Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

M. Louter-Nool

Block-Cholesky for parallel processing

Department of Numerical Mathematics Report NM-R9023 December

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

Copyright © Stichting Mathematisch Centrum, Amsterdam

Block-Cholesky for Parallel Processing

Margreet Louter-Nool
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

We concentrate on the Cholesky factorization of $A = L\ L^{T}$, 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.

1980 Mathematics subject classification (1985 Revision): 65F05,65W05,68M20.
1980 CR Categories: 5.14
Keywords and Phrases: Cholesky factorization, Block algorithms, Parallelism, Scheduling

Note: This report will be submitted for publication elsewhere.

1. INTRODUCTION

We discuss the Cholesky factorization

$$A = L\ L^{T},$$

with A a positive definite symmetric 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[11]. 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[6,9,10]. It is a straight-forward algorithm and, since A is positive definite, pivoting is not necessary to ensure or improve numerical stability. The general step for the Cholesky factorization looks like :

```
FOR-----
    FOR-----
        FOR----
```

$$a_{ij} = a_{ij} - l_{ik}\ .\ a_{kj} \qquad\qquad (l_{ik} = a_{ik}\ /\ a_{kk})$$

and it depends on the position of the loop parameters i, j, and k, respectively, whether we are dealing with

Report NM-R9023
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Row, Column or Submatrix Cholesky factorization[6,14,15]. 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[14] and parallel machines[15]. Column Cholesky appeared to be favorite for machines with vector-processing capabilities. For parallelism only local memory systems with row or column wrapped interleaved storage are considered[15]. We focus on block-Column Cholesky[5]. For convenience, we assume that the blocksize NB is a proper divisor of matrix order n and

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

The block factorization can be visualized as:

$$\begin{bmatrix} A_{1:j-1,1:j-1} & & \\ A_{j,1:j-1} & A_{jj} & \\ A_{j+1:K,1:j-1} & A_{j+1:K,j} & A_{j+1:K,j+1:K} \end{bmatrix} = \tag{2.2}$$

$$\begin{bmatrix} L_{1:j-1,1:j-1} & & \\ L_{j,1:j-1} & L_{jj} & \\ L_{j+1:K,1:j-1} & L_{j+1:K,j} & L_{j+1:K,j+1:K} \end{bmatrix} \cdot \begin{bmatrix} L^T_{1:j-1,1:j-1} & L^T_{j,1:j-1} & L^T_{j+1:K,1:j-1} \\ & L^T_{jj} & L^T_{j+1:K,j} \\ & & L^T_{j+1:K,j+1:K} \end{bmatrix}$$

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

$$A_{jj} = L_{j,1:j-1} \cdot L^T_{j,1:j-1} + L_{jj} \cdot L^T_{jj} \tag{2.3a}$$

$$A_{j+1:K,j} = L_{j+1:K,1:j-1} \cdot L^T_{j,1:j-1} + L_{j+1:K,j} \cdot L^T_{jj} \tag{2.3b}$$

From the first equation (2.3a) L_{jj} can be calculated. The operations involved are :

1. a symmetric rank-k update

$$A^{(1)}_{jj} \leftarrow A_{jj} - L_{j,1:j-1} \cdot L^T_{j,1:j-1} \tag{2.4a}$$

2. a Cholesky factorization on a single block

$$L_{jj} \leftarrow \mathit{Cholesky}(A^{(1)}_{jj}) \tag{2.4b}$$

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

3. a matrix-matrix product

$$A^{(1)}_{j+1:K,j} \leftarrow A_{j+1:K,j} - L_{j+1:K,1:j-1} \cdot L^T_{j,1:j-1} \tag{2.4c}$$

4. finally, we have to solve a triangular system

$$L_{j+1:K,j} \leftarrow A^{(1)}_{j+1:K,j} \cdot L^{-T}_{jj} \tag{2.4d}$$

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

$$A^{(1)}_{jj} \leftarrow A_{jj} - \sum_{i=1}^{j-1} L_{ji} . L^T_{ji} \tag{2.5a}$$

$$A^{(1)}_{lj} \leftarrow A_{lj} - \sum_{i=1}^{j-1} L_{li} . L^T_{ji} \qquad l = j+1, \cdots, K \tag{2.5c}$$

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

$$B^i_j = L_{ji} \,.\, L^{\mathrm{T}}_{ji} \text{ and } C^i_{lj} = L_{li} \,.\, L^{\mathrm{T}}_{ji} \qquad i = 1, \cdots, j-1;\ l = j+1, \cdots, \mathrm{K}$$

are generated. In the end, the matrices B^i_j and C^i_{lj} have to be subtracted from A_{jj} and A_{lj}, respectively. An alternative way to perform the j-th step of the factorization is to translate (2.4a-d) into

$$A^i_{jj} \leftarrow A^{i-1}_{jj} - L_{ji} \,.\, L^{\mathrm{T}}_{ji} \qquad i = 1, \cdots, j-1 \tag{2.6a}$$

$$L_{jj} \leftarrow Cholesky(\, A^{j-1}_{jj} \,) \tag{2.6b}$$

$$A^i_{lj} \leftarrow A^{i-1}_{lj} - L_{li} \,.\, L^{\mathrm{T}}_{ji} \qquad i = 1, \cdots, j-1;\ l = j+1, \cdots, \mathrm{K} \tag{2.6c}$$

$$L_{lj} \leftarrow A^{j-1}_{lj} \,.\, L^{-\mathrm{T}}_{jj} \qquad l = j+1, \cdots, \mathrm{K} \tag{2.6d}$$

where A^0_{ij} 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 not 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-d) has been broken up into smaller units of computations. We will show that it is not 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-d) 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 $\mathrm{K} \times \mathrm{K}$ blocks. Both A and the diagonal blocks are symmetric. From (2.6a-d) we derive the number of processes needed to compute the complete factorization of A:

$$\begin{aligned} &\text{_LLT} &&: \mathrm{K} \\ &\text{_SYRK} &&: \tfrac{1}{2}\,\mathrm{K}\,(\,\mathrm{K}-1\,) \\ &\text{_TRSM} &&: \tfrac{1}{2}\,\mathrm{K}\,(\,\mathrm{K}-1\,) \\ &\text{_GEMM} &&: \tfrac{1}{6}\,\mathrm{K}\,(\,\mathrm{K}-1\,)\,(\,\mathrm{K}-2\,). \end{aligned} \tag{3.1}$$

This implies that the total number of tasks will be

$$\mathrm{M} = \frac{1}{6}\,\mathrm{K}\,(\,\mathrm{K}+1\,)\,(\,\mathrm{K}+2\,) \tag{3.2}$$

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

$$L_{\mathrm{M}} = \left\{ T_1, T_2, \cdots, T_{\mathrm{M}} \right\} \tag{3.3}$$

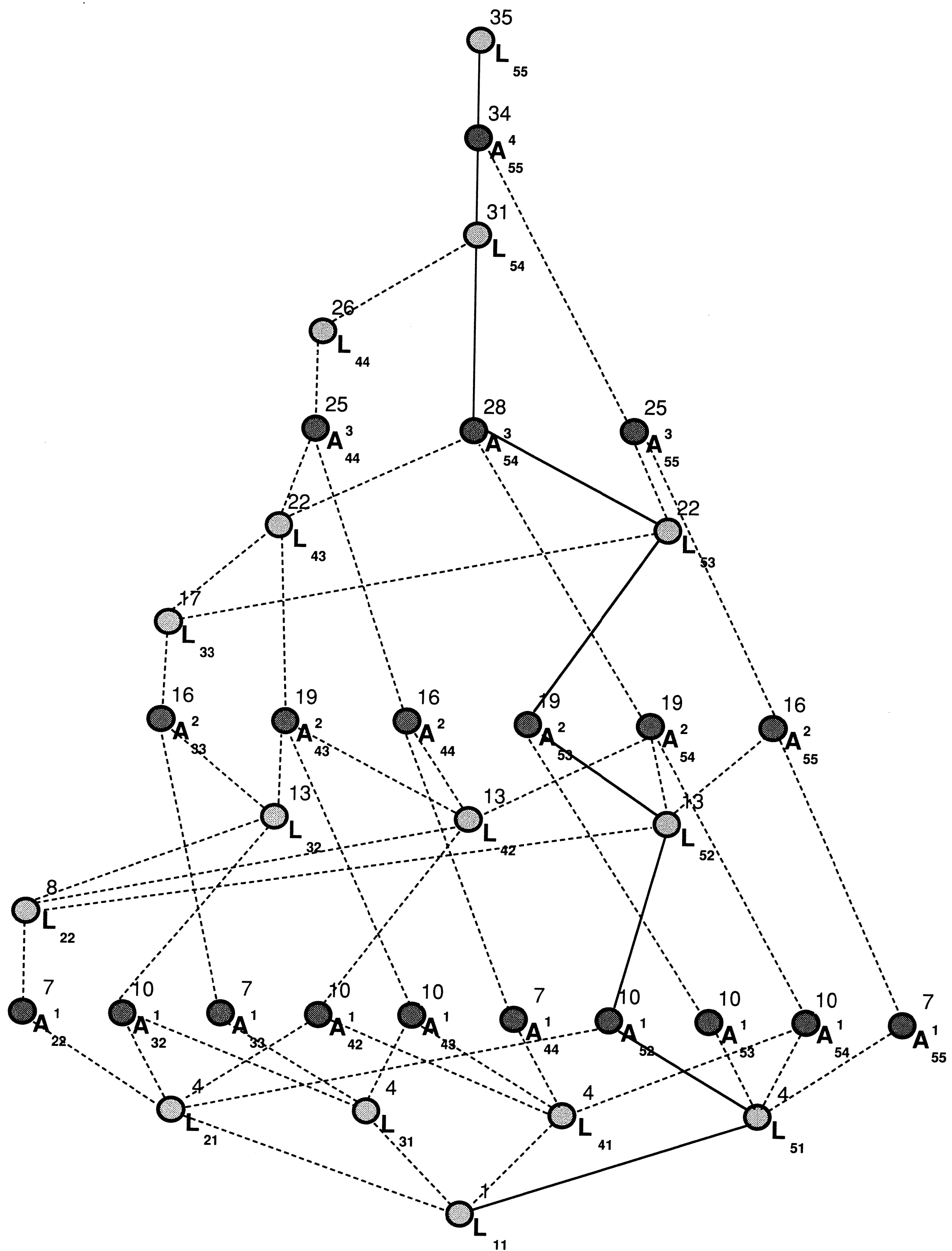

FIGURE 3.1, The data dependency graph for the Cholesky factorization of a matrix partitioned into 5×5 blocks on 4 processors.

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 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.

During execution we must know the status of a task. For that purpose an integer value IREADY is assigned to each task T_i to indicate its status:

IREADY=−1 : this process has been successfully completed.

IREADY=0 : schedulable process. This means that its data dependencies have all been satisfied.

IREADY=+1 : non-schedulable process.

The aim of our investigations is to apply a simple scheduling strategy and to find an optimal value for K, the partitioning parameter (3.1). To obtain a good speedup with a multitasked code, we have to keep the processors concurrently active and have to reduce 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, K−1 tasks become schedulable, namely the calls to _TRSM on the first column matrices $A_{j1}, j = 2, \cdots, K$. Assume that $p \leq K-1$, then we continue with the next p schedulable tasks. For $p > K-1$ only K−1 processors can be active and $p-(K-1)$ processors will still be idle.

B. If a process has finished on processor P_α then its IREADY value becomes -1. Other tasks may become schedulable and their IREADY value will be changed to zero. The next task on processor P_α will be the first ready task in list L_M, i.e., the first one with IREADY = 0. 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 3.1 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

$A_{ij}{}^k$: k-th temporary update of submatrix A_{ij} ($k \leq j-1$)

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 Figure 3.1, namely that graph spanned by the nodes $A_{ij}{}^k$ and L_{ij} with $1 \leq j \leq i \leq 4$. 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 following computational costs

Operation	FLOPs	CU
_LLT	⅓ $NB^3 + \cdots$	1
_SYRK	$NB^3 + \cdots$	3
_TRSM	$NB^3 + \cdots$	3
_GEMM	$2\ NB^3 + \cdots$	6

TABLE 1 Theoretical execution times expressed in CU's

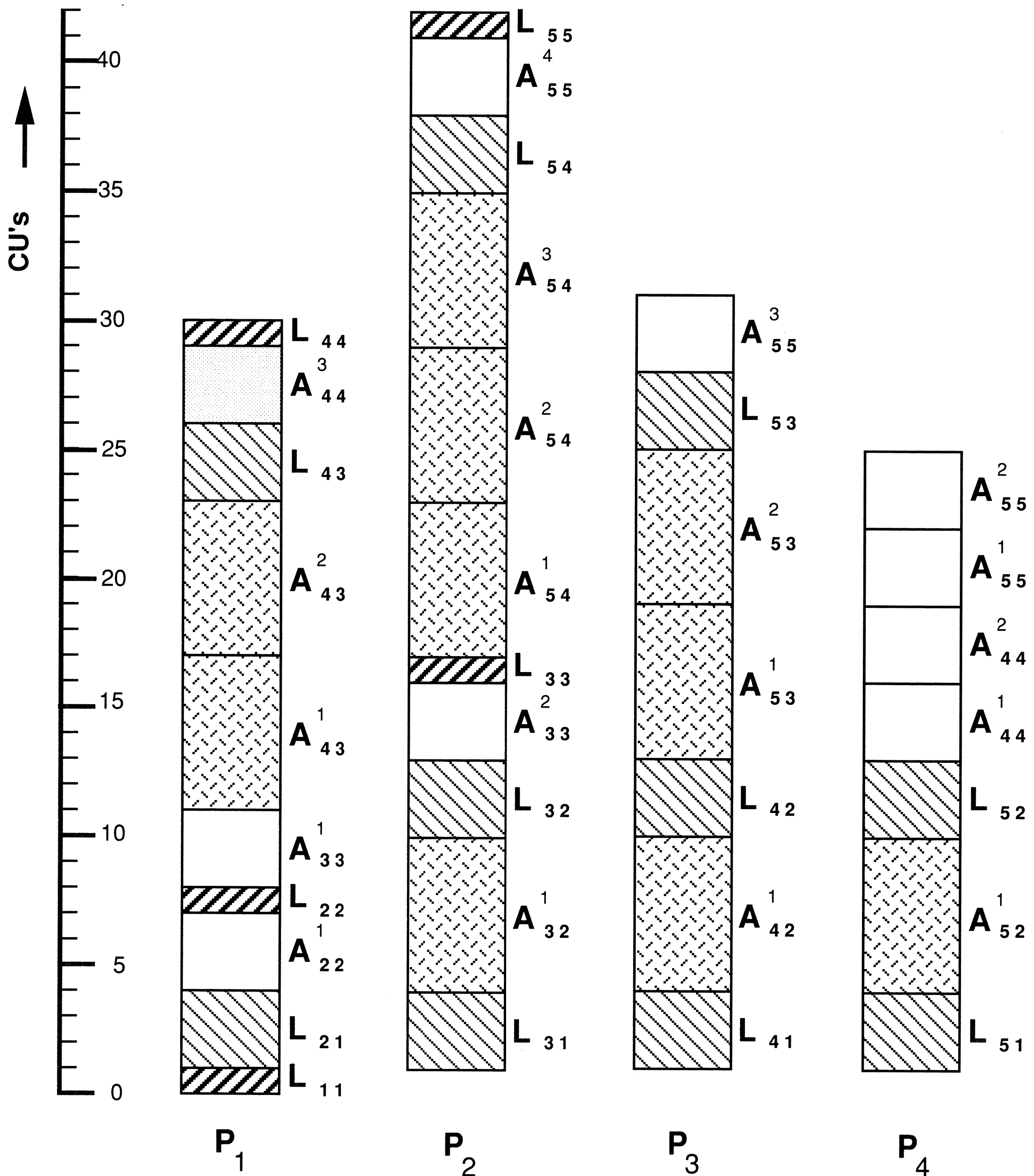

FIGURE 3.2, Scheduling of the Cholesky factorization of a matrix partitioned into 5 × 5 blocks on 4 processors.

for the different operations on equally sized blocks of order NB. The value at the top right of a node in Figure 3.1. stands for the minimal execution time expressed in CU's on an unbounded number of processors to perform the associated process and its preceding tasks. The solid line in Figure 3.1 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

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

From (3.1) and the values of Table 1 we may conclude that, for our example with K = 5, the maximum speedup is

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

assuming enough processors to be available. For the graph of Figure 3.1 the maximum number of processes which can be computed in parallel is 10, namely the first updates A_{22}^{1}, A_{32}^{1}, $\cdots$, A_{55}^{1} which cover the whole matrix except for the first column. Hence, on 10 or more processors, the efficiency can not exceed 35.7 %.

Let us return to the scheduling as described in this Section. Suppose 4 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 Figure 3.2. 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 10 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[13].

Both at the beginning and at the end of the factorization process, operations can not be performed in parallel. To minimize the execution time of the initial and final phase, a smaller value of the blocksize NB can 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 show in the next Section that machine specific aspects will play an important role in the performance, too.

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[11]. 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.

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. We will discuss this in Section 5.1. Moreover, the influence of possible reuse of the cache can hardly be expressed in terms of the above mentioned variables. In the next section, we focus on the scheduling by SCHEDULE, which rather differs from the scheduling we proposed in Section 3. We expect the theoretical values will not agree to the numerical results. Nevertheless, we attempt to determine the influence of the K, p, L_M and CU.

4.2. The scheduling by SCHEDULE

The data dependencies for SCHEDULE are described by (only) four parameters:

- JOBTAG - a unique user supplied identifier
- ICANGO - the number of processes that must be completed before a process can start.
- NCHEKS - specifies the number of processes that depend on this process
- MYCHKN - an integer array whose first NCHEKS entries contain the identifiers of the processes that depend on this process.

Obviously, the SCHEDULE package does not provide for assigning priorities to tasks, for example, to reuse cache (if possible) or to rank time-consuming tasks above less time-consuming tasks. Moreover, it turns out that the influence of a particular ordering can hardly be measured. We will briefly illustrate this. In order to reduce excessive referencing of the locks by the processors trying to get access to the ready queue of jobs ready to execute, the queue is partitioned into component queues or subqueues. So, each processor has its own READYQ of executable tasks and the tasks are equally distributed among the available processors (i.e., JOBTAG mod p, where p denotes the number of processors). Jobs that are ready for computation are added at the tail of one of the READYQ's. Processors get jobs of the ready queue at the head position. This technique is not sensible for any ordering. Of course, we can prescribe that the job with the lowest index has to be allocated first, but in that case all ready queues have to be considered. Only in case that no executable tasks are available, it is allowed to look for schedulable tasks in the READYQ's of other processors.

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 4 different execution times $T_{_GEMM}$, $T_{_TRSM}$, $T_{_SYRK}$ 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. After the factorization of the first diagonal block, p equal tasks _TRSM are compiled concurrently. It is not possible to say which of them will finish first. Consequently, it can not be predicted in which order the ready jobs are added to the ready queues. Therefore, the allocation to processors by SCHEDULE based on JOBTAG values makes it inconvenient to experiment with different orderings. In Section 5.2 we present a few examples of the scheduling by SCHEDULE.

5. NUMERICAL RESULTS

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 4-dimensional array

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

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 to DPOTRF from LAPACK[1]. The matrix to factorize is stored in the traditional FORTRAN way, which means columnwise in a 2-dimensional array. The routine DLLT3 exploits parallelism within the BLAS kernels. Operations are not performed on single blocks but on much larger blockmatrices. Figure 5 illustrates how such blocks are composed.

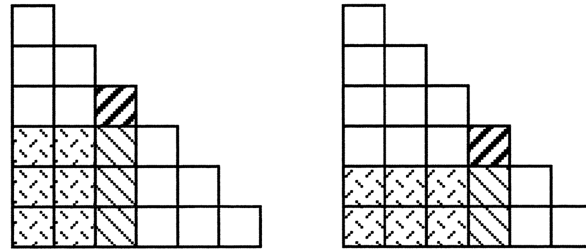

FIGURE 5, The combinations of blocks per step.

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.[9]).

The machines used in the numerical experiments are an Alliant FX/4 with 4 processors, an Alliant FX/8 with 8 processors and an IBM 3090/VF with 6 processors. The FX/4 is stationed at the CWI. The FX/8 at Argonne can be reached by using telnet. 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 accomplished in double precision.

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

Since the SCHEDULE tasks execute on a single processor, we consider the BLAS performance obtained for one processor. We have experimented with several blocksizes, viz., 32, 48, 64, 80, 96. Table 2 lists the results for the Alliant FX/4 and FX/8.

	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
	(2.5)	(2.6)	(2.1)	(2.7)	(2.4)					

TABLE 2 Performance of Alliant BLAS 3 kernels on a single processor.

For analysis it would be easy if the BLAS performances for both machines were comparable. Unfortunately, this is not true. The BLAS of the new Linear Algebra Library on the FX/4 is much faster than the one available on the FX/8. The performance of DGEMM and DTRSM on the Alliant FX/4 has significantly been enhanced compared to previous releases (see also Supplement of Louter-Nool and Winter[12]). The Mflops rates for DSYRK are a little disappointing. We have accelerated its speed by using the Alliant intrinsic function DOTPRODUCT. The improved values are also listed in Table 2 (between brackets).

Figures 5.1.1a-b show the BLAS performance on 4(FX/4) and 8(FX/8) processors. For each step j, we

measured the performance of the operations (2.3a), (2.3c) and (2.3d), denoted by DSYRK, DGEMM, and DTRSM, respectively. These BLAS operations add up to the main part of computation of DLLT3. The matrix A to be factorized was of order 1024 and was partitioned into 16×16 blocks of order 64. Figure 5.1.1b illustrates that the BLAS performance on the Alliant FX/8 strongly depends on j, whereas the results of the Alliant FX/4 are constant. Finally, we remark that for DLLT3 the alternative code for DSYRK based on the intrinsic DOTPRODUCT performs even worse than the vendor-provided code. DSYRK computes the innerproduct of 2 rows embedded in a matrix of large order (for our example, $n = 1024$). Cache effects completely dominate the performance of the alternative DSYRK.

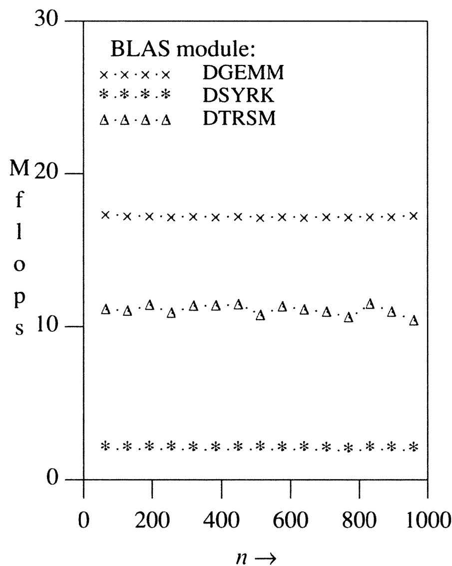

FIGURE 5.1.1a, Alliant FX/4, p=4
Performance Level 3 BLAS

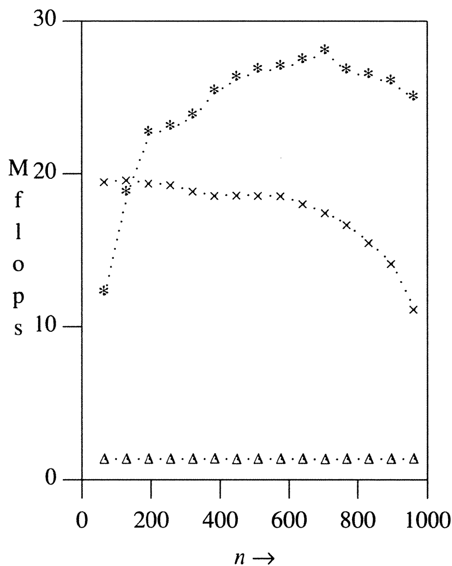

FIGURE 5.1.1b, Alliant FX/8, p=8
Performance Level 3 BLAS

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 workstation can interpret this output. It is possible to construct the dependency graph and to show the execution sequence that was run on a parallel machine. We used these output files to display the course of execution analogue to Figure 3.2. In this Section we present three of them.

Figure 5.2.1 is concerned with the computed factorization of a matrix divided into 5×5 blocks, and was executed on an Alliant FX/4 with $p = 4$. It is one of the first pictures we made using the SCHEDULE trace facility. One can see that the most significant difference between the theoretical example presented in Figure 3.2 and this picture is the proportion between the different BLAS operations. The length of a DSYRK block is about 4 times the length of a DGEMM block. This means that either the matrix-matrix product is very fast or the rank-k update performs extremely bad. From the previous Section we know by now that DSYRK can be improved.

Figure 5.2.2 demonstrates the scheduling based on the accelerated DSYRK. The BLAS blocks in this picture are more or less of the same length, since DGEMM reaches a higher speed than the other BLAS operations. The total execution time is reduced from 3.78 seconds to 2.07 seconds, a gain of 55%. Figure 5.2.2 displays some holes, which indicate that there are periods of idle waiting for executable tasks. It turns out, that the update of the diagonal blocks is very crucial in our scheme. If the computation of such blocks and their preceding tasks are performed as soon as possible then less holes will occur.

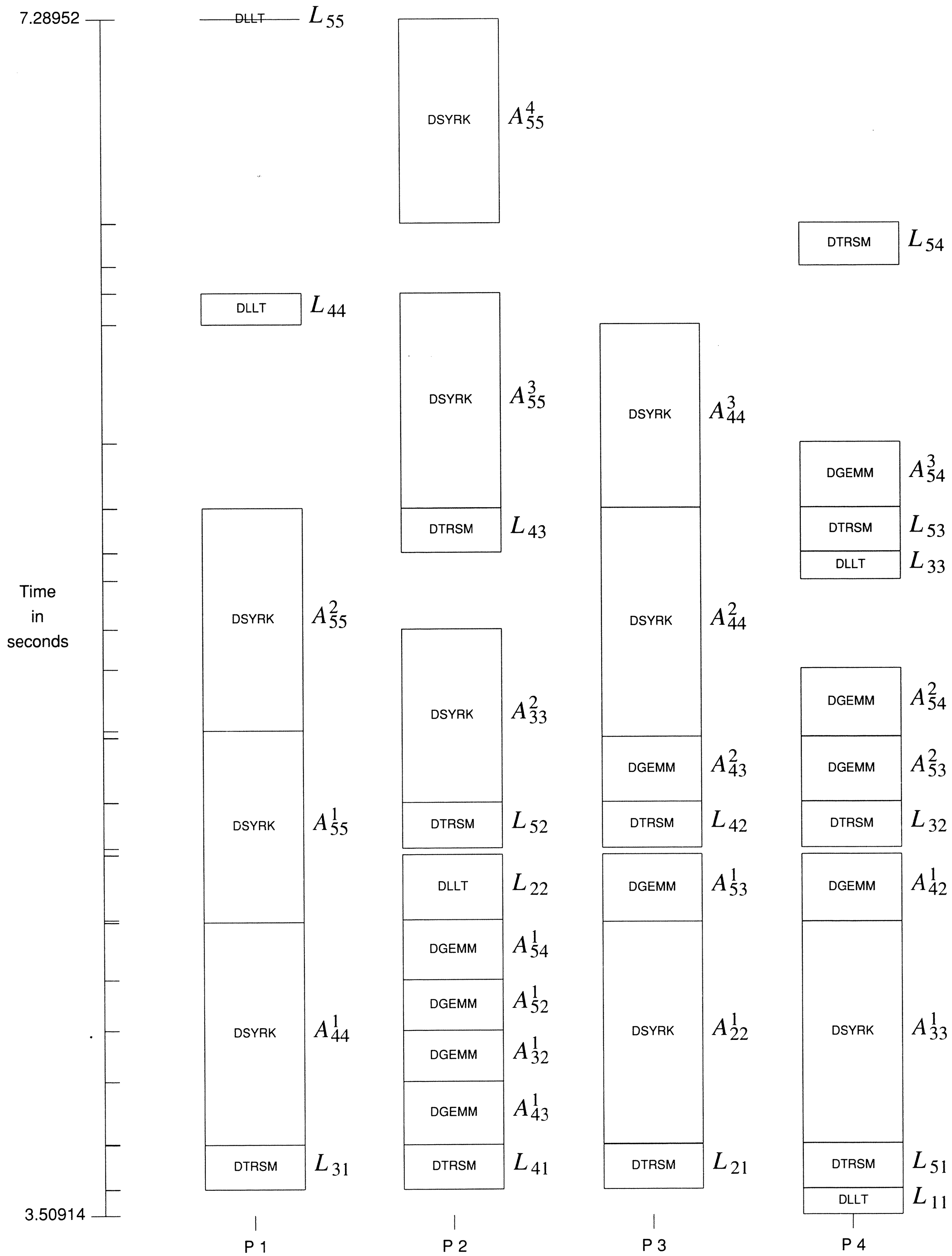

FIGURE 5.2.1, Scheduling of the Cholesky factorization by SCHEDULE of a matrix of order 400 partitioned into 5×5 blocks for the Alliant FX/4 on 4 processors.

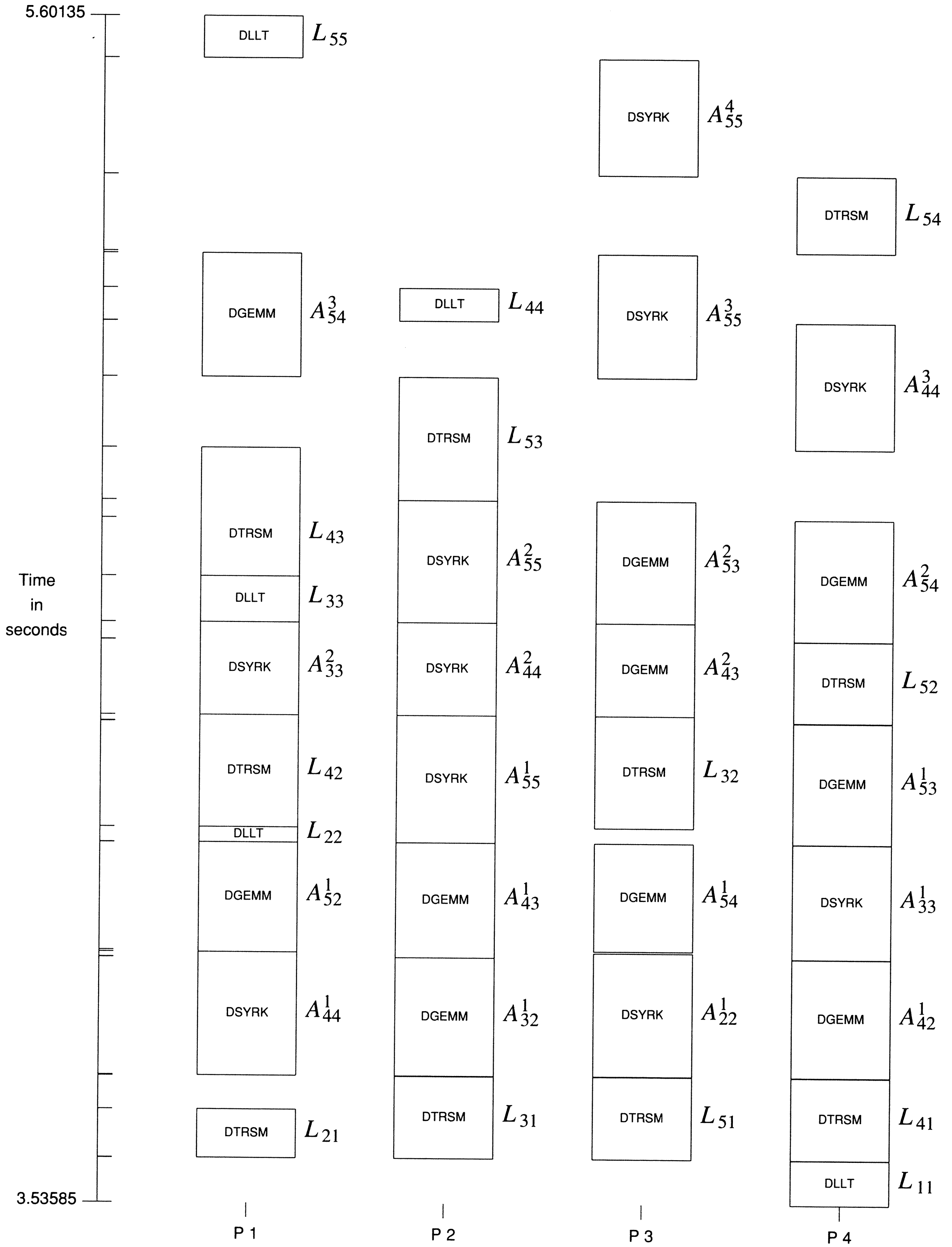

FIGURE 5.2.2, Scheduling of the Cholesky factorization by SCHEDULE of a matrix of order 400 partitioned into 5×5 blocks on 4 processors with improved DSYRK based on intrinsic function DOTPRODUCT for an Alliant FX/4 on 4 processors.

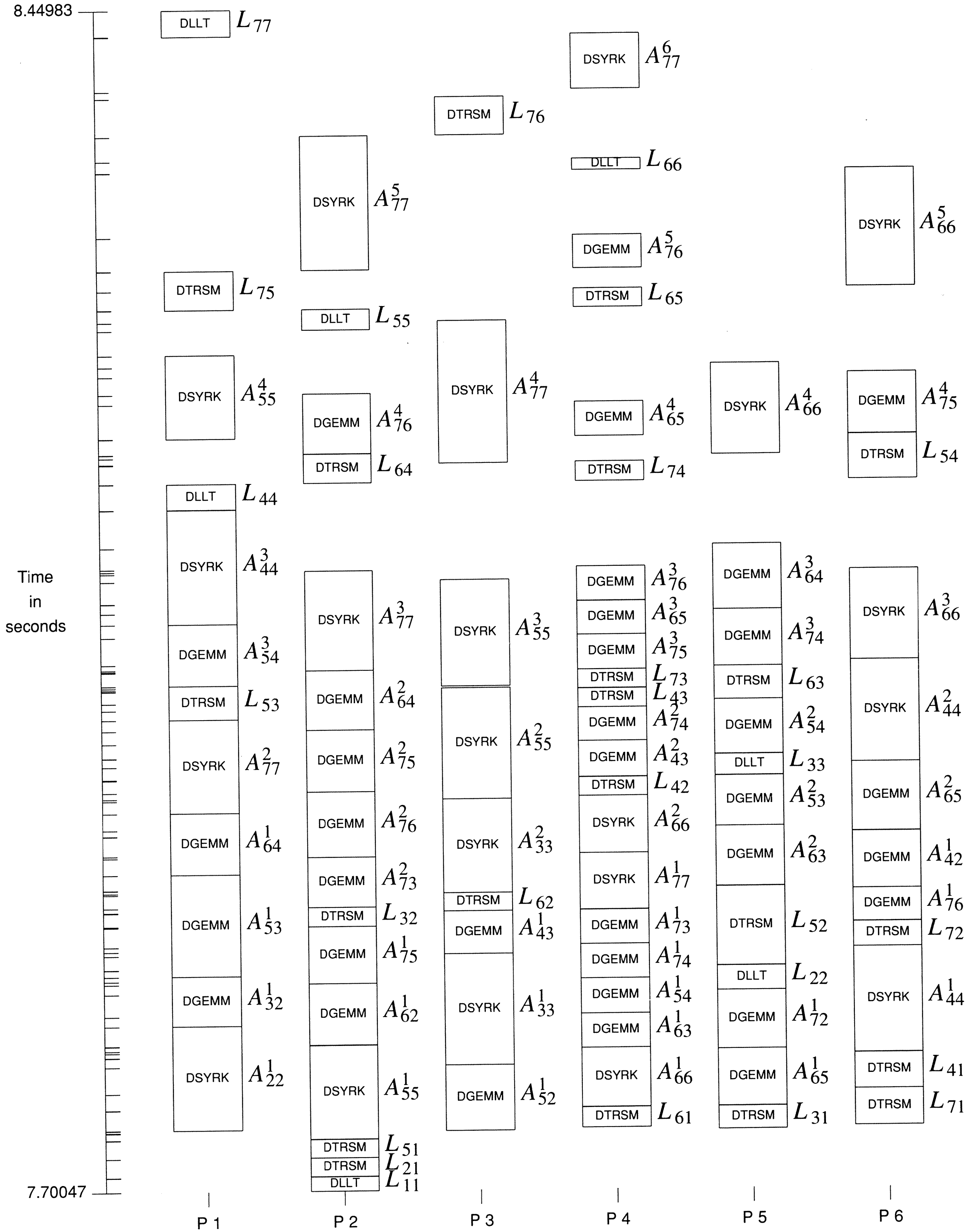

FIGURE 5.2.3, Scheduling of the Cholesky factorization by SCHEDULE of a matrix of order 672 partitioned into 7×7 blocks for the IBM 3090/VF on 6 processors.

As mentioned is Section 4.2, the tasks are equally distributed among the available processors. From Figure 5.2.2, it is not clear which tasks are waiting for each other. Let us rearrange the allocation of tasks. L_{43}, A_{44}^3 and L_{44} can be executed on the same processor, for instance on processor P_1. Concurrently, A_{53}^2, L_{53} and A_{55}^3 can be run on processor P_3. From the data dependency graph of Figure 3.1 we know that A_{54}^3, L_{54}, A_{55}^4 and L_{55} can not be executed in parallel. These jobs can run on P_4. In that case, we obtain a picture similar to Figure 3.2 with concatenated tasks. By this rearrangement of tasks no holes are saved, because all of them arise from data dependency. It can easily be concluded now, that the sooner A_{53}^2 starts the sooner the whole computation ends. By giving no priorities to jobs it may happen that the computation of A_{54}^2 starts before A_{53}^2 only because of its JOBTAG value.

By moving operations to other processors the execution time will not decrease, unless we can reuse data from cache. The allocation of tasks as suggested above provides that succeeding tasks operate on the same block or at least needs data calculated in its preceding task. Another possibility to force reuse of cache is to concatenate tasks. For the computation of L_{lj} $(l = j, \cdots, K)$ from the original block A_{lj} j steps are required (see formulas (2.6a-d)). Assume that these steps are performed in one *supertask* S_{lj} then the execution of such a *supertask* can not begin before $L_{j,1:j}$ and $L_{l,1:j-1}$ have been finished. Moreover, the execution time of the *supertasks* increases with the value of j. Precisely the most expensive tasks like $S_{K-1,K-1}$, $S_{K,K-1}$ and S_{KK} are strongly data dependent and cannot run with any other job concurrently. Summarizing, the application of supertasks will decline the degree of parallelism considerably, and it is not expected that the reuse of data will compensate this loss.

Figure 5.2.3 illustrates the scheduling on the IBM 3090/VF on 6 processors with $k = 7$. The variation in execution time for the BLAS operations is more obvious. It appears, that the rank-k update is the most expensive operation, since an optimal vendor-supported implementation of routine DSYRK is not available. Again the scheduling of the Cholesky factorization on the diagonal blocks turns out to be very important. During the computation of L_{44} and L_{55}, the other processors are idle, which means that no executable jobs are available. In Section 3, we proposed a scheduling in a prescribed order, for instance columnwise. Periods of idle waiting were saved by performing an arbitrary executable job. Apparently, SCHEDULE performs all executable jobs in an "arbitrary" order instead of scheduling in a prescribed order.

Furthermore, we remark that not until 3 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}, $i = 2, \cdots, 7$ become executable simultaneously. We do not precisely understand why this only happens at the start of the execution. Other holes, which occur during the course, arise from empty READYQ's.

Note, that the values at the top right of each node in Figure 3.1 are no longer valid. From Figure 5.2.1 and 5.2.2, it follows that a good estimate can not be made, since the length of the blocks vary too much. Finally we remark that the trace facility was only used to analyze the course of execution and it was not used in timing programs.

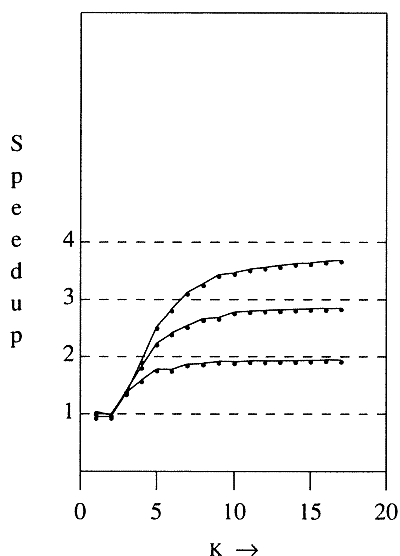

FIGURE 5.3.1a, Alliant FX/4, NB=64
Speedup for 2 - 4 processors

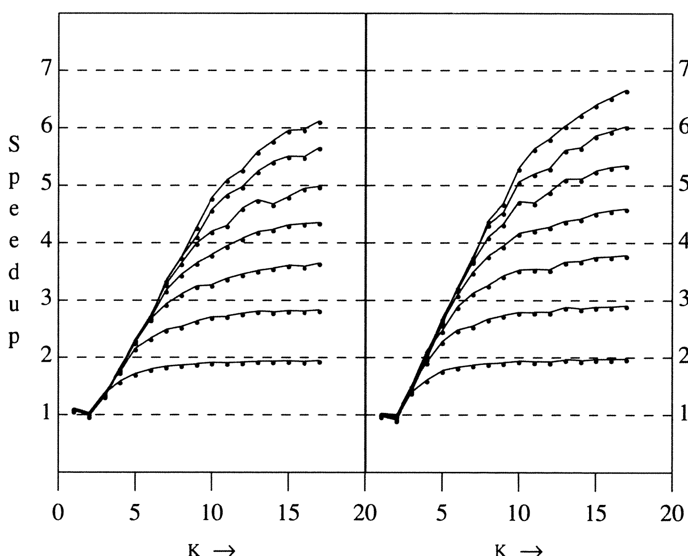

FIGURE 5.3.1b-c, Alliant FX/8, NB=32(left), NB=96(right)
Speedup for 2 - 8 processors

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[11], 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. In the newest version of SCHEDULE the cumulative number of jobs can be many times more than 1000. However, we only apply *static* allocation and we think that our application requires the number of cumulative jobs to be equal to the number of active jobs. 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 5.3.1a 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 Figures 5.3.1b-c, the speedup for the 8 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. The speedup obtained on 8 processors is more than 6 and is expected to come close to 7 for K > 17. These pictures illustrate that the number of blocks must be large enough to keep all processors busy. Recall that the overhead of scheduling is minimal for 1 processor, since exactly one queue of *ready to activate tasks* is provided, and processes never become data dependent.

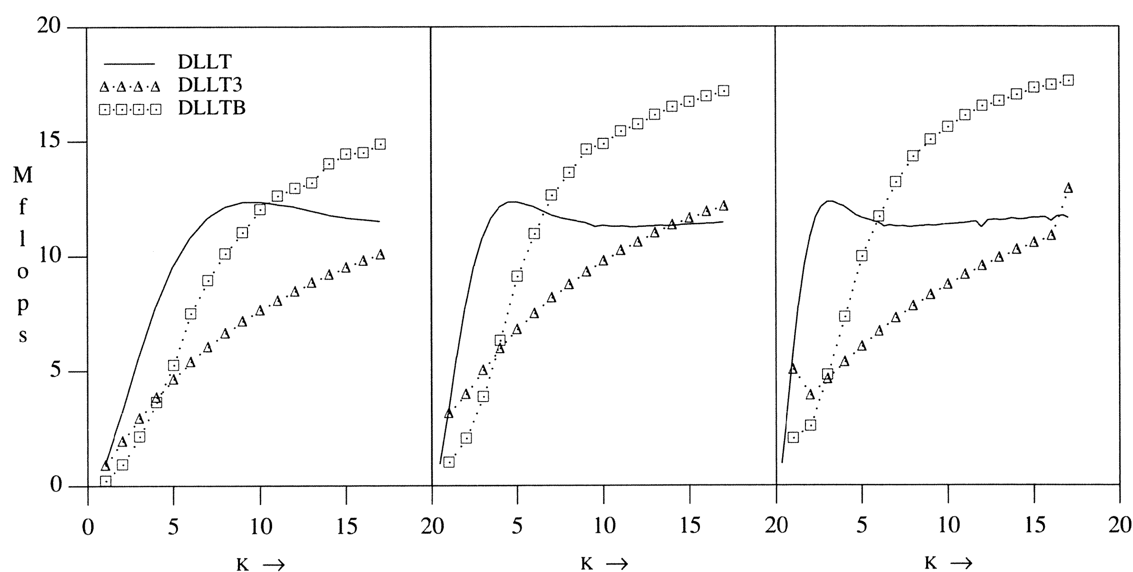

FIGURE 5.3.2a-c, Alliant FX/4, $p = 4$, NB=32(left), NB=64(centre), and NB=96(right)
Performance of DLLT, DLLT3, DLLTB

The blocksizes used in our experiments are 32,40, ··· ,96. Results are given for NB = 32, 64, 96; for other blocksizes similar pictures can be shown. In Figure 5.3.2a-c 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 matrix order. This declares why its shape differs in each picture. The maximum performance of DLLT is reached for $n = 256$. For $n > 256$ its performance decreases due to cache effects. The block algorithms DLLT3 and DLLTB are less sensitive to cache effects. 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.

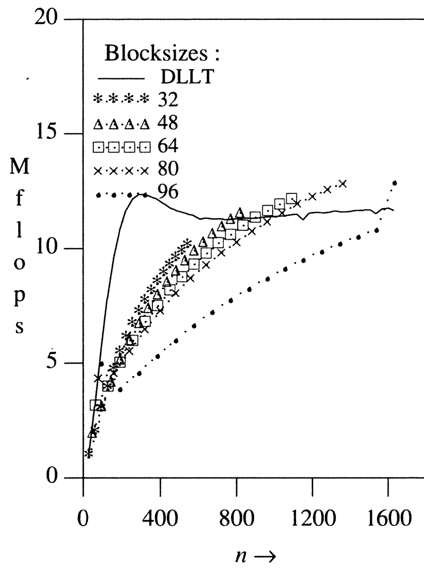

FIGURE 5.3.3a, Alliant FX/4, p=4
Performance of DLLT and DLLT3
NB=32,48,64,80,96

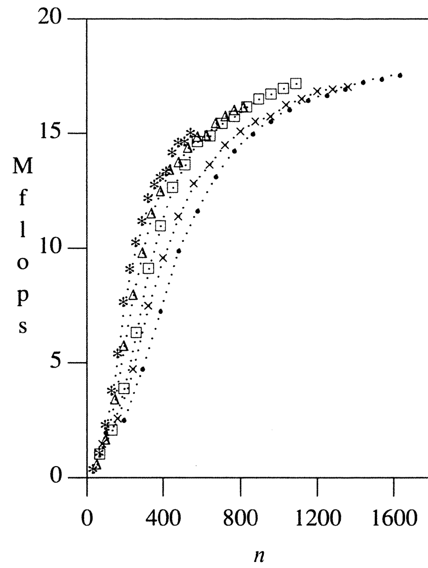

FIGURE 5.3.3b, Alliant FX/4, p=4
Performance of DLLTB
NB=32,48,64,80,96

The way of presentation in Figures 5.3.2a-c does not indicate how a matrix has to be partitioned to obtain optimal performance. For that purpose we present the results of DLLT3 and DLLTB, separately. In Figure 5.3.3a, 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]). A good choice for the blocksize will be 32. 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 5.3.3b gives the results for DLLTB for the Alliant FX/4. Again a blocksize of 32 gives rise to the highest efficiency. However, in this case, the degree of parallelism must be considered, as well. A small blocksize is associated to a large value of K. The question arises whether the speed is completely determined by K, as in the theoretical case, or by the blocksize NB, as well. In Figure 5.3.3b, the lines associated to NB = 48 and NB = 64 appeared to coincide for large values of n(n>800). This indicates that the efficiency does not only depend on K, but on the blocksize, as well. To make this more plausible, we present the performance of DLLTB as obtained on 1 and 2 processors. Figure 5.3.4b illustrates that on 2 processors a blocksize of 64 results in higher performances than a blocksize of 48, despite its smaller K value.

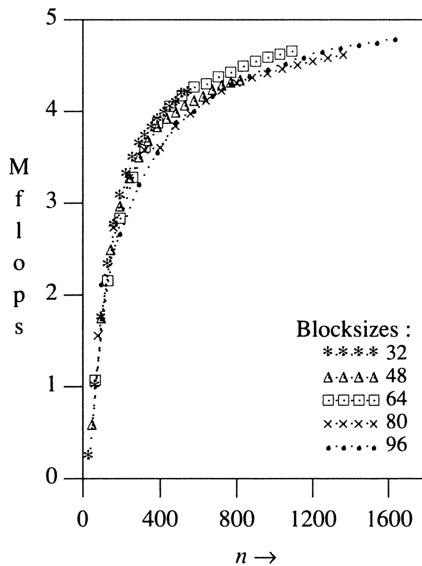

FIGURE 5.3.4a, Alliant FX/4, p=1
Performance of DLLTB
NB=32,48,64,80,96

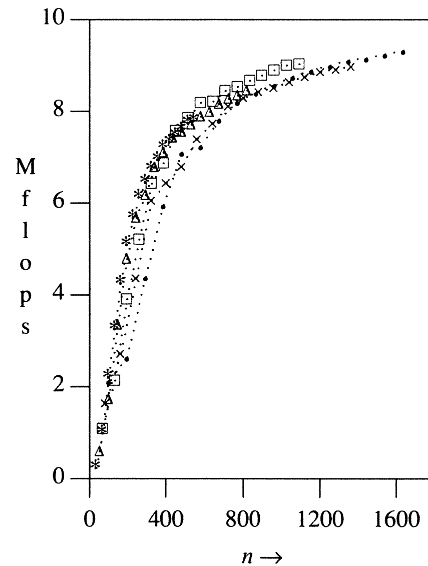

FIGURE 5.3.4b, Alliant FX/4, p=2
Performance of DLLTB
NB=32,48,64,80,96

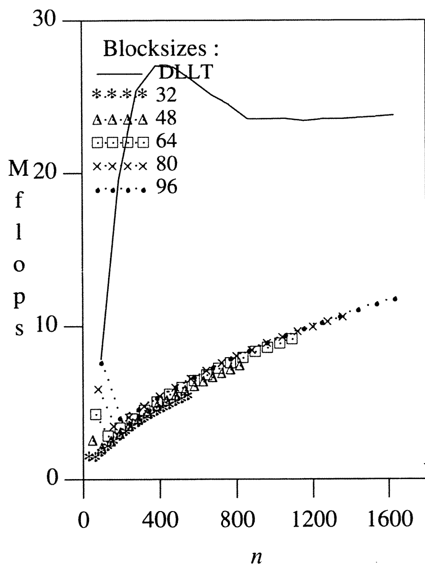

FIGURE 5.3.5a, Alliant FX/8, $p=8$
Performance of DLLT and DLLT3
NB=32,48,64,80,96

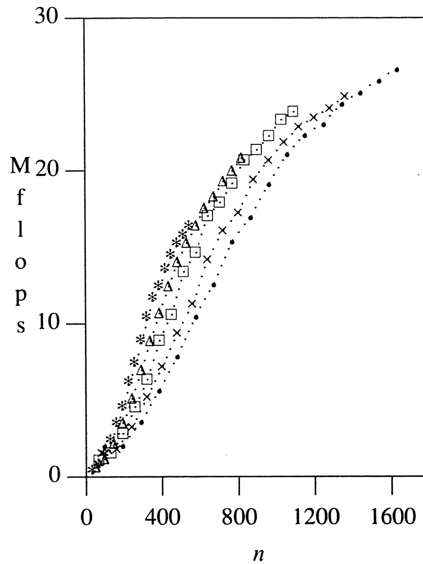

FIGURE 5.3.5b, Alliant FX/8, $p=8$
Performance of DLLTB
NB=32,48,64,80,96

Comparing the results for DLLT for the Alliant FX/4 (Figure 5.3.3a) and the Alliant FX/8 (Figure 5.3.5a) we see that the performance has been doubled, as might be expected, on twice the number of processors. The speed of DLLT3 on the Alliant FX/8 is very disappointing, probably due to the low performance of the BLAS routine DTRSM (cf. Figure 5.1.1b). Figure 5.3.5b presents the performance of DLLTB on the Alliant FX/8 based on exactly the same BLAS implementation as used for DLLT3. The large difference in performance between DLLT3 and DLLTB is also caused by the different data structure.

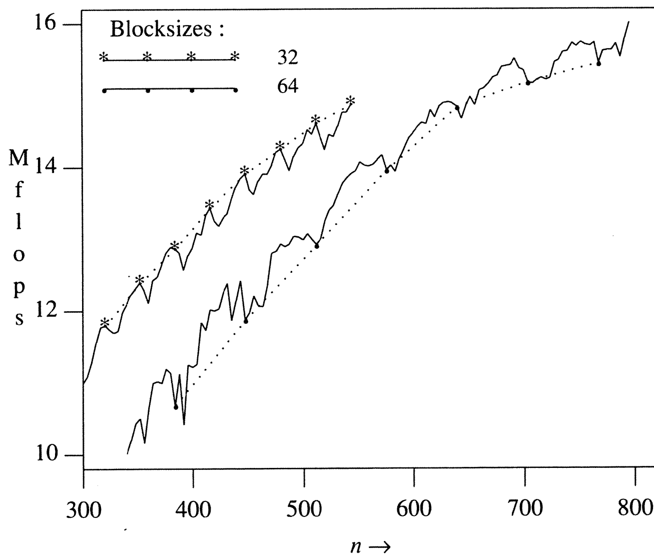

FIGURE 5.3.6, Alliant FX/4, NB=32, 64
Performance of DLLTB with unequally sized blocks

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 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 5.3.6 displays the performance for matrices of order $n = 300, 304, \cdots, 800$ and for NB=32 and NB=64. The dotted lines connect the points with $n = K \times NB$. 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. For that reason we do not compare the efficiency of parallelized DLLTB with the efficiency of the unparallelized DLLT and DLLT3.

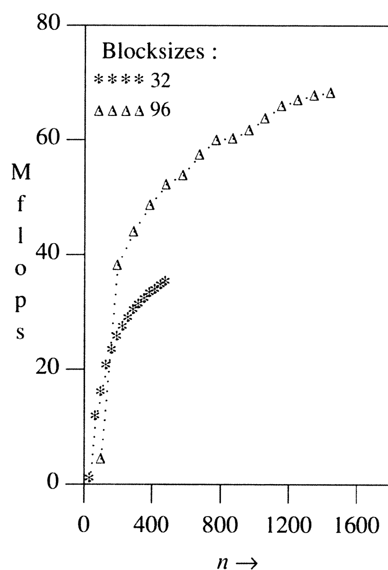

FIGURE 5.4.1a, IBM 3090/VF, $p=1$
Performance of DLLTB, NB=32,96

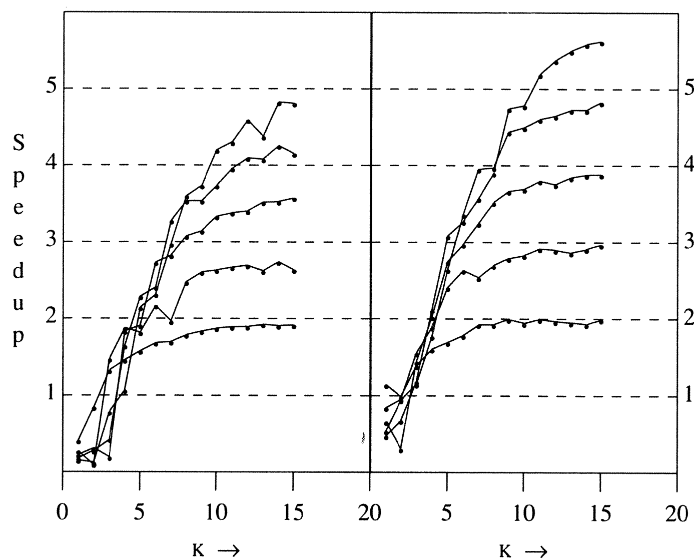

FIGURE 5.4.1b-c, IBM 3090/VF, NB=32(left), NB=96(right)
Speedup for 2 - 6 processors

In Figure 5.4.1b-c the speedup for the IBM 3090/VF is given. For small values of K the speedup is less then 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 Figures 5.3.1a-c the speedup increases with the blocksize. For the behaviour of DLLTB on a single processor we refer to Figure 5.4.1a. For the IBM an older version of SCHEDULE[7] is available. It turns out that only the results for $K \leq 15$ are correct. We do not go into further detail in this paper.

Finally, in Figures 5.4.2a-b we give the results of DLLTB on 4 and 6 processors, respectively. We observe that the performance strongly depends on the blocksize, whereas for the Alliant FX/4 (cf. Figure 5.3.3b) and the Alliant FX/8 (cf. Figure 5.3.5b) the value of K mainly influences the speed. Especially the timings in Figure 5.4.2a point to a slow communication between the processors in proportion to the IBM BLAS performance. A blocksize of 64 will be a good choice for the IBM 3090/VF.

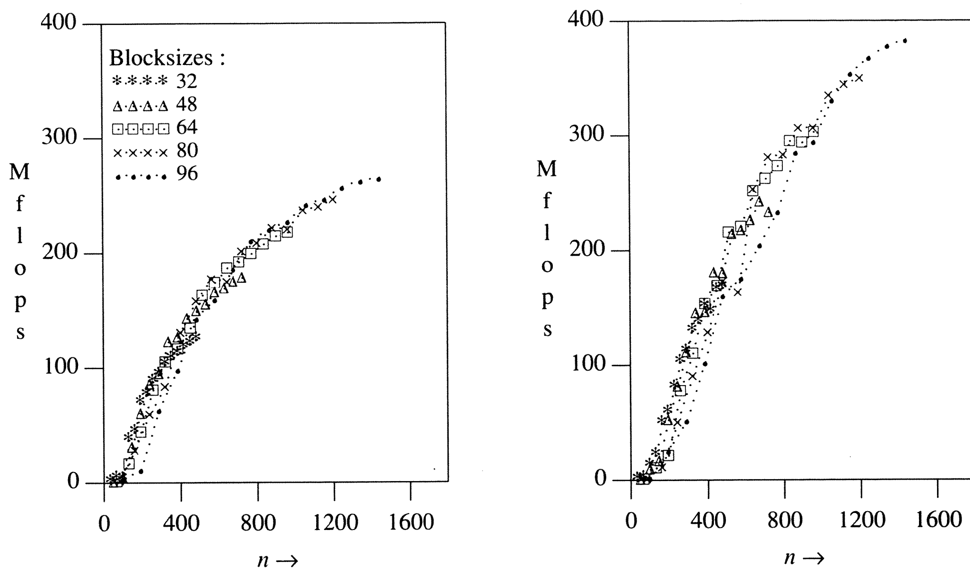

FIGURE 5.4.2a-b, Performance of DLLTB on p=4(left) and p=6(right) processors
IBM 3090/VF, NB=32,48,64,80,96

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 usage of the SCHEDULE package helps to introduce parallelism in a transportable way. For the machines on which we run 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. Not only the CPU time for DLLTB was less than for DLLT (Level 2 BLAS) and DLLT3 (Level 3 BLAS), but also the wall clock time was much shorter, caused by the different data access pattern. We remark that at most $3p$ submatrices, explicitly stored blockwise, are needed at a time.

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 non-parallel 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 non-parallel BLAS version is available. Therefore we suggest to distinguish between parallel and non-parallel BLAS implementations. We believe that both versions should be accessible for exploiting parallelism over and within the BLAS kernels.

ACKNOWLEDGEMENTS

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 (May 1990). *LAPACK: A Portable Linear Algebra Library for High-Performance Computers*, University of Tennessee, CS-90-105.

2. M.J. DAYDE and I.S. DUFF (1990). 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.
3. J.J. DONGARRA, J. DU CROZ, I. DUFF and S. HAMMERLING (March 1990). A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1), pp. 1-17.
4. J.J. DONGARRA, J. DU CROZ, S. HAMMERLING, and R.J. HANSON (1988a). An extended set of Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1), pp. 1-17;18-32.
5. J.J. DONGARRA, I.S. DUFF, D.C. SORENSEN and H.A. VAN DER VORST *Linear Systems Solving on Vector and Shared Memory Computers* (to appear).
6. J.J. DONGARRA, F.G. GUSTAVSON and A. KARP (1984). Implementing linear algebra algorithms for dense matrices on a vector pipeline machine, *SIAM Rev.* 26, pp. 91-112.
7. J.J. DONGARRA and D.C. SORENSEN (November, 1986). Schedule: Tools for developing and Analyzing Parallel Fortran Programs, Argonne National Laboratory Report, ANL-MCS-TM-86.
8. K. GALLIVAN, W. JALBY and U. MEIER (1987). The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory. Timely communications, *SIAM J. Sci. Statist. Comput., 8, pp. 1079-1084.*
9. A. GEORGE, M.T. HEATH and J. LIU (1986). Parallel Cholesky Factorization on a Shared-Memory Multiprocessor, *Linear Algebra and its Applications*, 77, pp. 165-187.
10. G.H. GOLUB and C.F. VAN LOAN (1989). *Matrix Computations.* The Johns Hopkins Press, Baltimore, Maryland, 2nd edition.
11. F.B. HANSON and D.C. SORENSEN (January, 1989). The SCHEDULE Parallel Programming Package with Recycling Job Queues and Iterated Dependency Graphs. Argonne National Laboratory Report, ANL-MCS-P22-0189.
12. M. LOUTER-NOOL and D.T. WINTER (1989). Benchmark of the initial release of the LAPACK library, Note NM-N8903 + supplement, CWI, Amsterdam.
13. E. LUQUE, A. RIPOLL, P. HERNANDEZ and TOMAS MARGALEF (1990). 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.
14. J.M. ORTEGA (1988). The ijk forms of factorization methods I. Vector computers, *Parallel Computing*, 7, pp. 135-147.
15. J.M. ORTEGA and C.H. ROMINE (1988). The ijk forms of factorization methods II. Parallel Systems, *Parallel Computing*, 7, pp. 149-162.