Applied Numerical Mathematics 10 (1992) 37–57
North-Holland

APNUM 298

Block-Cholesky for parallel processing

Margreet Louter-Nool
CWI, P.O. Box 4079, 1009 AB Amsterdam, Netherlands

Abstract

Louter-Nool, M., Block-Cholesky for parallel processing, Applied Numerical Mathematics 10 (1992) 37–57.

We concentrate on the Cholesky factorization of $A = LL^T$, where A is a positive definite symmetric matrix and L is a lower triangular matrix. A blocked algorithm based on Level 3 BLAS is discussed. When using Level 3 BLAS kernels in a multiprocessing mode, one can parallelize within each kernel, or can obtain parallelism by performing different matrix–matrix operations on different processors. We apply parallelism over the blocks. We study the amount of parallelism and we discuss the data dependency graph. The SCHEDULE package is used to obtain a portable scheduling of the tasks. Numerical results of our method are presented and compared with the results for a block algorithm parallelized within the Level 3 BLAS kernels.

Keywords. Cholesky factorization, block algorithms, parallelism, scheduling.

1. Introduction

We discuss the Cholesky factorization

$$A = LL^T,$$

with A a positive definite symmetrix matrix and L a lower triangular matrix. The matrices A and L are partitioned into submatrices, or blocks. The algorithm presented here is described in terms of matrix–matrix operations on distinct blocks and we study parallelism over the blocks. We consider the data dependency of the block operations and we discuss some aspects of scheduling of tasks involved with block operations.

Since the execution time of algorithms on high performance computers does not merely depend on the number of floating-point operations, we consider machine-dependent aspects like the Megaflop rates attained for different blocksizes, the performance ratio for different matrix–matrix operations, and the influence of data movements on the performance. To obtain an efficient and portable implementation we used calls to Level 3 BLAS [3] and for the scheduling we made use of the SCHEDULE package of Hanson and Sorensen [12]. We present some results for the Alliant FX/4, the Alliant FX/8, and the IBM 3090/VF. Finally, we draw some conclusions.

2. The Cholesky factorization

The Cholesky factorization is one of the most analyzed algorithms in numerical algebra [6,10,11]. It is a straightforward algorithm and, since A is positive definite, pivoting is not

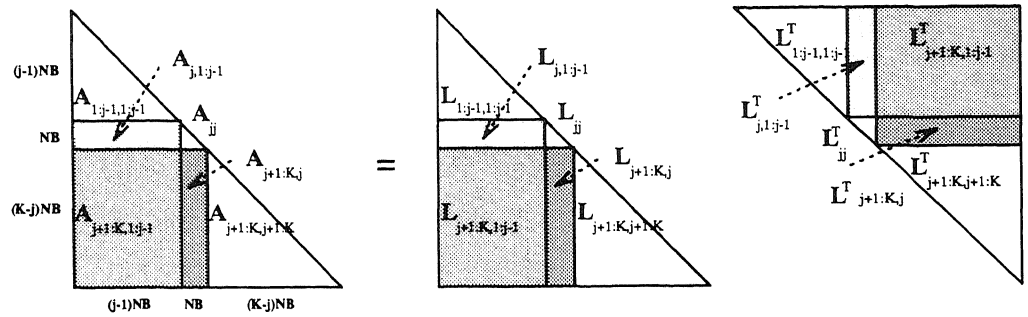


Fig. 1.

necessary to ensure or improve numerical stability. The general step for the Cholesky factorization looks like:

```
FOR ...
  FOR ...
    FOR ...
      a_ij = a_ij − l_ik · a_kj   (l_ik = a_ik/a_kk)
```

and it depends on the position of the loop parameters i, j, and k, respectively, whether we are dealing with row, column or submatrix Cholesky factorization [6,16,17]. All forms have the same number of floating-point operations, indicating that the amount of arithmetic is exactly the same for all variants, although the data access and the updating patterns are different.

How do we design an efficient algorithm which performs well on a large variety of machines? Ortega analyzed all forms for vector [16] and parallel machines [17]. Column Cholesky appeared to be favorite for machines with vector-processing capabilities. For parallelism only ›cal memory systems with row or column wrapped interleaved storage are considered [17]. We :us on block-column Cholesky [5]. For convenience, we assume that the blocksize NB is a)per divisor of matrix order n and

$$\mathrm{K} = \frac{n}{\mathrm{NB}}. \tag{2.1}$$

The block factorization can be visualized as in Fig. 1. All elements of A are blockmatrices: $A_{1:j-1,1:j-1}$, A_{jj}, and $A_{j+1:\mathrm{K},j+1:\mathrm{K}}$ are symmetric block-matrices containing $(j-1)\times(j-1)$ blocks, a single block, and $(\mathrm{K}-j)\times(\mathrm{K}-j)$ blocks, respectively. Assume the first $(j-1)\times \mathrm{NB}$ columns, or the matrices $L_{1:j-1,1:j-1}$, $L_{j,1:j-1}$ and $L_{j+1:\mathrm{K},1:j-1}$ to be known. In step j, we compute the next block-column (a set of NB single columns) of L. By equating LL^{T} and A, we obtain the following two relations:

$$A_{jj} = L_{j,1:j-1}\cdot L_{j,1:j-1}^{\mathrm{T}} + L_{jj}\cdot L_{jj}^{\mathrm{T}}, \tag{2.2}$$

$$A_{j+1:\mathrm{K},j} = L_{j+1:\mathrm{K},1:j-1}\cdot L_{j,1:j-1}^{\mathrm{T}} + L_{j+1:\mathrm{K},j}\cdot L_{jj}^{\mathrm{T}}. \tag{2.3}$$

From the first equation (2.2) L_{jj} can be calculated. The operations involved are:

(1) A symmetric rank-k update

$$A_{jj}^{(1)} \leftarrow A_{jj} - L_{j,1:j-1}\cdot L_{j,1:j-1}^{\mathrm{T}}. \tag{2.4a}$$

(2) A Cholesky factorization on a single block

$$L_{jj} \leftarrow Cholesky\left(A_{jj}^{(1)}\right). \tag{2.4b}$$

The second equation (2.3) delivers $L_{j+1:\kappa,j}$.

(3) A matrix–matrix product

$$A_{j+1:\kappa,j}^{(1)} \leftarrow A_{j+1:\kappa,j} - L_{j+1:\kappa,1:j-1} \cdot L_{j,1:j-1}^{\mathrm{T}}. \tag{2.4c}$$

(4) Finally, we have to solve a triangular system

$$L_{j+1:\kappa,j} \leftarrow A_{j+1:\kappa,j}^{(1)} \cdot L_{jj}^{-\mathrm{T}}. \tag{2.4d}$$

If operations are only performed on single blocks and if, in addition, all components are single blocks as well, then (2.4a) can be rewritten as

$$A_{jj}^{(1)} \leftarrow A_{jj} - \sum_{i=1}^{j-1} L_{ji} \cdot L_{ji}^{\mathrm{T}} \tag{2.5a}$$

where $L_{j,1:j-1}$ has been subdivided into the single blocks L_{ji} $(i = 1, \ldots, j-1)$. Analogously, operation (2.4c) can be translated into

$$A_{lj}^{(1)} \leftarrow A_{lj} - \sum_{i=1}^{j-1} L_{li} \cdot L_{ji}^{\mathrm{T}}, \quad l = j+1, \ldots, \kappa. \tag{2.5c}$$

The matrix–matrix products in (2.5a) and (2.5c) are data-independent and can be carried out in parallel. This approach, however, requires additional memory, since the temporary result matrices

$$B_j^i = L_{ji} \cdot L_{ji}^{\mathrm{T}} \quad \text{and} \quad C_{lj}^i = L_{li} \cdot L_{ji}^{\mathrm{T}}, \qquad i = 1, \ldots, j-1,\ l = j+1, \ldots, \kappa$$

are generated. In the end, the matrices B_j^i and C_{lj}^i have to be subtracted from A_{jj} and A_{lj}, respectively.

An alternative way to perform the jth step of the factorization is to translate (2.4a)–(2.4d) into

$$A_{jj}^i \leftarrow A_{jj}^{i-1} - L_{ji} \cdot L_{ji}^{\mathrm{T}}, \qquad i = 1, \ldots, j-1, \tag{2.6a}$$

$$L_{jj} \leftarrow Cholesky\left(A_{jj}^{j-1}\right), \tag{2.6b}$$

$$A_{lj}^i \leftarrow A_{lj}^{i-1} - L_{li} \cdot L_{ji}^{\mathrm{T}}, \qquad i = 1, \ldots, j-1,\ l = j+1, \ldots, \kappa, \tag{2.6c}$$

$$L_{lj} \leftarrow A_{lj}^{j-1} \cdot L_{jj}^{-\mathrm{T}}, \qquad l = j+1, \ldots, \kappa, \tag{2.6d}$$

where A_{ij}^0 denotes the original submatrix A_{ij}. In this case, the symmetric rank-k update (2.6a) and the matrix–matrix multiplications of the i-loop of (2.6c) can no longer be executed simultaneously, since each update requires data of the previous computed update. However, for different values of l the matrix–matrix products, possibly followed by the solution of the triangular system (2.6d), are data-independent. Summarizing, the computation of (2.4a)–(2.4d) has been broken up into smaller units of computations. We will show that it is no longer necessary to execute the units in the same order as described. In the next section, we present the execution dependencies between them in order to specify a parallel computation.

3. Presentation of the method

Throughout this paper we will use computational kernels for basic operations in linear algebra. These kernels are termed the BLAS, for Basic Linear Algebra Subprograms. The Level 2 BLAS [4] incorporates matrix–vector operations, and the Level 3 BLAS comprises matrix–matrix kernels [3]. In (2.6a)–(2.6d) we have translated the original computation into smaller units of computations. Each of these will be associated with its BLAS subroutine name. The Level 3 BLAS used are:

_SYRK for performing a symmetric rank-k update on the diagonal blocks,
_TRSM for solving a number of systems with the same triangular coefficient matrix,
_GEMM for multiplying two matrices.

The fourth operation to perform is the Cholesky factorization, referred to as

_LLT for factorizing a diagonal block.

In this paper we use the term process or task rather than unit of computation.

A is partitioned into $K \times K$ blocks. Both A and the diagonal blocks are symmetric. From (2.6a)–(2.6d) we derive the number of processes needed to compute the complete factorization of A:

$$\begin{aligned} &\_\text{LLT}: && K, \\ &\_\text{SYRK}: && \tfrac{1}{2}K(K-1), \\ &\_\text{TRSM}: && \tfrac{1}{2}K(K-1), \\ &\_\text{GEMM}: && \tfrac{1}{6}K(K-1)(K-2). \end{aligned} \tag{3.1}$$

This implies that the total number of tasks will be

$$M = \tfrac{1}{6}K(K+1)(K+2). \tag{3.2}$$

For the description of our algorithm, it is convenient to number the tasks. A list schedule L_M of M tasks denoted by

$$L_M = \{T_1, T_2, \ldots, T_M\} \tag{3.3}$$

represents a certain order of the M tasks. The choice of ordering will determine the scheduling. If we number the tasks on the matrices of the first column from 1 to K and those of the second column (i.e., 2 operations/block) from $K+1$ to $K+2(K-1)$ and so on then we obtain an ordering that corresponds to the column Cholesky. Analogously, a numbering along the rows will result in a row Cholesky, and a submatrix Cholesky corresponds with a numbering starting with the first updates succeeded by the second updates etcetera.

The aim of our investigations is to apply a simple scheduling strategy and to find an optimal value for K, the partitioning parameter (2.1). To obtain a good speedup with a multitasked code, we have to keep the processors concurrently active as much as possible and have to minimize memory conflicts between processors. An execution of the tasks strictly in conformity with one of our proposed numberings will not generate an optimal code. Many processes are data-dependent and processors may be idle while several tasks ready to be executed are waiting to be activated. Our algorithm for a fixed number of parallel processors, say p, will execute tasks, even when their results are not needed at that time.

The scheduling strategy.

(A) The only schedulable task to start with is the factorization of the first diagonal block. As soon as this process has been completed, $\kappa - 1$ tasks become schedulable, namely the calls to _TRSM on the first column matrices A_{j1}, $j = 2, \ldots, \kappa$. Assume that $p \leqslant \kappa - 1$, then we continue with the next p schedulable tasks. For $p > \kappa - 1$ only $\kappa - 1$ processors can be active and $p - (\kappa - 1)$ processors will still be idle.

(B) When a process has finished on processor P_α then other tasks may become schedulable. The next task on processor P_α will be the first ready task in list L_M. The ordering of the tasks is determined by the selected numbering. If no schedulable tasks are available then processor P_α has to wait until schedulable tasks are generated by tasks on other processors.

(C) Repeat (B) until all tasks have been completed.

It is easy to determine for each process which dependencies have to be satisfied and to determine which processes depend on that specific process. Figure 2 shows the data dependency graph for a matrix partitioned into 5×5 blocks. A node is specified by either an A or an L denoting the computation of

A_{ij}^k: kth temporary update of submatrix A_{ij} $(k \leqslant j - 1)$,

L_{ij}: the final update of submatrix A_{ij}.

Note that the dependency graph of a matrix partitioned into 4×4 blocks is a part of the graph of Fig. 2, namely that graph spanned by the nodes A_{ij}^k and L_{ij} with $1 \leqslant j \leqslant i \leqslant 4$. We remark that the amount of parallelism decreases during the course of the factorization. At the end no parallelism is left: the computation of L_{54}, A_{55}^4, and L_{55} cannot be done simultaneously.

Assume that the computation costs only depend on the number of floating-point operations. If we define

1 CU = the number of floating-point operations for a Cholesky factorization of a block of order NB,

then we obtain the computational costs as listed in Table 1 for the different operations on equally sized blocks of order NB. The value at the top right of a node in Fig. 2 stands for the minimal execution time expressed in CUs on an unbounded number of processors to perform the associated process and its preceding tasks. The solid line in Fig. 2 shows the critical (minimal) path of 35 CU.

The speedup is defined by

$$S_p = \frac{\text{Time used by 1 processor}}{\text{Time used by } p \text{ processors}} \tag{3.4}$$

and the efficiency by

$$\textit{Efficiency} = \frac{S_p}{p} \times 100\%. \tag{3.5}$$

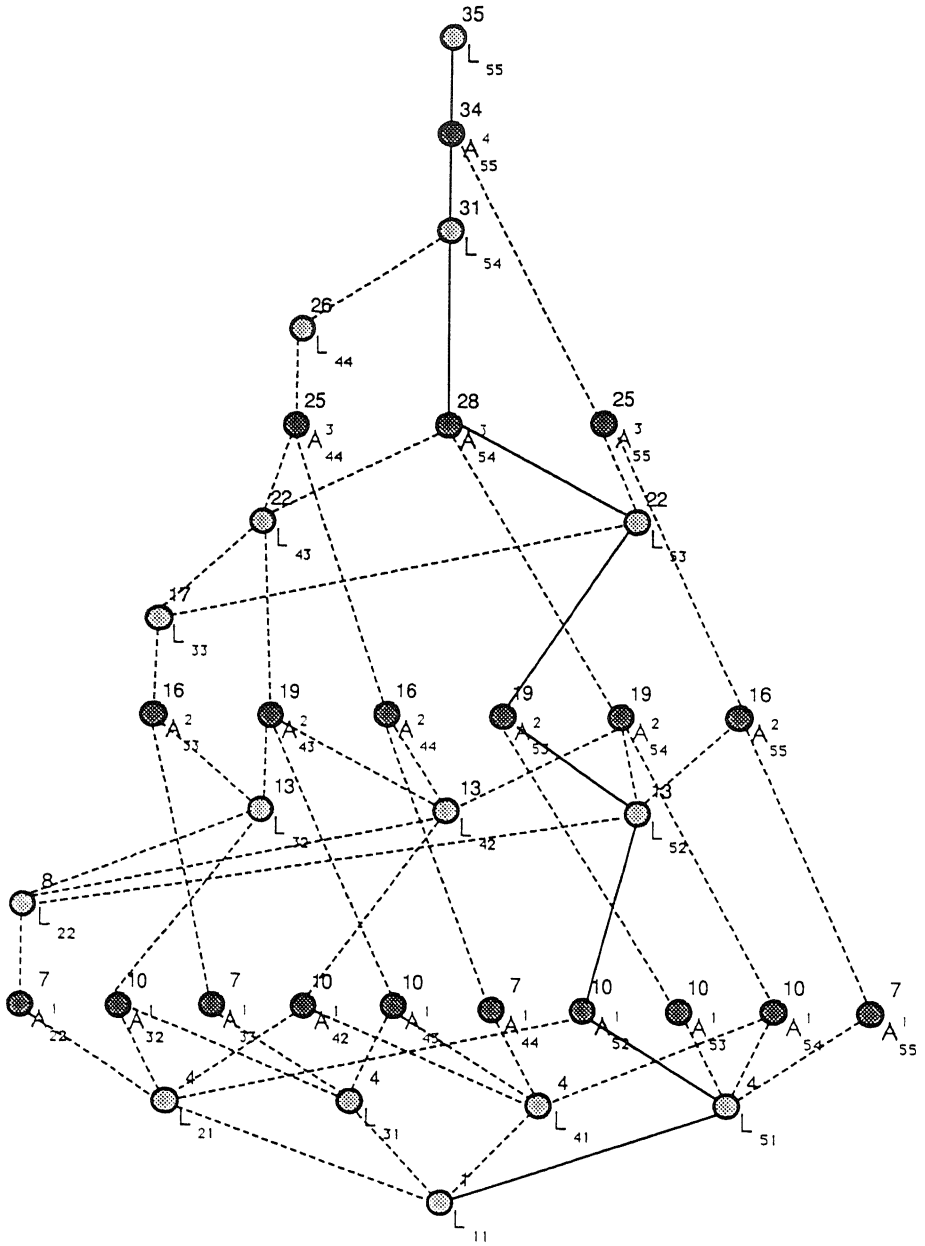


Fig. 2. The data dependency graph for the Cholesky factorization of a matrix partitioned into 5×5 blocks on four processors.

From Fig. 2 and the values of Table 1 we may conclude that, for our example with $\kappa = 5$, the maximum speedup is

$$S_{\max} = \frac{\text{Time required on 1 processor}}{\text{Time required for critical path}} = \frac{125 \text{ CU}}{35 \text{ CU}} = 3.57 \tag{3.6}$$

Table 1
Theoretical execution times expressed in CUs

Operation	FLOPs	CU
_LLT	$\frac{1}{3}$NB^3 + ···	1
_SYRK	NB^3 + ···	3
_TRSM	NB^3 + ···	3
_GEMM	2NB^3 + ···	6

assuming enough processors to be available. For the graph of Fig. 2 the maximum number of processes which can be computed in parallel is 10, namely the first updates A^1_{22}, $A^1_{32}, \ldots, A^1_{55}$ which cover the whole matrix except for the first column. Hence, on ten or more processors, the efficiency cannot exceed 35.7%.

Let us return to the scheduling as described in this section. Suppose four processors are available, and the tasks have been numbered corresponding to the column Cholesky. The scheduling based on the CU distribution of Table 1 is shown in Fig. 3. In this case, 42 CU are required, and the speedup is

$$S_4 = \frac{125 \text{ CU}}{42 \text{ CU}} = 2.98.$$

The efficiency of 74% is twice the efficiency obtained on ten processors. Another numbering of the tasks might result in another speedup, and we notice that the speedup for this example is not optimal. The critical path method (CP) considered as the most efficient heuristic method for solving the scheduling problem in hand needs 39 CU for our 5×5 problem. The CP method is based on initial execution time values T_i. If these estimated initial values T_i vary little from the values obtained during execution then the CP method will give rise to an inefficient execution [15].

At the beginning as well as at the end of the factorization process, operations cannot be performed in parallel. To minimize the execution time of the initial and final phase, a smaller value of the blocksize NB might be considered. Theoretically, on a fixed number of processors, the performance increases with the number of blocks and the maximum speedup will be reached for a blocksize of 1. We will show in the next section that machine characteristics will play an important role in the performance in the choice of the blocksize.

4. A portable implementation based on SCHEDULE and BLAS

When implementing a parallel block algorithm that has to be efficient on a wide variety of parallel machines one needs a portable implementation to define data dependencies and parallel structures, and to coordinate the parallel execution. For this purpose, we used the SCHEDULE package of Hanson and Sorensen [12]. In addition, the algorithm was implemented in terms of calls to Level 3 BLAS. For the single diagonal blocks of order NB an unblocked Level 2 BLAS implementation of the Cholesky factorization was used.

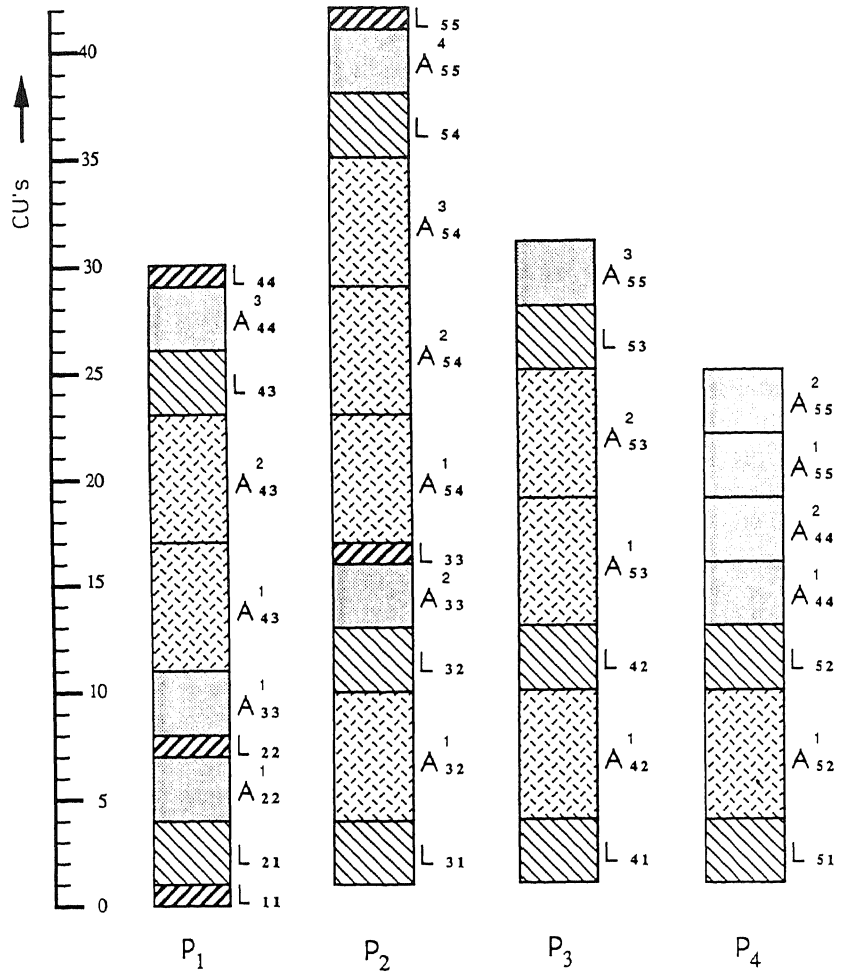


Fig. 3. Scheduling of the Cholesky factorization of a matrix partitioned into 5×5 blocks on four processors.

4.1. Machine dependencies

In the previous section we explained that the performance of the parallel block algorithm depends on:

- K: the partitioning parameter,
- p: the number of processors,
- L_M: the scheduling,
- CU: the ratio of the execution times for the different tasks.

Theoretically, we could calculate the speedup for fixed values of K, p, L_M, and some well-defined CU-distribution. In practice, however, machine-dependent aspects influence the CU-distribution. It is closely related to the BLAS implementation. The BLAS performance in turn strongly depends on the data structure and the blocksize. Moreover, the influence of possible reuse of the cache contents can hardly be expressed in terms of the above mentioned

variables (cf. Gallivan et al. [9]). In the next section, we focus on the scheduling by SCHEDULE, which rather differs from the scheduling we proposed in Section 3.

4.2. The scheduling by SCHEDULE

The SCHEDULE package does not allow assigning priorities to tasks, which could be desirable, for example, to reuse cache (if possible) or to rank time-consuming tasks above less time-consuming tasks. This implies that the influence of a particular ordering cannot be measured. In practice, even for small values of K, the scheduling on a fixed number of processors turns out to be unique for each run. This can be explained by the execution times of the tasks. For our problem we only distinguish four different execution times $T_{\text{-GEMM}}$, $T_{\text{-TRSM}}$, T_{SYRK} and T_{LLT}. Most scheduling problems are dealing with a larger variation in execution times which makes it easier to predict the flow of execution. In Section 5.2 we present a few examples of the scheduling by SCHEDULE.

5. Experiments

We will comment on three different implementations of the Cholesky factorization.

DLLTB

The algorithm as described in the previous sections will be referred to as DLLTB. The way the data is stored influences the performance. Our algorithm operates on single blocks. For that reason we explicitly partition the matrix A into blocks. This means that the matrix is stored blockwise by means of a four-dimensional array

$$A[1:\text{NB}, 1:\text{NB}, 1:\text{K}, 1:\text{K}].$$

The element $A[i, j, k, l]$ refers to element (i, j) of block (k, l).

DLLT3

In this paper we also consider the performance of an "ordinary" Level 3 BLAS implementation to perform the Cholesky factorization. It can be compared with DPOTRF from LAPACK [1]. The matrix to factorize is stored in the traditional FORTRAN way, which means columnwise in a two-dimensional array. The routine DLLT3 exploits parallelism within the BLAS kernels. Operations are not performed on single blocks but on much larger block-matrices. Figure 4 illustrates how such blocks are composed.

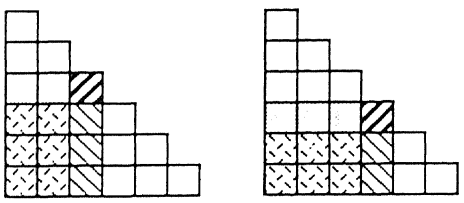


Fig. 4. The combinations of blocks per step.

DLLT

Both DLLTB and DLLT3 need a routine to compute the Cholesky factorization of a submatrix of order NB. The unblocked implementation we used for this job is the routine DLLT based on Level 2 BLAS. DLLT performs a column Cholesky factorization which is well suited for vector machines (see George et al. [10]).

The machines used in the numerical experiments are an Alliant FX/4 with four processors (at CWI), an Alliant FX/8 with eight processors (at Argonne) and an IBM 3090/VF with six processors. The IBM 3090/VF is located at the Amsterdam Academic Computer Centre (SARA). In Section 4, we proposed to use BLAS to obtain high performances. On the Alliants, all levels of BLAS are vendor-supported and these codes are more powerful than model implementations written in portable FORTRAN. For the IBM vectorized codes are available for DGEMM and DTRSM, but neither of them has been parallelized. All experiments are carried out in double precision.

5.1. Performances of BLAS for the Alliant FX/4 and FX/8

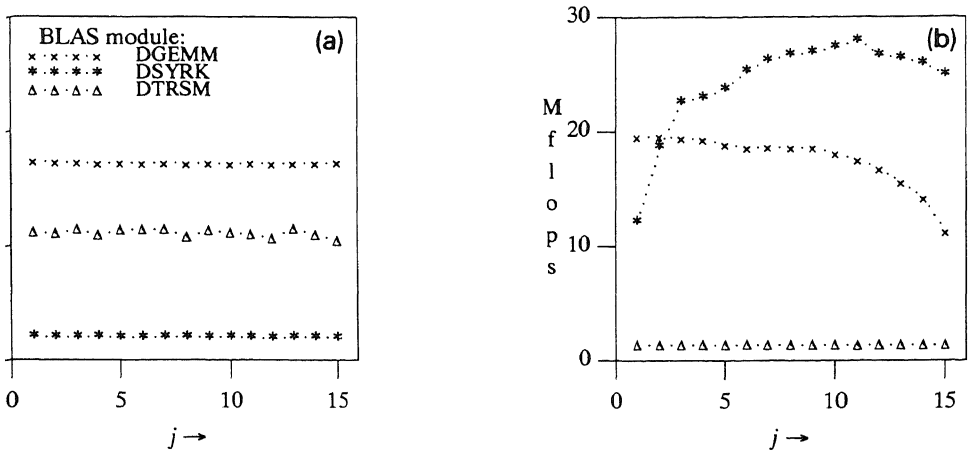
We consider both the single and multi-processor BLAS performance. We have experimented with several blocksizes, among which 32, 48, 64, 80, 96. The results for single NB × NB blocks (cf. Table 2) can be used to analyze the performance of DLLTB, since the SCHEDULE tasks are single processor tasks each. Figure 5 shows the BLAS performance on four (FX/4) and eight (FX/8) processors. For each step j, we measured the performance of the operations (2.4a), (2.4c), and (2.4d), denoted by DSYRK, DGEMM, and DTRSM, respectively. Figure 5(b) illustrates that the BLAS performance on the Alliant FX/8 strongly depends on j. Presently, a better BLAS release is available—installed on the FX/4, but not yet on the FX/8—which does not suffer from such dependencies (see also Jalby and Meyer [13]). In Louter-Nool and Winter [14] the difference in performance between both BLAS releases on the FX/4 is illustrated. For the single processor case, an improved version of DSYRK is used based on the Alliant intrinsic function DOTPRODUCT. This alternative code is not well suited for DLLT3, since $n \gg$ NB. For that case, it is better to use the original vendor-provided DSYRK implementation.

5.2. Graphic output by SCHEDULE

The use of SCHEDULE has some nice properties. The package is able to produce an output file that records the units of computation as executed. A graphic program of a SUN worksta-

Table 2
Performances of Alliant BLAS 3 kernels on a single processor

	Mflops on Alliant FX/4, $p = 1$					Mflops on Alliant FX/8, $p = 1$				
	NB = 32	NB = 48	NB = 64	NB = 80	NB = 96	NB = 32	NB = 48	NB = 64	NB = 80	NB = 96
DGEMM	5.2	5.1	5.3	5.2	5.1	4.2	4.3	4.7	4.6	4.9
DTRSM	3.0	3.5	3.8	4.0	4.1	0.7	1.0	1.2	1.3	1.5
DSYRK	0.9	1.3	1.5	1.8	1.9	1.7	2.5	3.5	4.1	4.6
(DSYRK)	(2.5)	(2.6)	(2.1)	(2.7)	(2.4)					



nance Level 3 BLAS on Alliant FX/4, $p = 4$, NB = 64 (a) and on Alliant FX/8, $p = 8$, NB = 64 (b).

·pret this output. It is possible to construct the dependency graph and to show the ıuence as it was run on a parallel machine. We used these output files to display execution, analogously to Fig. 3.

based on Alliant DSYRK) and 7 (DOTPRODUCT DSYRK) are both concerned with the :torization of a matrix divided into 5×5 blocks on an Alliant FX/4. We observe execution time is reduced from 3.78 seconds to 2.07 seconds, a gain of 55% due to l DSYRK implementation. In the following we will only consider results of DLLTB d DSYRK. From Fig. 7, it is not clear which tasks are waiting for each other. Let us : allocation of tasks. L_{43}, A^3_{44}, and L_{44} can be executed on the same processor, for)rocessor P_1. Concurrently, A^2_{53}, L_{53}, and A^3_{55} can be run on processor P_3. From endency graph of Fig. 2 we know that A^3_{54}, L_{54}, A^4_{55}, and L_{55} cannot be executed 'hese jobs can run on P_4. In that case, we obtain a picture similar to Fig. 3 with . tasks. By this rearrangement of tasks no holes are saved, because all of them arise :pendency. It can easily be concluded now, that the sooner A^2_{53} starts the sooner mputation ends. It turns out that the update of the diagonal blocks is very crucial ıe. If the computation of such blocks and their preceding tasks are performed as ble then less holes will occur.

ıtion of tasks as suggested above provides that succeeding tasks operate on the r at least needs data calculated in its preceding task. A possibility to force reuse of oncatenate tasks. For the computation of L_{lj} $(l = j, \ldots, \kappa)$ from the original block re required (see formulas (2.6a)–(2.6d)). Assume that these steps are performed in : S_{lj}. The execution of such a supertask cannot begin before $L_{j,1:j}$ and $L_{l,1:j-1}$ have d. Moreover, the execution time of the supertasks increases with the value of j. most expensive tasks like $S_{\kappa-1,\kappa-1}$, $S_{\kappa,\kappa-1}$, and $S_{\kappa,\kappa}$ are strongly data-dependent un with any other job concurrently. Summarizing, the application of supertasks will legree of parallelism considerably, and it is not expected that the reuse of data will this loss.

llustrates the scheduling on the IBM 3090/VF on six processors with $\kappa = 7$. The execution time for the BLAS operations is more obvious; the rank-k update is the

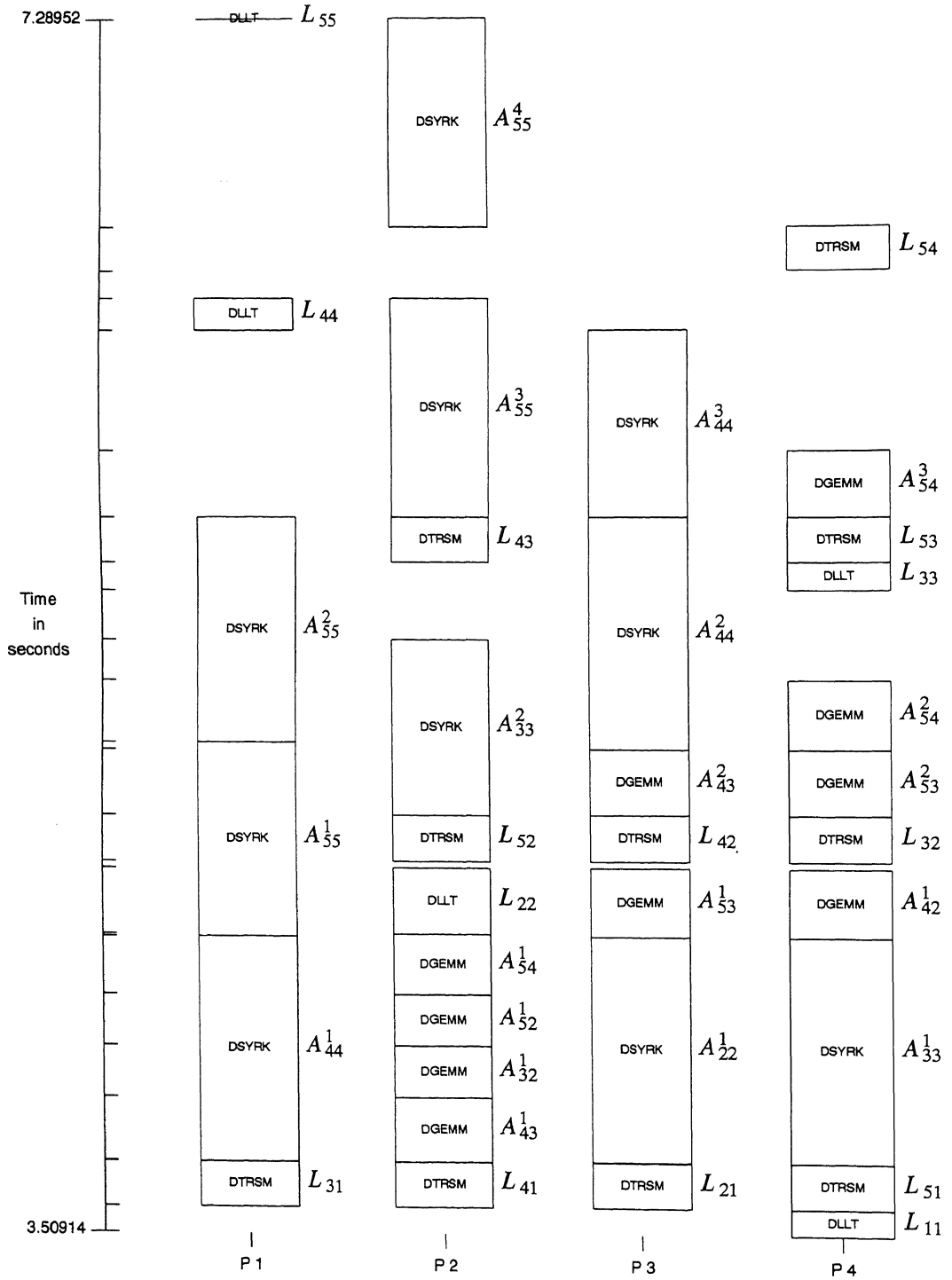


Fig. 6. Scheduling of the Cholesky factorization by SCHEDULE of a matrix of order 400 partitioned into 5×5 blocks for the Alliant FX/4 on four processors.

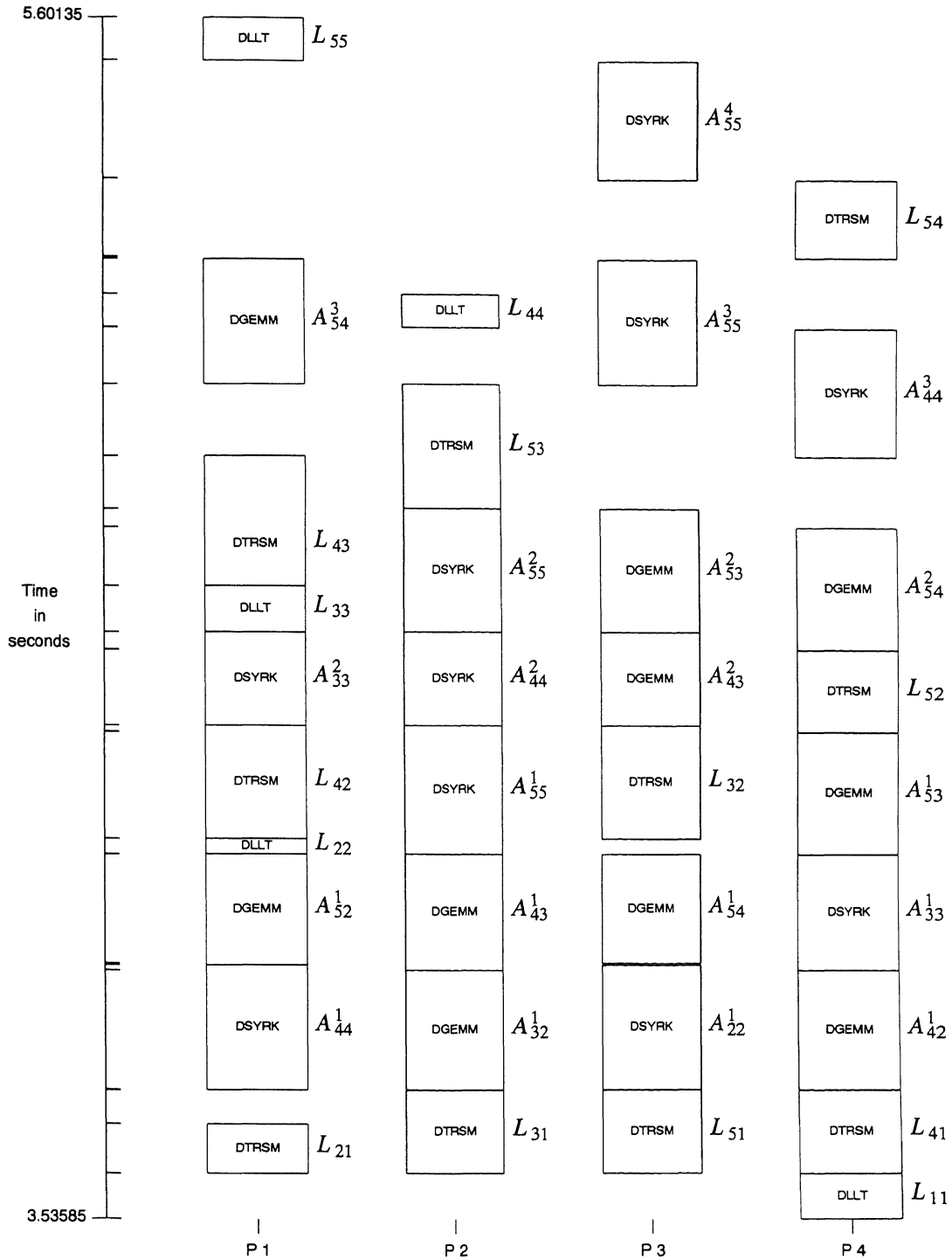


Fig. 7. Scheduling of the Cholesky factorization by SCHEDULE of a matrix of order 400 partitioned into 5×5 blocks on four processors with improved DSYRK based on intrinsic function DOTPRODUCT for an Alliant FX/4 on four processors.

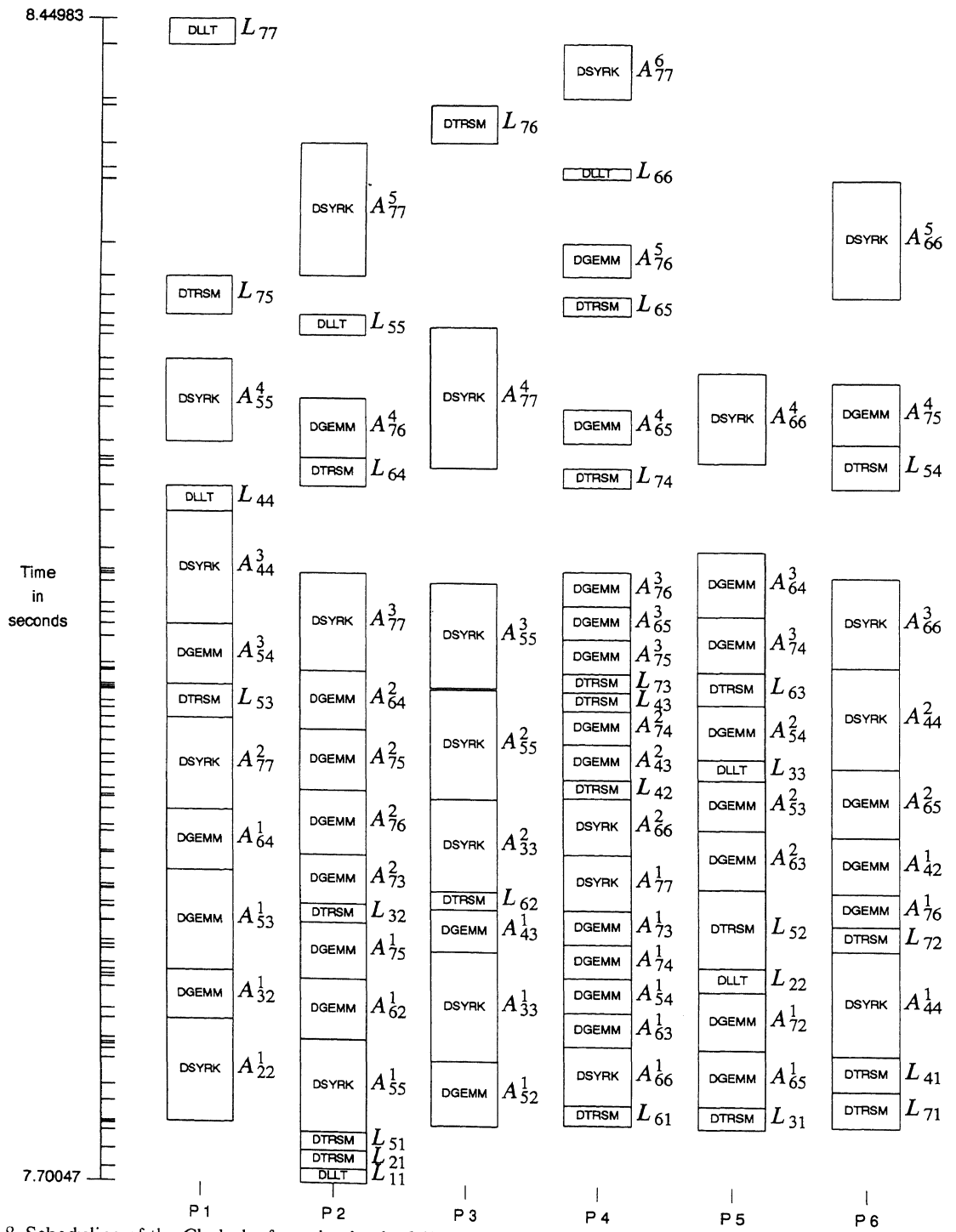


Fig. 8. Scheduling of the Cholesky factorization by SCHEDULE of a matrix of order 672 partitioned into 7×7 blocks for the IBM 3090/VF on six processors.

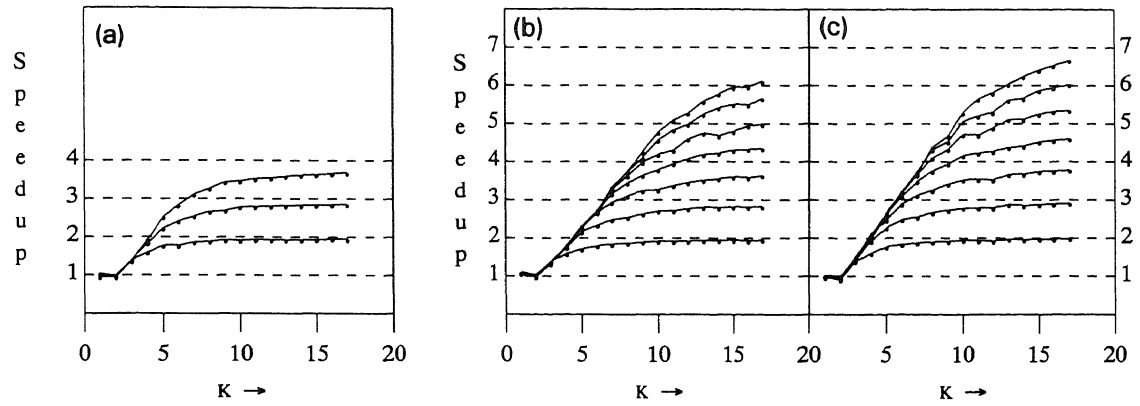


Fig. 9. Speedup for 2–4 processors on Alliant FX/4, NB = 64 (a). Speedup for 2–8 processors on Alliant FX/8, NB = 32 (b) and NB = 96 (c).

most expensive operation, since an optimal vendor-supported implementation of routine DSYRK is not available. Furthermore, we remark that not until three jobs have been completed on processor P_2 other processors start to execute. This does not correspond to the data dependency graph; the jobs L_{i1}, $i = 2, \ldots, 7$, become executable simultaneously. We do not precisely understand why this only happens at the start of the execution.

The values at the top right of each node in Fig. 2 represent the theoretical time, which is needed to compute the corresponding task, measured from the start of execution. From Figs. 6–8, we conclude that a good estimate for the expected computational time cannot be made, since the length of the blocks vary too much. Finally we remark that the trace facility, producing output for pictures like Figs. 6–8 was only used to analyze the course of execution and it was not used in timing programs, since it acts upon the execution time.

5.3. Performances of Cholesky factorizations for the Alliant FX/4 and FX/8

Unfortunately, the number of active tasks, that can be handled by SCHEDULE [12], is restricted to 1000. From the number of tasks M given by formula (3.2), we derive that K, presenting a partitioning into K × K blocks, may not exceed 17. The number of jobs is of order K^3, which implies that an extension of the array lengths of SCHEDULE hardly conduces to a larger K value.

Figure 9(a) displays the speedup (cf. formula (3.4)) obtained for $p = 2, 3, 4$ processors on an Alliant FX/4. The blocksize for this experiment is 64, and K varies from 1 up to 17. In Figs. 9(b)–(c), the speedup for the eight-processor Alliant FX/8 is shown for NB = 32 and NB = 96, respectively. For the theoretical case, the speedup is independent of the blocksize. In practice, however, the speedup will increase when the blocksize increases, since the overhead of scheduling, such as the creation of the data dependency graph, will proportionally decline to the total computation time. Recall that the overhead of scheduling is minimal for one processor, since processes never become data-dependent.

In Fig. 10 the Mflops obtained for the Alliant FX/4 for DLLT (Level 2 BLAS), DLLT3 (Level 3 BLAS) and DLLTB (SCHEDULE combined with Level 3 BLAS) are listed. The speed of the unblocked DLLT is, of course, independent of the number of blocks, but it does depend on the

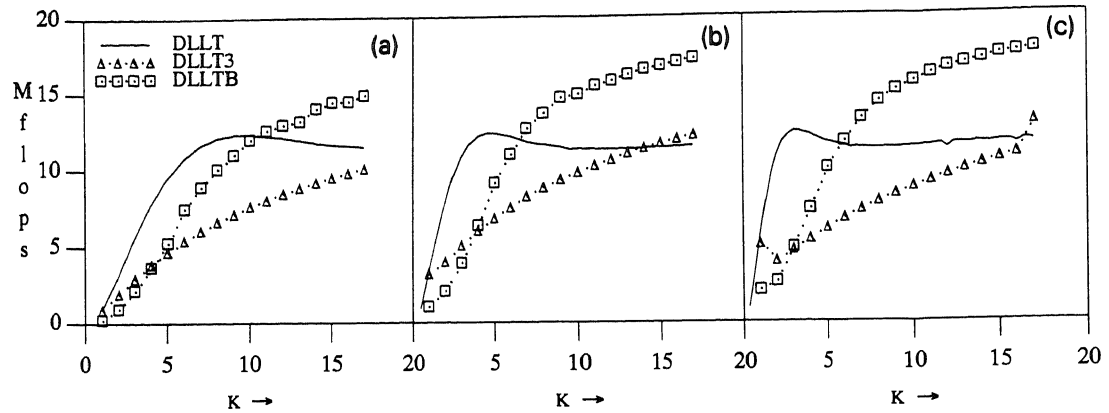


Fig. 10. Performance of DLLT, DLLT3, DLLTB on Alliant FX/4, $p = 4$, NB = 32 (a), NB = 64 (b), and NB = 96 (c).

matrix order. This declares why its shape differs in each picture. The maximum performance of DLLT is reached for $n = 256$. We remark that a call to DLLT3 with K = 1 corresponds to a single call to DLLT. The same is true for DLLTB. However, in that case, DLLT will be performed on a single processor.

In Fig. 11(a), presenting the results for both DLLT and DLLT3 on the Alliant FX/4, it can be seen that for large n a blocked implementation is to be preferred to an unblocked one (see Gallivan et al. [8] and Dayde and Duff [2]). We expect that, also for small matrices, DLLT3 in combination with an improved DSYRK will give higher performance than DLLT based on Level 2 BLAS. Figure 11(b) gives the results for DLLTB for the Alliant FX/4. Again a blocksize of 32 gives rise to the highest efficiency. The question arises whether the speed is completely determined by K, as in the theoretical case, or by the blocksize NB as well. Figure 12 illustrates that on one and two processors a blocksize of 64 results in higher performances than a blocksize of 48, despite its smaller K value.

The speed of DLLT3 on the Alliant FX/8 (Fig. 13(a)) is very disappointing, probably due to the low performance of the BLAS routine DTRSM (cf. Fig. 5(b)). Figure 13(b) presents the performance of DLLTB on the Alliant FX/8 based on exactly the same BLAS implementation as

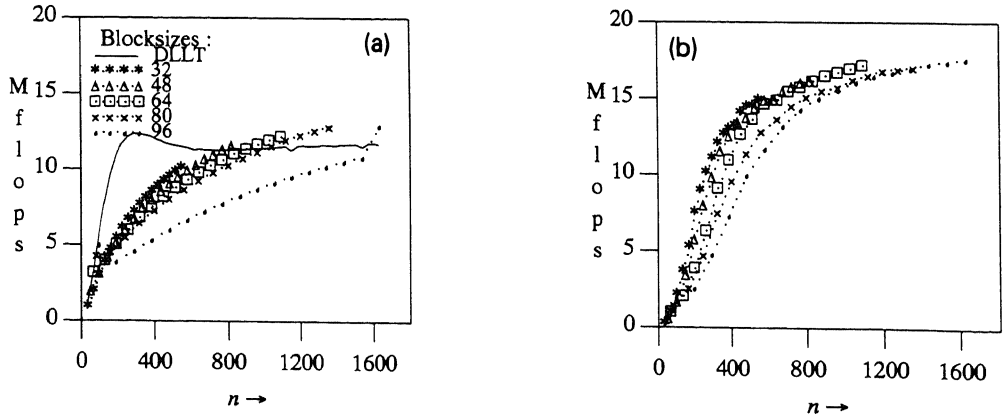


Fig. 11. Performance of DLLT and DLLT3 (a) and DLLTB (b) on Alliant FX/4, $p = 4$.

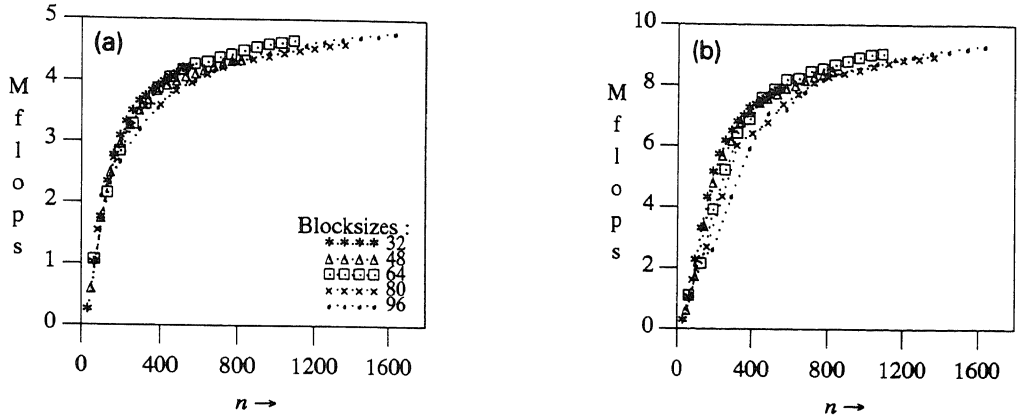


Fig. 12. Performance of DLLTB on Alliant FX/4, $p = 1$ (a) and $p = 2$ (b).

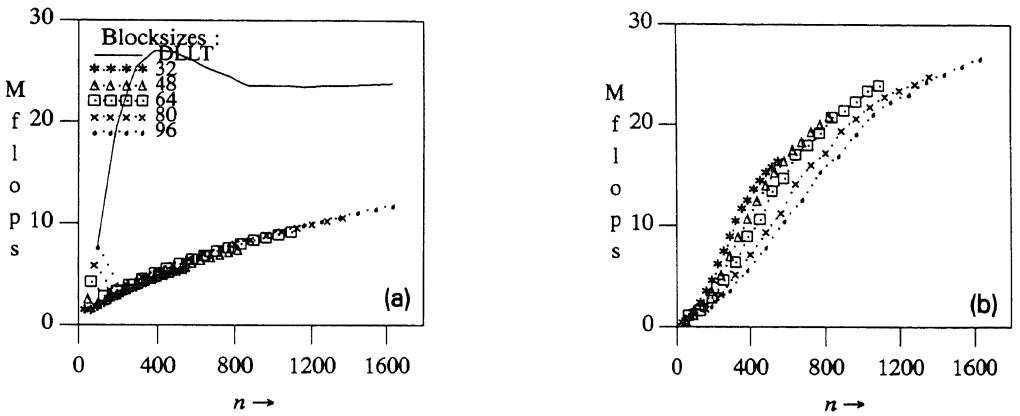


Fig. 13. Performance of DLLT and DLLT3 (a) and DLLTB (b) on Alliant FX/8, $p = 8$.

used for DLLT3. The large difference in performance between DLLT3 and DLLTB is also caused by the different data structure. To illustrate this we have experimented with DLLTB, where the matrix to be factorized was stored blockwise (four-dimensional) as well as in the traditional FORTRAN way (two-dimensional). The results for the Alliant FX/4, including the number of page faults and swaps and the elapsed time, are listed in Table 3.

For all experiments we discussed up till now, the matrix order n was a multiple of the blocksize NB. This yields that all blocks are of order NB. If not, then the submatrices of the last

Table 3
Alliant FX/4, influence of data structure

	Number of blocks	NB	Mflops	Number of page faults and swaps	Elapsed time
DLLTB (4-dim)	17	96	17.5	31 pf + 0w	3:03
DLLTB (2-dim)	17	96	15.1	1707 pf + 2w	18:38
DLLT3 (2-dim)	17	96	12.8	553 pf + 1w	19:37

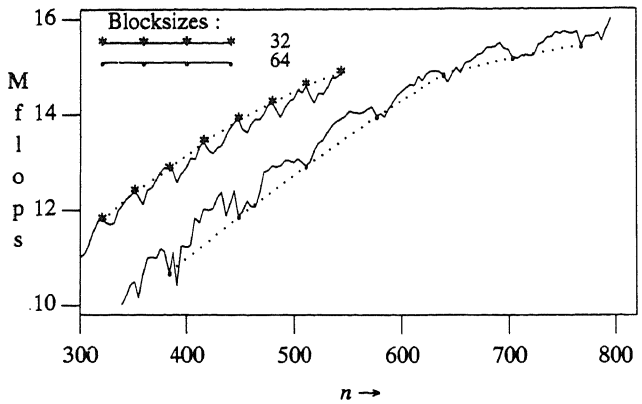


Fig. 14. Performance of DLLTB with unequally sized blocks on Alliant FX/4.

row K are of dimension $n-(K-1)\times NB$ by NB, whereas the diagonal block is of order $n-(K-1)\times NB$. As a consequence, the execution time of operations on such blocks will differ from the time required for square blocks of order NB. Figure 14 displays the performance for matrices of order $n = 300, 304, \ldots, 800$ and for NB = 32 and NB = 64. The dotted lines connect the points with $n = K \times NB$ and K a positive integer. We see that for NB = 32 the highest performance is obtained for such points. For NB = 64 the opposite is true; subroutine DLLTB performs even better in case of unequally sized blocks.

5.4. *Performances of Cholesky factorizations for the IBM 3090 / VF*

For the IBM 3090/VF optimized Level 2 and 3 BLAS implementations are available only for a single processor. Unfortunately, only an older version of SCHEDULE [7] is available. It turns out that only the results for $K \leqslant 15$ are correct. We do not go into further detail in this paper. For the behaviour of DLLTB on a single processor we refer to Fig. 15(a). In Figs. 15(b)–(c) the speedup for the IBM 3090/VF is given. For small values of K the speedup is less

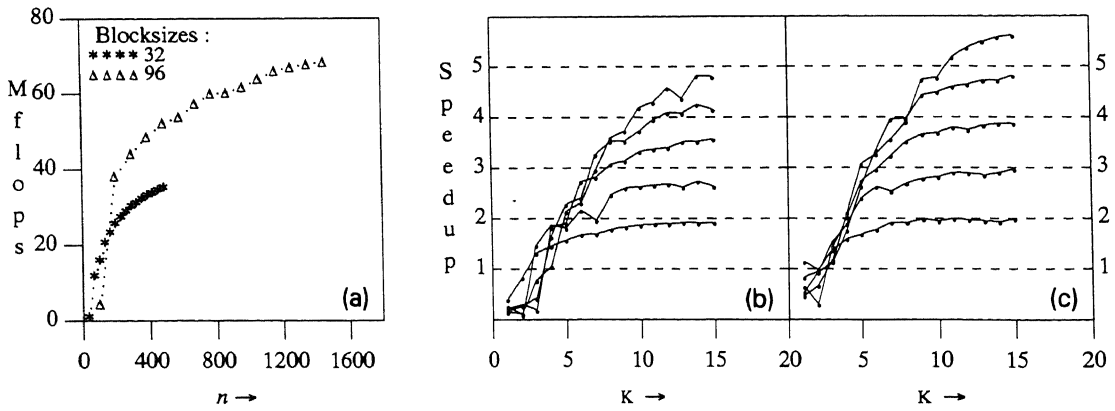


Fig. 15. Performance of DLLTB on IBM 3090/VP, $p = 1$ (a). Speedup for 2–6 processors on IBM 3090/VF, NB = 32 (b) and NB = 96 (c).

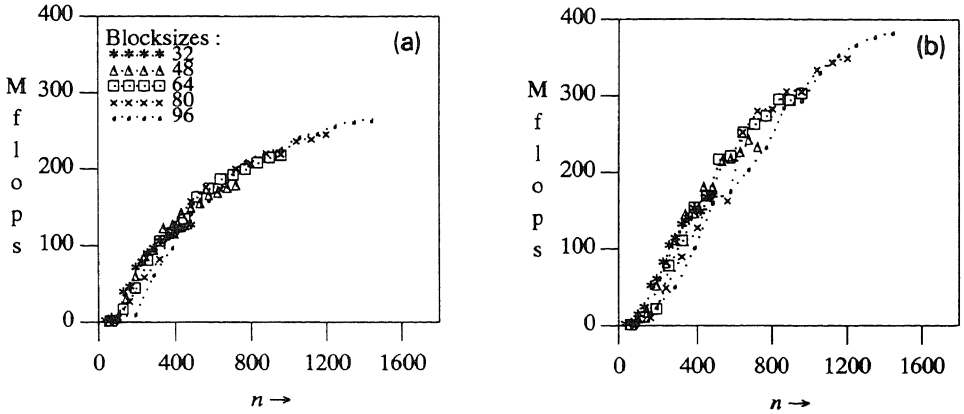


Fig. 16. Performance of DLLTB, $p = 4$ (a) and $p = 6$ (b) processors IBM 3090/VF.

than 1, probably caused by the "slow" communication between the processors. The more processors involved in the process the lower the speedup for small values of K. According to Fig. 9 the speedup increases with the blocksize.

In Fig. 16 we give the results of DLLTB on four and six processors, respectively. We observe that the performance strongly depends on the blocksize, whereas for the Alliant FX/4 (cf. Fig. 11(b)) and the Alliant FX/8 (cf. Fig. 13(b)) the value of K mainly influences the speed. Especially the timings in Fig. 16(a) point to a slow communication between the processors in proportion to the IBM BLAS performance.

6. Conclusions and remarks

We conclude that parallelism over the blocks is a useful way to achieve high efficiency. In this paper we focussed on the Cholesky factorization, but our technique can also be applied to other problems in linear algebra. The use of the SCHEDULE package helps to introduce parallelism in a transportable way. For the machines on which we ran our code, i.e., the Alliant FX/4, the Alliant FX/8, and the IBM 3090/VF, high performances are obtained. On the Alliant machines higher Mflop rates are achieved for our code which applies parallelism over the kernels than for codes exploiting parallelism within the BLAS kernels. Moreover, the amount of data traffic has been reduced; for each processor at most three submatrices of order NB are needed at a time. This happens in case of a _GEMM operation, other operations need less data. If these submatrices are explicitly stored blockwise, then for a suitable blocksize data can be kept in cache. For DLLT2 and DLLT3 the situation is different. Here, the data needed per operation is not bounded by the blocksize. The data management can only be organized within a BLAS kernel and it is hoped that this is well done by the manufacturer (cf. Fig. 5). One important side effect on the different data access pattern we would like to mention here. Not only was the CPU time for DLLTB less than for DLLT (Level 2 BLAS) and DLLT3 (Level 3 BLAS), but also the wall clock time was much shorter.

Nevertheless, the performance of DLLTB based on SCHEDULE in combination with tuned BLAS can still be increased: firstly, when a more efficient BLAS particularly tuned for a single

processor can be used, and secondly, when a better scheduling of the tasks can be applied. A partitioning into more than 17×17 blocks must be possible. We believe that the amount of parallelism is potentially high enough to experiment with other computation orderings which may result in a higher performance.

To achieve high performance for algorithms based on parallelism over the kernels, optimized BLAS for a single processor is needed, the so-called nonparallel BLAS. The reason for this is that a highly tuned parallelized BLAS implementation will not always perform optimally on a single processor. Nowadays, it is not always clear whether a given BLAS version has been parallelized or not. For the machines discussed in this paper either a parallel or a nonparallel BLAS version is available. Therefore we suggest to distinguish between parallel and nonparallel BLAS implementations. We believe that both versions should be accessible for exploiting parallelism over and within the BLAS kernels.

Acknowledgement

The author would like to thank Dik T. Winter for executing the code on the IBM 3090/VF and for rectifying the SCHEDULE version on that machine.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammerling, A. McKenney and D.C. Sorenson, LAPACK: a portable linear algebra library for high-performance computers, University of Tennessee, CS-90-105 (1990).

[2] M.J. Dayde and I.S. Duff, Use of parallel Level 3 BLAS in LU factorization on three vector multiprocessors; the Alliant FX/80, the CRAY-2, and the IBM 3090 VF, in: *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam (1990).

[3] J.J. Dongarra, J. Du Croz, I. Duff and S. Hammerling, A set of level 3 basic linear algebra subprograms, *ACM Trans. Math. Software* 16 (1) (1990) 1–17.

[4] J.J. Dongarra, J. Du Croz, S. Hammerling, and R.J. Hanson, An extended set of Fortran basic linear algebra subprograms, *ACM Trans. Math. Software* 14 (1) (1988) 1–32.

[5] J.J. Dongarra, I.S. Duff, D.C. Sorensen and H.A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers* (SIAM, Philadelphia, PA, 1991).

[6] J.J. Dongarra, F.G. Gustavson and A. Karp, Implementing linear algebra algorithms for dense matrices on a vector pipeline machine, *SIAM Rev.* 26 (1984) 91–112.

[7] J.J. Dongarra and D.C. Sorensen, Schedule: tools for developing and analyzing parallel Fortran programs, Argonne National Laboratory Report, ANL-MCS-TM-86 (1986).

[8] K. Gallivan, W. Jalby and U. Meier, The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory. Timely communications, *SIAM J. Sci. Statist. Comput.* 8 (1987) 1079–1084.

[9] K. Gallivan, W. Jalby, U. Meier and A. Sameh, Impact of hierarchical memory systems on linear algebra algorithm design, *Internat. J. Supercomput. Appl.* 2 (1988) 12–48.

[10] A. George, M.T. Heath and L. Liu, Parallel Cholesky factorization on a shared-memory multiprocessor, *Linear Algebra Appl.* 77 (1986) 165–187.

[11] G.H. Golub and C.F. Van Loan, *Matrix Computations* (The Johns Hopkins Press, Baltimore, MD, 2nd ed., 1989).

[12] F.B. Hanson and D.C. Sorensen, The SCHEDULE parallel programming package with recycling job queues and iterated dependency graphs, Argonne National Laboratory Report, ANL-MCS-P22-0189 (1989).

[13] W. Jalby and U. Meier, Optimizing matrix operations on a parallel multiprocessor with a hierarchical memory system, in: *Proceedings of the 1986 International Conference on Parallel Processing*, St. Charles (1986).
[14] M. Louter-Nool and D.T. Winter, Benchmark of the initial release of the LAPACK library, Note NM-N8903+ supplement, CWI, Amsterdam (1989).
[15] E. Luque, A. Ripoll, P. Hernandez and T. Margalef, Impact of task duplication on static-scheduling performance in multiprocessor systems with variable execution-time tasks, in: *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam (1990).
[16] J.M. Ortega, The *ijk* forms of factorization methods I. Vector computers, *Parallel Comput.* 7 (1988) 135–147.
[17] J.M. Ortega and C.H. Romine, The *ijk* forms of factorization methods II. Parallel systems, *Parallel Comput.* 7 (1988) 149–162.