | *1.97* | *1.83* | *1.97* |
| 3 proc | 112.0 | 217.4 | 193.7 | 276.6 | 196.4 | 277.6 | 212.2 | 292.8 |
| *Speed-up* | *1.34* | *2.27* | *2.32* | *2.81* | *2.31* | *2.83* | *2.39* | *2.96* |
| 4 proc | 110.7 | 231.2 | 219.8 | 355.1 | 223.4 | 355.2 | 226.9 | 377.9 |
| *Speed-up* | *1.32* | *2.41* | *2.63* | *3.61* | *2.63* | *3.62* | *2.56* | *3.82* |
| 5 proc | 104.5 | 253.1 | 234.8 | 430.2 | 239.9 | 431.0 | 257.8 | 457.2 |
| *Speed-up* | *1.25* | *2.64* | *2.81* | *4.37* | *2.83* | *4.39* | *2.91* | *4.62* |

TABLE 6.1
Performance of parallel block LU algorithms

**Discussion.** The difference in performance between the implicit and the explicit parallel algorithms increases with the number of processors. In this context it is interesting to study the best possible speed-up of the implicitly parallel block algorithms. Table 6.2 shows the real execution time level-3 fraction ($f$) and real execution time level-2 fraction ($1-f$) of the block right looking algorithms that are implicitly parallelized (numbers from Table 4.1). The third column of Table 6.2 shows the fractions for the $LU$ factorization if we assume that all permutations also can be performed in parallel. In the second column all permutations are included in the level-2 fraction.

| *m=n=1200* | *LU* | *LU*(pivot //) |
|---|---|---|
| $f$ (level-3) | 0.862 | 0.942 |
| $1-f$ (level-2) | 0.138 | 0.058 |

TABLE 6.2
Real execution time level-3 and level-2 fractions

Table 6.3 shows the speed-up factors $S_p$ of the implicitly parallel block algorithms if we assume that all level-3 operations are perfectly parallelized (with no overhead, see below) and all level-2 operations are performed on one processor. They are computed from Amdahl's law for parallel processing (see e.g. [9]):

$$S_P = \frac{P}{f + (1-f)P}.$$

If $f < 1$, $S_P$ is bounded by $(1-f)^{-1}$ and this factor is displayed in the last column of Table 6.3.

| *m=n=1200* | *# processors* | | | | | |
|---|---|---|---|---|---|---|
| Routine | 1 | 2 | 3 | 4 | 5 | $(1-f)^{-1}$ |
| *LU* | *1.00* | *1.76* | *2.35* | *2.83* | *3.22* | *7.25* |
| *LU* (pivot //) | *1.00* | *1.89* | *2.69* | *3.41* | *4.06* | *17.24* |

TABLE 6.3
Perfect (theoretical) speed-up factors of the implicitly parallel block algorithms

One conclusion from these numbers and Tables 6.1-6.3 is that it requires a small level-2 fraction in order to be able to match the performance of the best explicitly parallel block

algorithms. Possibly, the performance of the implicitly parallel block algorithms can be improved by also parallelizing the level-2 BLAS. However, our experiences on the IBM 3090 VF/600J are that we do not gain anything by doing so. The granularity of the level-2 operations of the block algorithms are too small and the overhead for parallelization is to costly. Similar results are also shown in [6]. Note that, some level-2 operations have been successfully parallelized on other parallel shared memory systems [19].

**Parallelization overhead.** The results in Table 6.1 show very small speed-up factors for matrices of size $500 \times 500$. The reason is that the overhead due to synchronization primitives is more or less constant while the work between synchronizations decreases with the size of the problem. Most of the overhead is associated with `ORIGINATE` (and to a less extent `TERMINATE`) and therefore the creation of tasks is done once in the beginning of the program. The overhead costs associated with all other parallelization and synchronization primitives are normally much lower. The overhead caused by `ORIGINATE` and `TERMINATE` is a function of the number of tasks created and the size of each task. The following theoretical model estimates the overhead cost associated with originating and terminating $p$ tasks:

$$T_{ORIG}(p) = (o1 + o2 \cdot S) \cdot p$$

where $S$ corresponds to the storage space in $Mbytes$ that is allocated for each task (including code and memory for local variables and arrays etc). The parameters $o1$ and $o2$ in the linear model are determined from a least squares fit of overhead (caused by `ORIGINATE` and `TERMINATE`) timings for different matrix sizes. The values used are $o1 = 0.0075secs$ and $o2 = 0.0236secs$.

By considering this overhead we can predict an upper bound on the theoretical speed-up of our explicitly parallel block algorithms (see Table 6.4). The time on $p$ processors is predicted to be:

$$T_p = \frac{T_1}{p} + T_{ORIG}(p-1)$$

where $T_{ORIG}(p)$ is of the same magnitude for the explicitly and implicitly parallel block algorithms.

| *m=n=1200* | *# processors* | | | | |
|---|---|---|---|---|---|
| Routine | 1 | 2 | 3 | 4 | 5 |
| *LU* | 98.1 | 195.0 | 289.2 | 378.9 | 463.0 |
| | *1.00* | *1.99* | *2.95* | *3.86* | *4.72* |

TABLE 6.4
**Upper bounds on performance and speed-up factors of the explicitly parallel block algorithm**

**7. Some Conclusions.** Performance results show that explicitly parallel block algorithms can reach close to practical peak performance on $1-5$ processors by using optimized uniprocessor level-3 BLAS [14], [16], [15]. Large problems ($n = 1200$ here) are required for high performance. Although the parallel efficiency decreases as a function of the number of processors we have seen up to 90% parallel efficiency (defined as speed-up factor divided by number of processors) on five processors of an IBM 3090 VF/600J system for the *LU* factorization. This result is justified by a theoretical model that predicts an upper bound on the best possible speed-up with respect to the most significant parallelization overhead of language constructs and their implementations.

The corresponding best parallel efficiency of the implicitly parallel block algorithm is just over 50% for the *LU* factorization. By assuming perfect parallel speed-up of the parallel level-3 BLAS, we have shown that, at least in theory, it is possible to obtain around

80% parallel efficiency for the implicitly parallel block *LU* algorithm The proportionately lower efficiency of the implicitly parallel block algorithm can also be explained in terms of the parallelization overhead, which is architecture dependent. In order to exploit the full potential of hierarchical memory multiprocessor systems, implicit parallelism (as in LAPACK) will require highly efficient parallel level-3 BLAS, or explicit parallelism will be required. The benefit from explicitly parallel block algorithms increases with the number of physical processors.

## REFERENCES


[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, "LAPACK: A Portable Linear Algebra Library for High-Performance Computers", Tech. Report CS-90-105, Univ. of Tennessee, Knoxville, May 1990. (LAPACK Working Note #20).

[2] E. Andersson and J. Dongarra, "Implementation Guide for LAPACK", Tech. Report CS-90-101, Univ. of Tennessee, Knoxville, April 1990. (LAPACK Working Note #18).

[3] E. Andersson and J. Dongarra, "Evaluating Block Algorithm Variants in LAPACK", in J. Dongarra, P. Messina, D. Sorensen and R. Voigt (eds), *Parallel Processing for Scientific Computing*, SIAM Publications, 1990, pp 3-8.

[4] E. Cohen, G. King and J. Brady, "Storage Hierarchies", *IBM Systems Journal*, Vol. 28(1) (1988) pp 62-76.

[5] K. Dackland, E. Elmroth, B. Kågström, C. Van Loan, "Parallel Block Matrix Factorizations on the Shared Memory Multiprocessor IBM 3090 VF/600J", Report UMINF-91.07, Inst. of Information Processing, Univ. of Umeå, S-901 87 Umeå, January 1991.

[6] M. Daydé and I. Duff, "Use of Level 3 BLAS in LU Factorization in a Multiprocessing Environment on Three Vector Multiprocessors: the ALLIANT FX/80, the CRAY-2, and the IBM 3090 VF", Technical Report CERFACS, August 1990.

[7] J. Dongarra, J. Du Croz, S. Hammarling and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms", *ACM Trans. on Mathematical Software*, Vol. 14 (1988) pp 1-17, 18-32.

[8] J. Dongarra, J. Du Croz, S. Hammarling and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms", *ACM Trans. on Mathematical Software*, Vol. 16 (1990) pp 1-17, 18-28.

[9] J. Dongarra, I. Duff, D. Sorensen and H. Van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM Publications, 1991.

[10] K. Gallivan, W. Jalby, U. Meier and A. Sameh, "Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design", *Int. J. Supercomputer Applications*, Vol 2 (1988), pp 12-48.

[11] K. Gallivan, R. Plemmons and A. Sameh, "Parallel Algorithms for Dense Linear Algebra Computations", *SIAM Review*, Vol. 32 (1990), pp 54-135.

[12] G. Golub and C. Van Loan, *Matrix Computations*, John Hopkins Press, 2nd edition, 1989.

[13] IBM, *Engineering and Scientific Subroutine Library Guide and Reference*, SC23-0184-3, November 1988.

[14] B. Kågström and P. Ling, "Level 2 and 3 BLAS Routines for IBM 3090 VF: Implementation and Experiences", in J. Dongarra, I. Duff, P. Gaffney and S. McKee(eds), *Vector and Parallel Computing*, Ellis Horwood, 1989, pp 229-240.

[15] B. Kågström and C. Van Loan, "GEMM-Based Level-3 BLAS", Technical Report, Dept. of Computer Science, Cornell University, December 1989.

[16] P. Ling, "A Set of High Performance Level-3 BLAS Structured and Tuned for the IBM 3090 VF and Implemented in Fortran 77", Report UMINF-179.90, Inst. of Information Processing, Univ. of Umeå, S-901 87 Umeå, May 1990.

[17] Parallel Computing Forum. *Proposal*, 1990.

[18] G. Radicati, Y. Robert and P. Sguazzero, "Block Processing in Linear Algebra on the IBM 3090 Vector Multiprocessor", *SUPERCOMPUTER*, Vol. 5, No. 1 (1988) pp 15-25.

[19] Q. Sheikh and J. Liu, "Performance of Block Matrix Factorization Algorithms and LAPACK on CRAY Y-MP and CRAY-2", Cray Research Inc., 1990.

[20] L. Toomey, E. Plachy, R. Scarborough, R. Sahulka, J. Shaw and A. Shannon, "IBM Parallel Fortran", *IBM Systems Journal*, Vol. 27 (1988) pp 416-435.