Explicit parallel block Cholesky algorithms on the CRAY APP

M. Nool

Department of Numerical Mathematics

Report NM-R9425 December 1994

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Explicit Parallel Block Cholesky Algorithms on the CRAY APP

Margreet Nool <greta@cwi.nl>

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

In this paper we consider the CRAY APP, the Attached Parallel Processor of the CRAY S-MP, which consists of seven buses with each bus supporting up to 12 processing elements. Processing elements on different buses can communicate simultaneously with the shared main memory, but processing elements sharing the same bus can not, since only one processing element per bus can access memory at a given time. Applications with a high level of data reuse, or, with a high compute intensity, and applications being highly parallel are very suitable to run on the APP. An example of such an algorithm is matrix-matrix multiplication. We illustrate how the data traffic's restriction influences the performance and we discuss the scalability of the CRAY APP.

Furthermore, two different algorithms for Cholesky factorization are discussed: a block left-looking algorithm and a block right-looking algorithm. The maximum achievable speed on the CRAY APP is mainly determined by the performance of the matrix-matrix multiplication. Parallelism is applied explicitly over the blocks, which makes it possible to concatenate different block operations in cache. The results obtained on CWI's APP (a machine having twenty-eight processing elements) indicate how block algorithms can be parallelized on machines with hundreds or thousands of processors.

AMS Subject Classification (1991): Primary: 65-04. Secondary: 65M20, 65M55, 65Y99.
CR Subject Classification (1991): G.1.8
Keywords & Phrases: software, parallelization, vectorization.

1. INTRODUCTION

The Attached Parallel Processor APP of the CRAY S-MP is a system of one up to seven buses, each consisting of several processing elements. In the case of one processing element per bus, it is possible to obtain a speed-up which is close to the number of buses involved. However, in the case of more than one processing element sharing the same bus, the speed-up is restricted by bus traffic, since only one processing element per bus can access main memory at a given time. For that reason, only parallel algorithms with a high computation intensity compared to data traffic are suitable to run efficiently on the APP. In earlier papers[11, 8, 7], which report on APP performance, the speed-up was mainly limited by the number of buses rather than by the number of processing elements. Here, we concentrate on such parallel applications, for which we actually obtain speed-ups higher than the number of buses. For that purpose we introduce the bus speed-up being a function of the execution time on B buses with one processor and that on B buses with P processors. Moreover, we consider the scalability of the APP, and we discuss the influence on the performance in case the speed of the processing elements or the speed of data transfer increases. Finally, we consider the bus performance as a function of the cache contents.

The matrix-matrix multiplication serves as a key in the first part of the paper dealing with the APP and its suitability for this operation. The Cholesky factorization, considered in the second half, is based on the matrix-matrix multiplication as well. Its speed is mainly determined by the multiplication's speed. The factorization can easily be parallelized, but during the progress of the factorization the level of parallelism decreases. We have found that it is difficult to predict how a given matrix should be divided into blocks to achieve optimal performance. Two Cholesky factorizations are compared: a

Report NM-R9425
ISSN 0169-0388
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

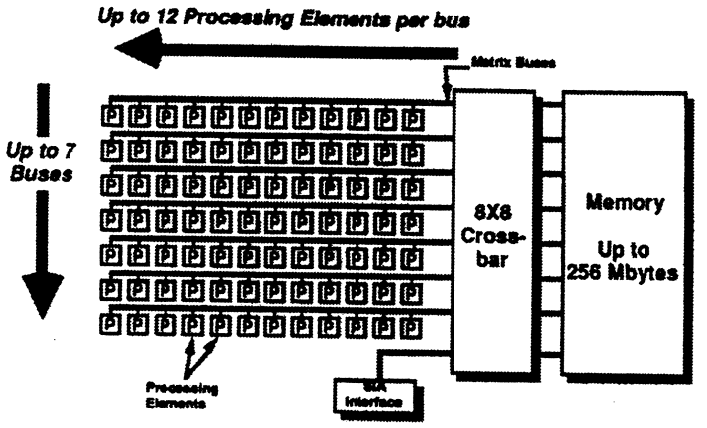

Figure 1: CRAY APP Architecture

left- and a right-looking algorithm. Though solving the same decomposition within the same number of steps, their performance differs significantly on a 28 processor CRAY APP.

The organization of the paper is as follows. In Section 2, we describe the CRAY APP configuration. In Section 3, we focus on the block matrix-matrix multiplication; it is performed in such a way that data traffic is minimal resulting in a high performance. The scalability of the CRAY APP is discussed in Section 4. In Section 5, two block Cholesky factorizations are described; much attention is paid to reduce the number of synchronization points leading to an increase of the performance. Finally, in Section 6, some conclusions are drawn.

2. The CRAY APP configuration

The CRAY APP, a multiple-bus, parallel processor, can be used in two modes: the client server programming model and the computer model. In the first model, the main program runs on the front end machine and highly parallel time consuming parts of the program are run on the CRAY APP. In the latter model, which we consider in this paper, *all* operations are performed on the CRAY APP.

Its architecture, as illustrated in Fig. 1, consists of 4 to 84 processing elements and one to seven buses. The buses are connected to the main memory by a crossbar. This main memory is shared by all of the processing elements. An important restriction in bus traffic is that at a given time only one processing element on each bus can access memory. As a consequence, for optimal use of the CRAY APP it is necessary to take care of a balanced bus traffic. An example of efficient bus usage is shown in Fig. 2. Due to the short load time compared to the computation time in cache, seven processing elements can be used efficiently. The addition of one or more processing elements to the bus will not result in a higher performance; several elements must wait to use the bus for loading. Obviously, the desired number of processing elements on a bus depends on the application.

There are two ways in which parallelism can be exploited on the CRAY APP:

- by using the auto-parallelizing feature of the compiler and insert directives for parallel processing wherever possible;
- by "bus-handoff" computing. Hardware features are available for loading data into caches and for storing updated values into the main memory.

In this paper, we focus on the optimization of bus traffic and the reuse of data wherever possible. To this aim, we will investigate on the application of "bus-handoff" computing.

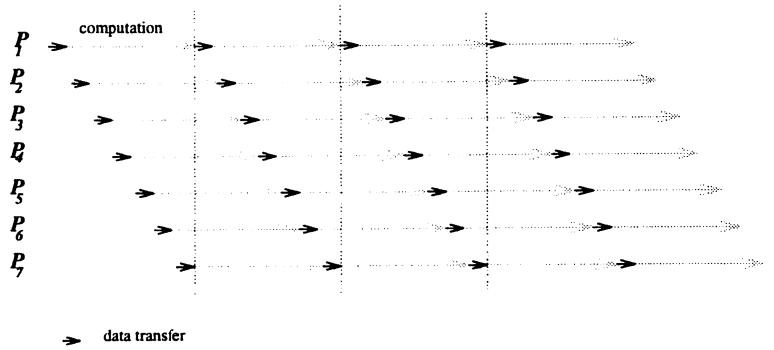

Figure 2: Efficient Bus Usage of 7 processors sharing the same bus

2.1 Some characteristics of the CRAY APP

The CRAY APP at CWI, Amsterdam consists of seven buses, each with four processing elements. The data cache of each processing element is a 8 Kbyte memory chip so that for `DOUBLE PRECISION` arithmetic only 1Kwords can be stored. This implies that for a matrix-matrix multiplication in cache involving two operand matrices and one result matrix, the block size for square matrices is restricted to 18. The processing elements, based on the Intel I860 64-bit microprocessor, can reach a peak performance of 60 Mflops for `DOUBLE PRECISION` operations. However, when performing an equal number of `DOUBLE PRECISION` multiples and adds, as in the matrix-matrix multiplication case, the maximum possible performance is 40 Mflops (not including data transfer). In addition, each bus provides a peak bandwidth of 160 Mbytes/second, i.e., 20 Mwords `DOUBLE PRECISION` words can be transferred per second. So, two flops and one data transfer can be done simultaneously.

2.2 Some remarks on the software available on the CRAY APP

From the previous section it seems easy to predict the performance of a parallel application, since everything needed for a good and reliable analysis is available, such as, the data cache size, a performance peak of 40 Mflops as well as the data transfer time. Moreover, the CRAY APP has a very accurate clock, and timing results for a parallel application on the APP appear to be reproducible. On the CWI's configuration a peak Mflop-rate of more than 1 Gflops should be possible. However, only a reasonable performance can be obtained by using *cache* programming and for this it is necessary to use the Extended Math Library routines [1]. Unfortunately, this has only a small set of appropriate routines for numerical programming. It would be helpful if at least a set of Level 1 BLAS [9] routines were included in this library, but even common-used operations like inner products and **daxpy** operations are missing. In fact, only a limited set is available, and its usability for a simple user is restricted by incomprehensible rules.

Consider, for instance, the routine `_dvsvma` , which performs the following operation on `DOUBLE PRECISION` data:

$$\mathbf{D}(i) = \mathbf{A}(i) + \mathbf{B} * \mathbf{C}(i); \quad for\ i = 1, \cdots, n. \tag{2.1}$$

The first restriction is that the vectors **A** and **D** may not share the same memory locations as they do in a **daxpy** operation. Therefore, this routine cannot be used as a simple **daxpy** operation without an extra vector copy neither can any other Extended Math Library routine. Secondly, the elements of **A**, **C** and **D** must be consecutive elements in memory; an increment parameter is not allowed. The third restriction gives rise to most complications: **A**, **C** and **D** must be **quad-word aligned**. Operations can only be carried out in pipelined mode if the initial elements of the input data are stored on **quad-word boundaries** in cache. A word in the w-th position in cache is quad

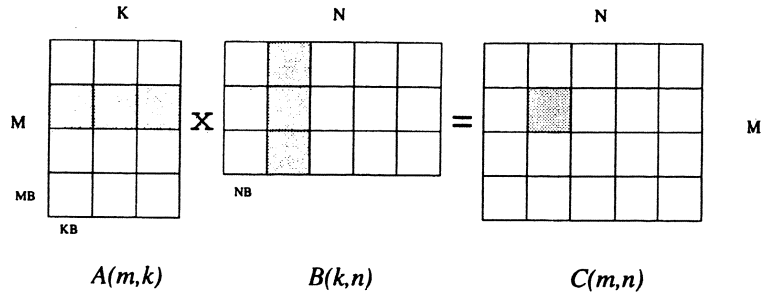

Figure 3: Matrix-matrix multiplication $A \times B = C$; the matrices A, B and C are partitioned into blocks of order MB × KB, KB × NB and MB × NB, respectively.

word aligned, or on a quad word boundary if w mod 4 = 1, as is described in [11]. For the DOUBLE PRECISION case either the odd or the even vector elements can be quad-word aligned; for SINGLE PRECISION case only each fourth element of a vector satisfies this restriction.

Finally, we mention the DOUBLE PRECISION matrix-matrix multiplication routine _dmmm which has a speed of more than 25 Mflops on a single processor. This routine computes in cache

$$\mathbf{C} = r * \mathbf{C} + s * \mathbf{A} * \mathbf{B},$$

where r and s may be $-1, 0$, or 1. Besides the restriction that the memory occupied by $\mathbf{C}$ should not intersect that occupied by $\mathbf{A}$ or $\mathbf{B}$, we must take into account that this routine is *parameterized*: all cache loads and stores are internal to the routine and all preloaded data in the cache will be lost. As a consequence, it does not seem to be possible to reuse the data, since each time the routine is called the cache will be refreshed. However, in section 3.4 we show results of experiments where in certain cases data are still reused by this routine.

The absence of a good compiler for *cache* programming and the large number of restrictions makes programming very complicated and reduces the readability of programs enormously. Most executions do not achieve the performances claimed in the specifications. In section 5.1.2, we return to this point.

3. The matrix-matrix multiplication

The matrix-matrix multiplication can be considered as one of the most suitable applications to run on the CRAY APP. Firstly, the multiplication is highly parallel; it can easily be divided into (many) parts which can run in parallel. Secondly, the (compute) intensity, defined by the ratio of the number of floating point operations to the number of words of data, is high. For a real matrix-matrix multiplication of the form $\mathbf{C} = \mathbf{C} + \mathbf{A} * \mathbf{B}$, the intensity is given by

$$\texttt{Intensity} = \frac{\#\texttt{Flops}}{\#\texttt{Data Words Transferred}} = \frac{2n^3}{4n^2} = \frac{1}{2}n. \tag{3.1}$$

3.1 Parallelism

Let the real matrices A, B and C of Fig. 3 have orders $m \times k$, $k \times n$, $m \times n$ and assume that KB, MB and NB are proper divisors of k, m and n, respectively, such that K × KB = k, M × MB = m and N × NB = n. Then, A can be partitioned into a block matrix of M × K blocks of order MB × KB. Matrices B and C can be partitioned analogously. An example of a block algorithm to perform the matrix-matrix multiplication $C = C + A \times B$ is given by Fig. 4. The inner loop is not particularly suitable for parallel processing, since for each l the same submatrix $C_{i,j}$ is updated. Introducing parallelism on the level of the j loop results in updating the N block columns of C in parallel. The

```
DO i=1,M
    DO j=1,N
    C_{i,j} = 0.0d0
        DO l=1,K
            C_{i,j} = C_{i,j} + A_{i,l}B_{l,j}
        END DO
    END DO
END DO
```

Figure 4: Block Matrix-matrix algorithm.
Each block $A_{i,l}$, $B_{l,j}$ and $C_{i,j}$ is of order MB × KB, KB × NB and MB × NB, respectively.

introduction of parallelism on the outer loop level leads to concurrent computation of the block rows of C and a level of parallelism equal to M. Note, that if loops i and j are interchanged parallelism over the block columns changes to parallelism over the block rows. However, there will be a significant difference in how operations are scheduled to the processing elements. The order of the operations will differ as well as the *idle* pattern if M or N are not a multiple of the number of processing elements. In [8], a block row and a block column scheme are considered and experiments on the CRAY APP display that the idle time becomes significant due to a lack of parallelism.

A higher level of parallelism and a possible reduction of idle time can be achieved by collapsing the middle and the outer loop. As a consequence, all computations on submatrices $C_{i,j}, i = 1, \cdots, \mathrm{M}, j = 1, \cdots, \mathrm{N}$ can be carried out independently, resulting in a level of parallelism of $\mathrm{M} \cdot \mathrm{N}$. Since each submatrix $C_{i,j}$ is stored once, the number of stores is minimal. On the other hand, the number of loads is high, since all submatrices of A and B are loaded K times.

3.2 Granularity

Assume that parallelism is exploited on submatrix level rather than block row or block column level. One way to enlarge the degree of parallelism is to change the block size. A reduction of the block size leads to a higher level of parallelism, but not automatically to a higher performance on a parallel machine. A reason for this is that the intensity per block operation – which is of order NB for square blocks – decreases, whereas M and N increase.

Let the cycle time for one floating point operation be τ_f, then for each block $C_{i,j}$ the computation time T_{c_block} will be

$$T_{c\_block}(k, \mathrm{MB}, \mathrm{NB}) = 2 \cdot \mathrm{K} \cdot \mathrm{KB} \cdot \mathrm{MB} \cdot \mathrm{NB} \cdot \tau_f = 2k \cdot \mathrm{MB} \cdot \mathrm{NB} \cdot \tau_f.$$

Define τ_t being the time to transfer one datum word from main memory to cache or vice versa. To compute one block $C_{i,j}$ requires $\mathrm{K} \cdot \mathrm{KB} \cdot \mathrm{MB} + \mathrm{K} \cdot \mathrm{KB} \cdot \mathrm{NB} + \mathrm{MB} \cdot \mathrm{NB}$ loads and $\mathrm{MB} \cdot \mathrm{NB}$ stores. So, the transfer time T_{t_block} will be

$$T_{t\_block}(k, \mathrm{MB}, \mathrm{NB}) = [k(\mathrm{MB} + \mathrm{NB}) + 2 \cdot \mathrm{MB} \cdot \mathrm{NB}]\tau_t.$$

The total time needed for computing one block, including loads and stores, T_{block} is given by

$$T_{block}(k, \mathrm{MB}, \mathrm{NB}) = T_{c\_block} + T_{t\_block} = 2k \cdot \mathrm{MB} \cdot \mathrm{NB} \cdot \tau_f + [k(\mathrm{MB} + \mathrm{NB}) + 2 \cdot \mathrm{MB} \cdot \mathrm{NB}]\tau_t.$$

The APP's uniprocessor time needed to carry out the matrix-matrix multiplication $C = A \times B$ will be

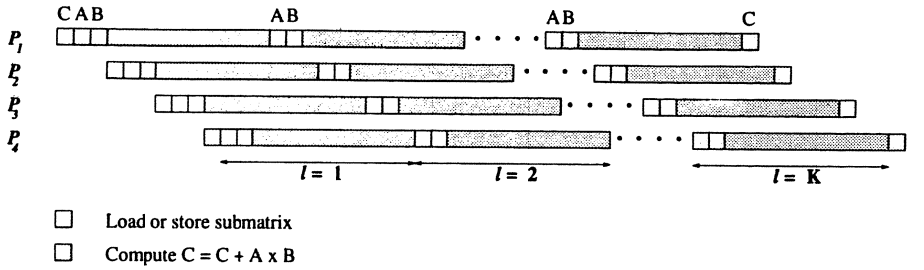

Figure 5: The load/store and computation process
for the matrix-matrix computation $C = C + A \times B$ on 4 processors sharing the same bus.

$$T_{matrix} = \mathrm{M} \cdot \mathrm{N} \cdot T_{block}(k, \mathrm{NB}, \mathrm{MB}) = 2kmn\tau_f + k(m \cdot \mathrm{N} + \mathrm{M} \cdot n)\tau_t + 2mn\tau_t. \tag{3.2}$$

A change of block size effects (i) the time required to load the matrices A and B and (ii) the maximum degree of parallelism. For the given algorithm, halving the block size causes a doubling of the load time for A and B and a change of 2^2 in the degree of maximum parallelism. From (3.2) we may conclude that the uniprocessor time for a matrix-matrix multiplication is minimal for M and N as small as possible. In other words, as the cache contents increases the processing time will decrease. Given the cache size, we can create a matrix-matrix multiplication of maximal performance. On the other hand, (3.2) does not indicate for a fixed m and n, and a given number of processors, for which block size the lowest wall clock time can be reached.

3.3 Bus-configuration

Till now, we have not considered the important restriction of the bus-configuration: processors on the same bus can not communicate with the main memory, concurrently. If the computation time is much larger than the transfer time the consideration will have a very limited effect. If not, processors will become idle because the required data are not available. For simplicity, we assume that there are p processors $p_1, p_2, \cdots, p_p$ on one bus and that to each processor exactly one block $C_{i,j}$ is assigned. The process of computation and load/store traffic on a processor is shown in Fig 5.

The process on processor p_1 computing C_{i_1,j_1} can be described as follows:

- load block matrix C_{i_1,j_1},
- load block matrices $A_{i_1,1}$ and B_{1,j_1},
- compute $C_{i_1,j_1} = C_{i_1,j_1} + A_{i_1,1} \times B_{1,j_1}$.

In the next step, data of C_{i_1,j_1} can remain in cache, and therefore, only $A_{i_1,2}$ and B_{2,j_1} must be loaded. After K steps the process ends by storing the final C_{i_1,j_1} data. The computation of submatrix C_{i_2,j_2} on processor p_2 requires three blocks to be loaded, too. The best solution is to load C_{i_1,j_1}, $A_{i_1,1}$ and B_{1,j_1} on p_1 and then, simultaneously, p_1 starts executing $C_{i_1,j_1} = C_{i_1,j_1} + A_{i_1,1} \times B_{1,j_1}$ and p_2 is loaded with data of C_{i_2,j_2}, $A_{i_2,1}$ and B_{1,j_2}. Processor p_2 starts updating C_{i_2,j_2} as soon as all data required are available in cache. The overhead of computing C_{i_1,j_1} and C_{i_2,j_2} simultaneously, on two processors sharing the same bus, is equal to the time required for loading $A_{i_1,1}$, B_{1,j_1} and C_{i_1,j_1} under the restriction that the time needed for loading is less than the computation time for $C_{i_1,j_1} = C_{i_1,j_1} + A_{i_1,1} \times B_{1,j_1}$.

We can expand this process to p processors sharing the same bus. The overhead can be expressed by

$$(p-1)\,T_{t\_block}(\mathrm{KB},\mathrm{MB},\mathrm{NB}) = (p-1)\,[\mathrm{KB}\cdot\mathrm{MB}+\mathrm{KB}\cdot\mathrm{NB}+\mathrm{MB}\cdot\mathrm{NB}]\cdot\tau_t. \tag{3.3}$$

If this time exceeds

$$T_{c\_block} = 2\cdot\mathrm{KB}\cdot\mathrm{MB}\cdot\mathrm{NB}\cdot\tau_f, \tag{3.4}$$

then more than one processor will try to communicate with the main memory at the same time and processors will become idle. Note the significance of the ratio $\tau_f : \tau_t$.

The maximum useful number of processors on a bus can be reflected in the *leverage* value, which is the ratio of the compute time to data transfer time. A leverage value l indicates that $l+1$ processors can share the same bus, effectively. In other words, the wall clock time can not be reduced using more than $l+1$ processors. For the matrix-matrix multiplication we obtain

$$\begin{aligned} \mathtt{Leverage} &= \frac{\mathtt{Time\ spent\ computing\ in\ cache}}{\mathtt{Time\ spent\ transferring\ across\ the\ bus}} \\ &= \frac{2\cdot k\cdot\mathrm{NB}\cdot\mathrm{MB}\cdot\tau_f}{[k(\mathrm{MB}+\mathrm{NB})+2\cdot\mathrm{MB}\cdot\mathrm{NB}]\cdot\tau_t}. \end{aligned} \tag{3.5}$$

We remark that for the example shown in Fig. 5, four processors can share the same bus efficiently. The addition of a fifth processing element on the bus, will not improve efficiency. It is worth noting, that when the computation starts all values must be available on the on-chip cache, for instance not only the submatrix elements, but also the block size must be known. Since the CRAY APP can be considered as a shared memory machine, all processors have access to the main memory. In case of values being not available on the on-chip cache, a processor will try to load them from main memory. This action will disturb the bus traffic, and processors can become idle resulting in a longer execution time.

3.4 Numerical experiments with matrix-matrix multiplication

In this section, the effect of data traffic on the performance is illustrated by means of the matrix-matrix multiplication. The multiplication has a high compute intensity and, therefore, it is possible to increase the performance by using several processors on the same bus.

It is not our intention to develop the most efficient matrix-matrix multiplication - there exists already a very efficient implementation, i.e., `rgmmul` appropriate for all kinds of matrix shapes - but in this paper we focus on the performance loss due to bus traffic for some fixed block sizes. A high computation speed is needed in order to demonstrate accurately the influence of data traffic. Therefore, we would like to develop a fast implementation for the matrix-matrix multiplication and to control data traffic explicitly, using APP routines to move data from memory into processor cache and vice versa. It appears that the highest possible performance for a matrix-matrix multiplication in cache can be obtained using the routine `_dmmm`. Unfortunately, all data traffic is governed internally by the routine (see subsection 2.2). We use the routine in order to achieve a high computational efficiency and can only speculate on how the routine manages bus traffic.

To get a clear view on the efficiency we have chosen timing experiments on matrices with sizes depending on the configuration chosen. This enables us to avoid idle time due to a lack of parallelism. For a partitioning of the matrices A, B and C into square blocks of order b we choose

$$m = b \times \#\ \mathtt{procs/bus}, \tag{3.6a}$$

$$n = b \times \#\ \mathtt{buses}, \tag{3.6b}$$

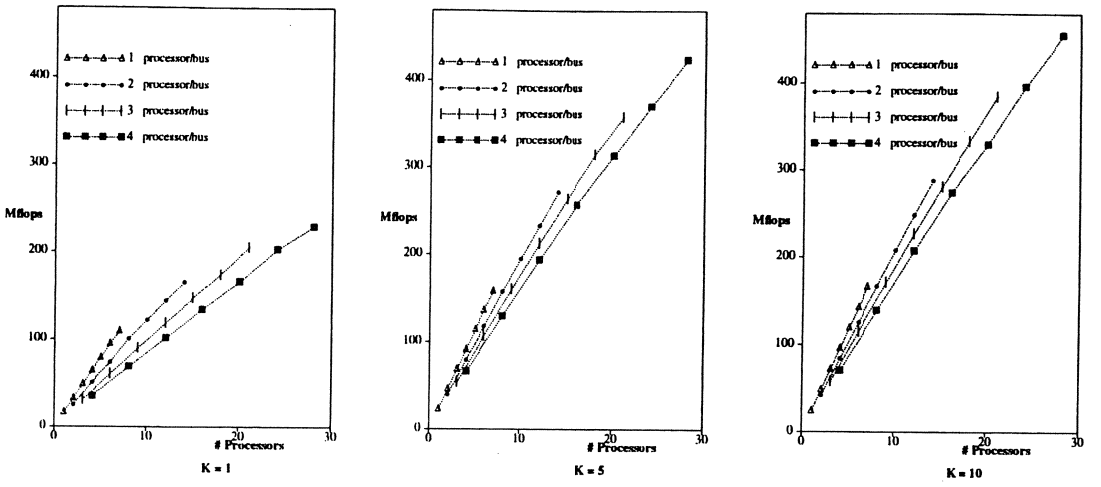

Figure 6: Floating-Point Rate in Mflops achieved for the matrix-matrix product for orders defined by (3.6a-c), where $b = 18$, the maximum attainable block size. K corresponds with the number of block matrix multiplications per resulting block $C_{i,j}$.

$$k = b \times \mathrm{K}, \tag{3.6c}$$

for $\mathrm{K} = 1, \cdots, 10$ (cf. Fig. 3). This implies that for a fixed K the wall clock time for each configuration will be approximately the same. However, the more processing elements involved the higher the Mflop rate will be. Since all cache loads and stores are internal to _dmmm we did not expect any performance improvement by increasing K. Instead of one load and one store for each submatrix $C_{i,j}$ K loads and K stores are required when preloaded data are lost when _dmmm is called. Fig. 6, however, shows that in practice the performance depends on K. Probably, it is recognized during execution, that the elements of $C_{i,j}$ are already in cache.

We have chosen to present the speed-up in two ways:

$$S_{processor} = \frac{\texttt{Execution time on 1 processor}}{\texttt{Execution time on B} \times \texttt{P processors}}, \tag{3.7a}$$

$$S_{bus} = \frac{\texttt{Execution time on B buses with 1 processor}}{\texttt{Execution time on B} \times \texttt{P processors}} \tag{3.7b}$$

For K=10 and block size $b = 18$ a processor speed-up close to 18 is reached for the full configuration. The bus speed-up lines in Fig. 7 are nearly horizontal, which implies that the gain of adding more processing elements to a bus does not depend on the number of buses already involved. In other words, in that direction the configuration is scalable. The effect of adding more processing elements to a bus is less obvious. Fig. 8 clearly displays that an increase in cache contents corresponds with an increase of bus speed-up and thus the performance will grow. In practice, it does not pay to use more than three processing elements per bus in case $b = 10$ for this application.

4. Scalability of the CRAY APP

From the previous section it is known that when the number of buses increases the speed-up will increase linearly. In contrast, adding more processing elements to a bus will not lead to the same speed-up. Only a few applications have such a high compute intensity that they can use the APP-configuration with four processors per bus efficiently. At least three other characteristics of the APP can be distinguished which determine the ultimate performance result:

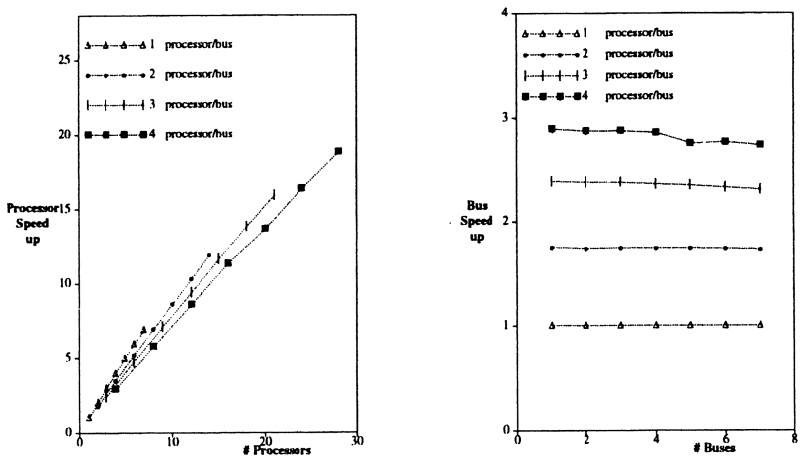

Figure 7: Processor and bus speed-up for the matrix-matrix product with K=10 and a block size $b = 18$

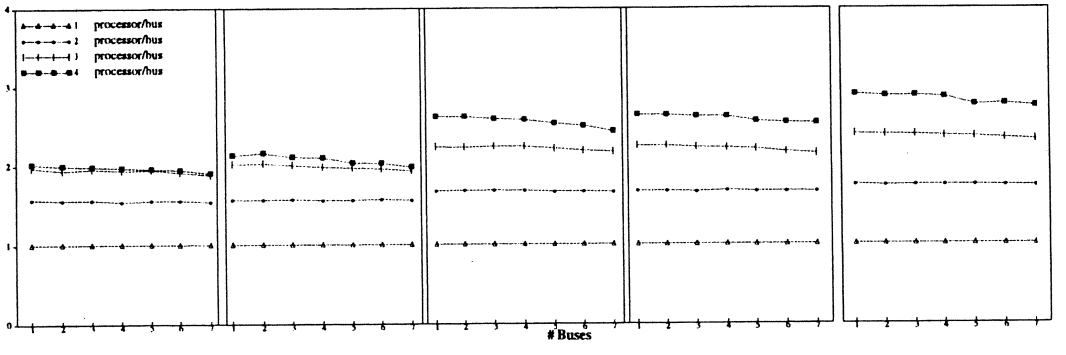

Figure 8: Influence of bus traffic for the matrix-matrix product.
From left to right the results of the bus speed-up for block sizes $b = 10, 12, 14, 16, 18$.

- the speed of a processing element
- the speed of data transfer
- the cache size of a processing element.

For the matrix-matrix multiplication (cf. Fig. 3) with $m = n = k$, block size KB = MB = NB = b and K $\times$ $b = k$ the number of floating point operations (flops) per block is given by

$$\#\,\texttt{flops} = 2\mathrm{K}b^3 = 2kb^2 \tag{4.1}$$

and the number of data transfers by

$$\#\,\texttt{transfers} = 2(\mathrm{K}+1)b^2. \tag{4.2}$$

Let α be the ratio of transfer time τ_t to the time τ_f needed for one floating point operation in DOUBLE PRECISION arithmetic, then the transfer time can be written as

$$\tau_t = \alpha\tau_f. \tag{4.3}$$

The processor performance can be expressed by

$$\Psi_{processor} = \frac{\#\,\texttt{flops}}{\texttt{execution time}} = \frac{2kb^2}{2kb^2\tau_f + 2(\mathrm{K}+1)b^2\tau_t} = \frac{k}{k+(\mathrm{K}+1)\alpha}\cdot\frac{1}{\tau_f}. \tag{4.4}$$

The leverage of the matrix-matrix multiplication (3.5) is denoted by

$$\texttt{Leverage} = \frac{2kb^2\tau_f}{2(\mathrm{K}+1)b^2\tau_t} = \frac{k}{(\mathrm{K}+1)\alpha}. \tag{4.5}$$

The performance per bus Ψ_{bus} is defined by

$$\Psi_{bus} = \texttt{Leverage}\cdot\Psi_{processor} = \frac{k^2}{[k+(\mathrm{K}+1)\alpha](\mathrm{K}+1)\alpha}\cdot\frac{1}{\tau_f}. \tag{4.6}$$

If the block size b is chosen such that the data cache is (nearly) fully occupied, the smallest possible value of K is obtained. This value results in a maximum processor performance and the leverage is maximal, too, and so is the bus performance Ψ_{bus}.

Let us return to the APP characteristics which influence the bus performance. First, we assume that it is possible to increase the speed of a processing element by a factor $\beta, \beta > 1$. At the same time we assume that the data transfer time remains invariable. In formula,

$$\left.\begin{array}{l}\tilde{\tau}_f = \frac{1}{\beta}\tau_f \\ \tilde{\tau}_t = \tau_t\end{array}\right\}\ \tilde{\alpha} = \frac{\tilde{\tau}_t}{\tilde{\tau}_f} = \alpha\cdot\beta. \tag{4.7}$$

The ratio of the *improved computation speed*'s bus performance and the *actual speed*'s bus performance becomes

$$\frac{\tilde{\Psi}_{bus}}{\Psi_{bus}} = \frac{k+(\mathrm{K}+1)\alpha}{k+(\mathrm{K}+1)\alpha\beta} < 1;\ \text{for } \beta > 1. \tag{4.8}$$

As a very undesirable effect, we get a decrease in the bus performance and the corresponding APP performance when the speed of the individual processing element increases.

Secondly, we assume that the transfer rate can be accelerated by a factor γ, $\gamma > 1$, and the computation speed will be unaltered. Then we get

$$\left.\begin{array}{l} \tilde{\alpha} = \frac{\alpha}{\gamma} \\ \tilde{\tau}_f = \tau_f \end{array}\right\} \tag{4.9}$$

and the bus performance becomes

$$\tilde{\Psi}_{bus} == \frac{k^2}{[\gamma k + (\mathrm{K}+1)\alpha](\mathrm{K}+1)\alpha} \cdot \frac{\gamma^2}{\tau_f}. \tag{4.10}$$

Compared to the original situation we get an acceleration of

$$\frac{\tilde{\Psi}_{bus}}{\Psi_{bus}} = \gamma \frac{k + (\mathrm{K}+1)\alpha}{k + (\mathrm{K}+1)\frac{\alpha}{\gamma}} > \gamma; \text{ for } \gamma > 1. \tag{4.11}$$

Without doubt a reduction of the transfer rate will improve the performance of the APP.

Thirdly, we consider the bus performance as a function of the cache contents. If the data cache size is enlarged by δ, then the new block size $\tilde{b}$ can be enlarged to

$$\tilde{b} = \sqrt{\delta} \cdot b \tag{4.12}$$

and for a fixed problem the new $\tilde{\mathrm{K}}$ will become

$$\tilde{\mathrm{K}} = \delta^{-\frac{1}{2}}\mathrm{K}. \tag{4.13}$$

The bus performance will increase then by a factor

$$\frac{\tilde{\Psi}_{bus}}{\Psi_{bus}} = \frac{k + (\mathrm{K}+1)\alpha}{k + (\delta^{-\frac{1}{2}}\mathrm{K}+1)\alpha} \cdot \frac{\mathrm{K}+1}{\delta^{-\frac{1}{2}}\mathrm{K}+1} > 1; \text{ for } \delta > 1, \tag{4.14}$$

since both the leverage and the performance per processing element grow. Notice that the gain is less spectacular than obtained for a reduced transfer speed.

Summarizing, from the possibilities to improve the bus performance described above the largest gain can be expected by an increase of the transfer speed, whereas a decrease of the cycle time for a floating point operation will even result in a lower bus performance.

5. The Cholesky factorization

The block Cholesky matrix-factorization has a computational complexity which is comparable to that of the matrix-matrix multiplication. In this section we describe two variants: the block left-looking factorization and the block right-looking factorization. A third variant, the block top-looking, is not considered: for this we expect a similar behavior as for the left-looking variant. All variants have exactly the same number of floating point operations. The major steps are described in terms of the BLAS [4, 5, 9] and the organization is such that each processor will perform BLAS operations on single block matrices only.

The Cholesky factorization of a symmetric matrix A is given by

$$A = L \cdot L^{\mathrm{T}}, \tag{5.1}$$

where L is a unique lower triangular matrix. Assume A can be partitioned into K^2 square blocks of order b such that k, the order of A, is equal to $\mathrm{K} \times b$. We first derive the block left-looking variant which is block column oriented. Next, the right-looking algorithm is considered.

5.1 Left-looking algorithm
This well-known factorization can be derived from the following block-matrix product

$$\begin{bmatrix} A_{11} & & \\ A_{21} & A_{22} & \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \cdot \begin{bmatrix} L_{11}^{\mathrm{T}} & L_{21}^{\mathrm{T}} & L_{31}^{\mathrm{T}} \\ & L_{22}^{\mathrm{T}} & L_{32}^{\mathrm{T}} \\ & & L_{33}^{\mathrm{T}} \end{bmatrix}. \tag{5.2}$$

We assume that in the previous $l-1$ steps the block matrices L_{11}, L_{21} and L_{31} — each consists of $(l-1) \times b$ columns — have been computed. In the current step l, the block matrices L_{22} and L_{32}, with a column width of b columns, must be updated. Then from the block matrix equality, we obtain

$$\begin{aligned} L_{21} \cdot L_{21}^{\mathrm{T}} + L_{22} \cdot L_{22}^{\mathrm{T}} &= A_{22} \\ L_{31} \cdot L_{21}^{\mathrm{T}} + L_{32} \cdot L_{22}^{\mathrm{T}} &= A_{32} \end{aligned} \tag{5.3}$$

The computation of the diagonal block L_{22} consists of two steps:

$$A'_{22} \leftarrow A_{22} - L_{21} . L_{21}^{\mathrm{T}}, \tag{5.4}$$

or, in Level 3 BLAS terms, a DSYRK operation, followed by

$$L_{22} \leftarrow \mathrm{Cholesky}(A'_{22}), \tag{5.5}$$

which can be performed by a Level 2 BLAS algorithm. We refer to this algorithm as DLLT. The block column matrix L_{32} can be obtained by first computing

$$A'_{32} \leftarrow A_{32} - L_{31} . L_{21}^{\mathrm{T}}, \tag{5.6}$$

a straightforward matrix-matrix multiplication and subtraction, and then afterwards L_{32} becomes

$$L_{32} \leftarrow A'_{32} . L_{22}^{-\mathrm{T}}. \tag{5.7}$$

In (5.7) a triangular system is solved with a multiple right hand side, which can be performed by the Level 3 BLAS routine DTRSM. In the next step another block column of L is updated, and after precisely K steps the factorization is completed. This algorithm is known as a left-looking algorithm because the data referred to is mainly on the left-hand side of the current block column.

5.1.1 Explicit parallelism by operating on single blocks The BLAS operations can be parallelized either implicitly or explicitly. In case of implicit parallel processing the BLAS are parallelized such that within a BLAS operation the work is equally divided over several processors. In case of explicit parallel processing, calculations on a single block — both input and output matrices consist of single blocks of dimension b — are not parallelized; however, these block operations themselves, scheduled more or less arbitrarily (taking into account the data dependencies) can be performed in parallel.

In Louter-Nool [10] parallelism is exploited in a dynamic way. Before the computation starts the dependencies between single block operations are uniquely described and when a processor has

accomplished some single block operation then it "looks" for another operation in a queue of "ready-to-start" processes. After completion of a block operation other processes can become "ready-to-start". This asynchronous approach makes it possible to reduce idle time substantially and, therefore, a highly efficient method is obtained. An important disadvantage of this method, however, is that an extensive administration of the state of the processes and the dependencies is required. SCHEDULE [3], the package that was used for that purpose in [10], is not available on the CRAY APP. Therefore, parallelism is considered at a lower level: a simpler static scheduling of operations is applied, although reduction of idle time is still one of the main issues.

To start with we discuss how the multiple block operations (5.4, 5.6, 5.7) can be written as single block operations and how parallelism can be introduced explicitly. The symmetric rank-k update can be rewritten as $l-1$ single block DSYRK operations. Since all of the operations write to the same data block A'_{22} they cannot be carried out concurrently. Furthermore, the matrix-matrix multiplication can be split up into $(\mathrm{K}-l)\times(l-1)$ single block DGEMM multiplications of which the explicit level of parallelism is only $(\mathrm{K}-l)$. Finally, the multiple block operation DTRSM (5.7) can be partitioned into $(\mathrm{K}-l)$ single block operations to be performed simultaneously.

5.1.2 Implementation and performance of single block BLAS Programming the CRAY APP has been reduced to implementing the four basic single block operations: DSYRK, DGEMM, DTRSM and DLLT. As mentioned in subsection 2.2, local BLAS implementations as described in [6] are not available, but it is possible to get pipeline rather than scalar performance by means of a very restricted set of Extended Math. Library routines [1].

Both the matrix-matrix product DGEMM and the symmetric rank-k update DSYRK are performed by a call to _dmmm (see section 2.2). For the latter this implies that the symmetry of the resulting matrix is not taken into account and twice as many floating point operations are performed as actually required. Other attempts to get a better performance for DSYRK failed. Considerable effort has been put in the efficient implementation of DTRSM and DLLT, although we consider our ultimate performance given in Table 1 rather disappointing.

Block routine	Performance	# flops
DGEMM	25.6 Mflops	$2b^3$
DSYRK	13.6 Mflops	b^3
DTRSM	6.7 Mflops	b^3
DLLT	4.0 Mflops	$(2b^3+3b^2+b)/6$

Table 1: Performance results and number of Floating-Point operations of local BLAS operations for a block size $b=18$

5.1.3 Parallel implementation A straightforward implementation of the block left-looking factorization is given in the left column of Fig. 9. Obviously, the load balancing is far from optimal. The DSYRK operations are carried out by a single processor just as the DLLT operation on the diagonal blocks. A better balancing is achieved by performing the matrix-matrix multiplication DGEMM and the symmetric rank-k update DSYRK simultaneously. These processes are data independent: DSYRK updates the diagonal blocks whereas DGEMM writes to off-diagonal blocks. So, one processor updates the diagonal block, whereas the other processors perform single block matrix-matrix multiplications as is illustrated in the right column of Fig. 9. This approach is satisfactory if the execution time of a block multiplication is comparable with the execution time for a single block symmetric rank-k update. If not, then a better solution can be achieved when a "WHILE" construction is applied, especially when the number of blocks in the l-th column is larger than the number of processing elements p.

```
      SUBROUTINE LLCHOL ( ... )
C     Left-looking Block Cholesky
C      . . .
C     For Column L do
C      . . .
      DO I = 1, L-1
C Compute A[L,L]=
C   A[L,L]-A[I,L].Transpose(A[I,L])
         CALL DSYRK( ... )
      END DO
C
CPCF PARALLEL
CPCF PRIVATE I, J
CPCF PDO
     DO J = L+1, K
C
C
C
C
C
C
C
       DO I = 1, L-1
C Compute A[J,L]=
C    A[J,L]-A[J,I].Transpose(A[I,L])
          CALL DGEMM( ... )
       END DO
C
     END DO
C
CPCF SINGLE PROCES
C Compute Cholesky
C    factorization of A[L,L]
     CALL DLLT( ... )
CPCF END SINGLE PROCES
C
CPCF PRIVATE J
CPCF PDO
     DO J = L+1, K
C Solve A[J,L]=
C    A[J,L].Inverse(Transpose(A[L,L]))
       CALL DTRSM( ... )
     END DO
C
CPCF END PARALLEL
C
     RETURN
     END
```

```
      SUBROUTINE LLCHOL ( ... )
C     Improved Left-looking Block Cholesky
C      . . .
C     For Column L do
C       . . .
C
C
C
C
C
C
CPCF PARALLEL
CPCF PRIVATE I, J
CPCF PDO
      DO J = L, K
         IF( J.EQ.L )THEN
            DO I = 1, L-1
C Compute A[L,L]=
C    A[L,L]-A[I,L].Transpose(A[I,L])
               CALL DSYRK( ... )
            END DO
         ELSE
            DO I = 1, L-1
C Compute A[J,L]=
C    A[J,L]-A[J,I].Transpose(A[I,L])
               CALL DGEMM( ... )
            END DO
         END IF
      END DO
C
CPCF SINGLE PROCES
C Compute Cholesky
C    factorization of A[L,L]
      CALL DLLT( ... )
CPCF END SINGLE PROCES
C
CPCF PRIVATE J
CPCF PDO
      DO J = L+1, K
C Solve A[J,L]=
C    A[J,L].Inverse(Transpose(A[L,L]))
         CALL DTRSM( ... )
      END DO
C
CPCF END PARALLEL
C
      RETURN
      END
```

Figure 9: Two parallel implementations of left-looking Cholesky factorization. The straightforward implementation shown in the left column has a substantially lower level of parallelism than the right column's implementation.

```
      SUBROUTINE LLCHOL ( ... )
C     Left-looking Block Cholesky with ''WHILE'' construction
C      . . .
C     For Column L do
C      . . .
      JJ = L - 1
C
CPCF PARALLEL
CPCF PRIVATE I, J
C
   10 CONTINUE
C
CPCF CRITICAL SECTION
      JJ = JJ + 1
      IF( JJ.LE.K )THEN
         J = JJ
      ELSE
         CALL MCP_ECS( )
         GOTO 20
      END IF
CPCF END CRITICAL SECTION
      IF( J.EQ.L )THEN
         DO I = 1, L-1
C Compute A[L,L]=A[L,L]-A[I,L].Transpose(A[I,L])
            CALL DSYRK( ... )
         END DO
C Compute Cholesky factorization of A[L,L]
         CALL DLLT ( ... )
      ELSE
         DO I = 1, L-1
C Compute A[J,L]=A[J,L]-A[J,I].Transpose(A[I,L])
            CALL DGEMM( ... )
         END DO
      END IF
C
      GOTO 10
   20 CONTINUE
      CALL MCP_BARRIER( )
C
CPCF PRIVATE J
CPCF PDO
      DO J = L+1, K
C Solve A[J,L]=A[J,L].Inverse(Transpose(A[L,L]))
         CALL DTRSM( ... )
      END DO
C
CPCF END PARALLEL
C
      RETURN
      END
```

Figure 10: Parallel implementation of the left-looking Cholesky factorization with the "WHILE" construction. The update of the diagonal block and those of the off-diagonal blocks are carried out concurrently. As soon as either a diagonal or an off-diagonal block update is terminated, the processor may continue with updating another off-diagonal block.

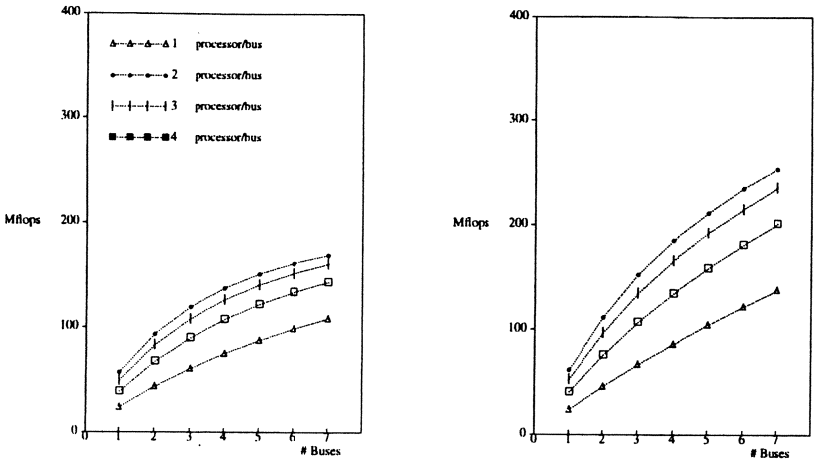

Figure 11: Performance results of two left-looking block Cholesky factorizations. The left figure corresponds to the implementation of the right column of Fig.9, whereas the right figure shows the Mflop-rate achieved for the "WHILE" construction of Fig. 10.

The "WHILE" construction, illustrated in Fig. 10, consists of an endless loop and the operations are performed as long as a logical expression — here, (JJ.LE.K) — is true. A critical section is used to allow processors to update the shared variable JJ. The loop is terminated when all processors have found a value JJ which is larger than K. The MCP_BARRIER is called to guarantee that all operations are finished when the final update of the off-diagonal blocks, performed by DTRSM, starts. This "WHILE" construction makes it possible to execute processes asynchronously. As soon as a sequence of DSYRK or DGEMM is terminated, the processor starts with another operation. Moreover, the sequence of DSYRK operations can immediately be followed by a call of DLLT, since its input is independent of the operations on the off-diagonal blocks in column l. We remark that the idle time can be reduced even more by performing the DTRSM immediately after the $(l-1)$-th DGEMM contribution under the condition that the DLLT on the diagonal block has been completed. However, a comprehensive registration of the status of the processes will be required then.

5.1.4 Performance of left-looking implementations Fig. 11 shows the performance results obtained for two left-looking variants: the left figure for the left column of Fig. 9, the right figure for the implementation presented in Fig. 10. So, for the left-looking block Cholesky factorization, with its decreasing amount of computational work per block column, asynchronous job scheduling can improve the performance substantially. Moreover, the performance gain obtained by going from 3 to 4 processors per bus is less obvious than from 1 to 2 processors per bus. However, an increase of the total number of processors may not result in a performance increase. For instance, the execution time achieved on 14 processors (viz., 7 buses × 2 processors/bus) is less than the time achieved on 15 processors (viz., 5 buses × 3 processors/bus) for both implementations. In Fig. 12, we present the *bus efficiency*, defined by the ratio of bus speed-up (3.7) to the number of processors per bus. Although the same steps are performed, we observe that due to a better job scheduling the bus efficiency for the full configuration has been raised from less than 30 % to nearly 40 %.

5.2 Right-looking algorithm

The block right-looking algorithm has an updating pattern which significantly differs from the left-looking algorithm. The latter does not perform operations before they are actually needed; information from the past — at the left side of the current column — is used to update that column. The right-

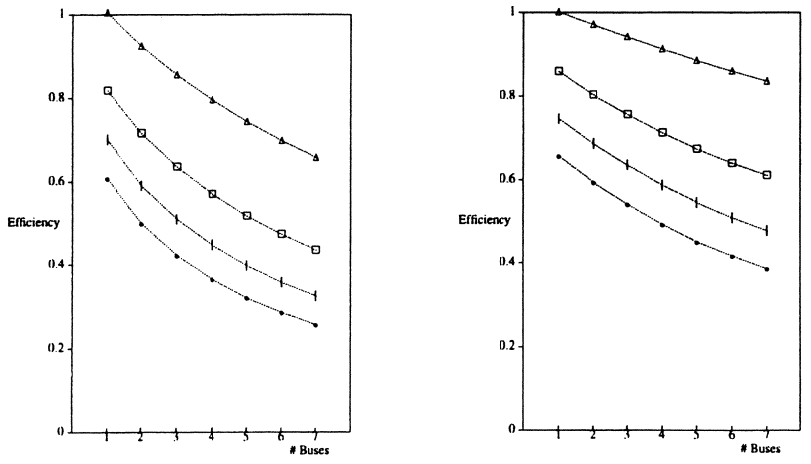

Figure 12: Bus efficiency
for the two left-looking block Cholesky factorizations of Fig. 11.

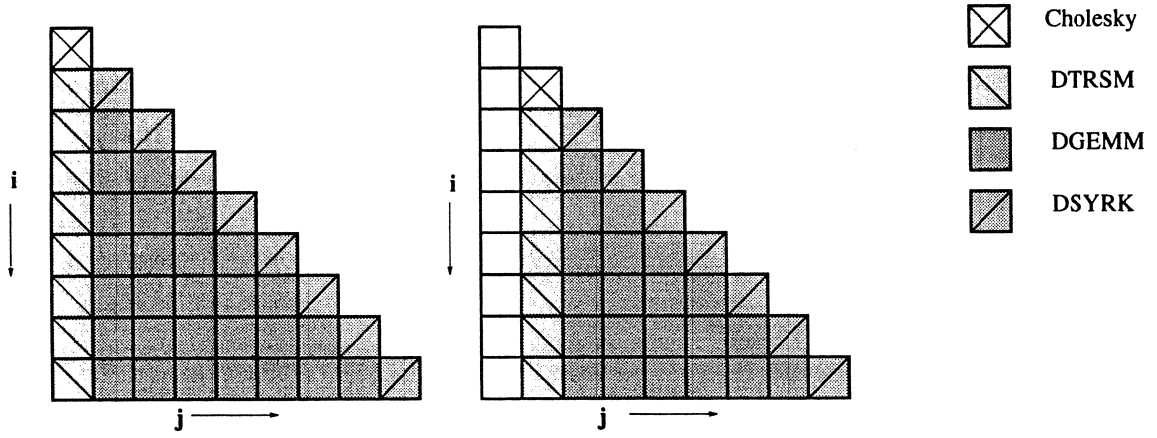

Figure 13: Update scheme for the first two steps of right-looking block Cholesky factorization.

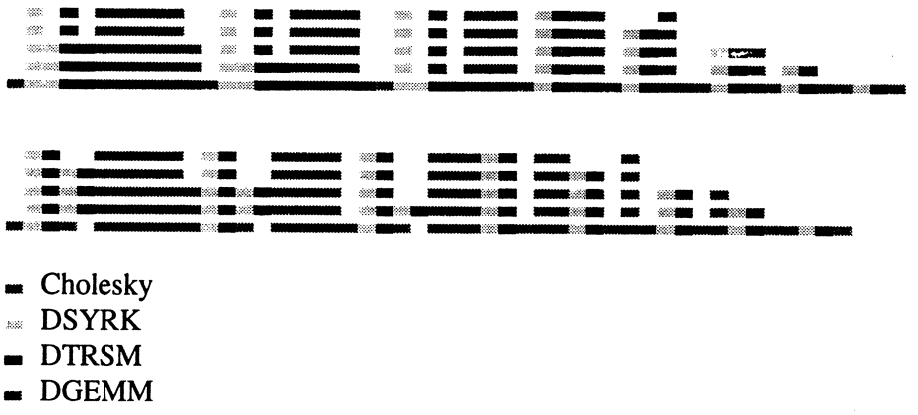

Figure 14: Right-looking block Cholesky factorizations on 5 processors.
The upper pattern illustrates the parallel execution of Fig. 13's implementation.
A combination of DSYRK and DTRSM and the Cholesky operations can reduce idle time as is illustrated by the lower pattern.

looking algorithm, however, processes information as soon as is becomes available. If, for instance, the first block column of L has been computed, then all blocks at the right-hand side can be updated with contributions from the first block column: the diagonal blocks by a DSYRK operation and the off-diagonal matrices by a matrix-matrix multiplication.

We prefer to describe the right-looking implementation in terms of single block operations directly. Again it is assumed that A (and L) are of order k and $k = \mathrm{K} \times b$, where b denotes the block size. In the first step the factorization can be performed analogously to the left-looking algorithm:

$$L[1,1] \leftarrow \text{Cholesky}(A[1,1]), \tag{5.8}$$

$$L[i,1] \leftarrow A[i,1].L[1,1]^{-\mathrm{T}}, \tag{5.9}$$

for $i = 2, \cdots, \mathrm{K}$. For the remaining part of A the diagonal blocks $A[i,i]$, $i = 2, \cdots, \mathrm{K}$, are updated with respect to the first block column of L :

$$A[i,i] = A[i,i] - L[i,1].L[i,1]^{\mathrm{T}}, \tag{5.10}$$

and the off-diagonal elements by a matrix-matrix multiply and add operation

$$A[i,j] = A[i,j] - L[i,1].L[j,1]^{\mathrm{T}}, \tag{5.11}$$

for $i = 3, \cdots, \mathrm{K}$, $j = 2, \cdots, i-1$. As soon as the first step is completed, the same operations can be applied on the recently updated submatrix of A. The operations of the first two steps are illustrated by Fig. 13. In Fig. 15 the implementation of the computation of the l-th block column and the update of its remaining right-hand part is given. Note that all updates of the right-hand side of the l-th block column can be done in parallel. Fig. 14 shows that some idle time can be saved by combining the symmetric rank-k update and the Cholesky factorization of the current block. Actually, the reduction of idle time will be larger than shown, because relatively expensive operations are combined.

```
      SUBROUTINE RLCHOL( ... )
C     Right-looking Block Cholesky
C      . . .
C     For Column L do
C
C Compute Cholesky factorization of A[L,L]
      CALL DLLT( ... )
C
      IF( L.LT.K )THEN
CPCF PARALLEL
CPCF PRIVATE J
CPCF PDO
         DO J = L+1, K
C Solve A[J,L]=A[J,L].Inverse(Transpose(A[L,L]))
            CALL DTRSM( ... )
         END DO
C
CPCF PRIVATE J
CPCF PDO
         DO J = L+1, K
C Compute A[J,J]=A[J,J]+A[J,L].Transpose(A[J,L])
            CALL DSYRK( ... )
         END DO
C
CPCF SINGLE PROCESS
C
         NUMBER = ( K - L - 1 ) * ( K - L ) / 2
         JJ = L
         KK = K
C
CPCF END SINGLE PROCESS
CPCF PRIVATE I, IJ, J
CPCF PDO
         DO IJ = 1, NUMBER
C
CPCF CRITICAL SECTION
C Determine the block matrix on which the matrix-matrix multiply
C routine DGEMM must be performed
            IF( KK.EQ.K )THEN
               JJ = JJ + 1
               KK = JJ + 1
               I  = KK
               J  = JJ
            ELSE
               KK = KK + 1
               I  = KK
               J  = JJ
            END IF
CPCF END CRITICAL SECTION
C
C Compute A[I,J]=A[I,J]+A[I,L].Transpose(A[J,L])
            CALL DGEMM( ... )
C
         END DO
CPCF END PARALLEL
C
      END IF
C
      RETURN
      END
```

Figure 15: Parallel implementation of the right-looking block Cholesky factorization.

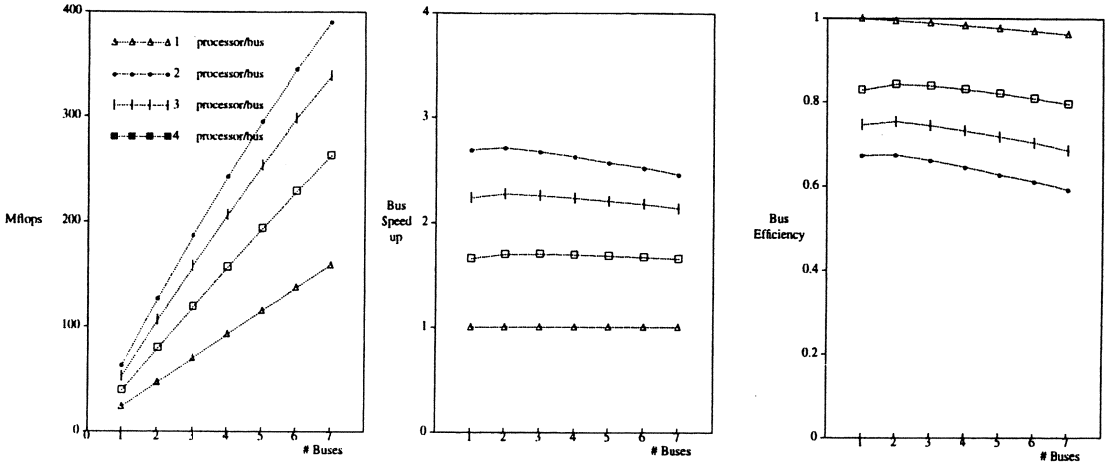

Figure 16: Results of the right-looking block Cholesky factorization.
From left to right: the Megaflops, the bus speed-up and the bus efficiency for $b = 18$, respectively.

5.2.1 Performance of right-looking implementation The main difference in performance for the left and right-looking implementations comes from the level of parallelism. For the left-looking algorithm the number of independent processes for the l-th block column is equal to the number of blocks in that column, viz., $\mathrm{K} - l$, in case diagonal and off-diagonal blocks are updated concurrently. For the right looking algorithm this number is much higher, viz.,

$$\frac{1}{2}\,(\mathrm{K}-l).(\mathrm{K}-l+1), \tag{5.12}$$

since all DSYRK and DGEMM updates are data-independent.

Fig. 16 displays performance results of the block right-looking variant. The results for a small number of processors are comparable to the left-looking implementations, but for a larger number of processors we see that the right-looking implementation performs much better due to the higher level of parallelism. The speed-up pattern approximates the one achieved for the matrix-matrix multiplication (cf. Fig. 8). We also refer to [2] and [10] for similar experiments.

It is worth noting, however, that for the left-looking algorithm the data reuse is better: only data from the current block is updated and can be kept in cache and as we have seen in section 3.4: this will probably happen. For the right-looking algorithm no data can be reused and, consequently, more bus traffic takes place. Apparently, this disadvantage does not play an important role.

Finally, we discuss performance results for different block sizes. In Fig. 17 all lower lines correspond to a block size of 10, the upper lines to a block size of 18, the third line in each picture corresponds to a block size of 16. Horizontally, the order of the matrix is displayed. For a fixed matrix order a decrease of the block size delivers a higher level of parallelism, which is not translated into a higher efficiency. From Fig. 17 it is clear that for a block size of 10 the bus traffic negatively influences the performances, whereas a larger block size with a corresponding higher matrix-matrix multiplication's leverage value takes care of a high speed on the full bus configuration.

6. Conclusions

We have shown the influence of bus traffic on the performance by means of simple block matrix-matrix products. A bus speed-up of nearly three was achieved for matrix-matrix multiplication and optimally

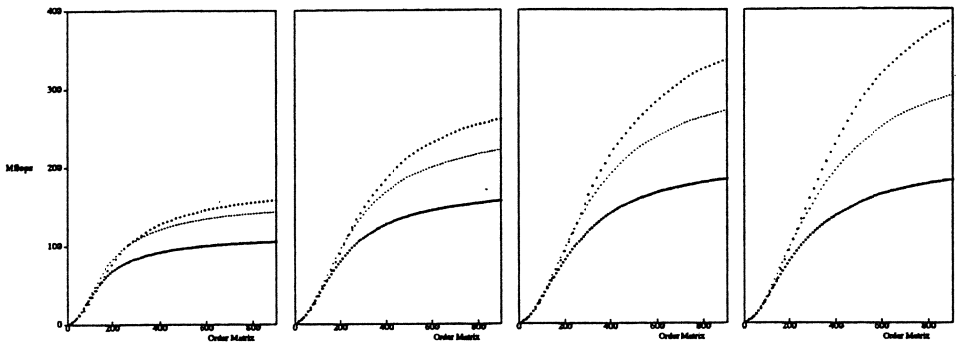

Figure 17: Performance of the right-looking Cholesky factorization for block sizes: $b = 10, 16, 18$. From left to right the Mflops obtained for runs on $7 \times 1, 7 \times 2, 7 \times 3, 7 \times 4$ processors

filled cache. We do not expect that better results can be gained with the addition of more processing elements on the same bus.

For the Cholesky decomposition a reasonable efficiency was achieved, not only because of the high matrix-matrix multiplication performance but also because of the choice of algorithm. The right-looking implementation with its higher level of parallelism performs significantly better than the left-looking implementations. For both algorithms the number of synchronization points is significant. By means of taking different jobs together with the so-called "WHILE" construction considerable execution-time reduction can be achieved. the best results for the APP configuration were obtained for partitionings with cache contents as large as possible. A partitioning into smaller blocks with a corresponding higher level of parallelism does not lead to a better performance (as is shown by Fig. 17) because of increased data traffic.

ACKNOWLEDGEMENTS

The author wishes to thank Alan Stewart and Herman J.J. te Riele for their constructive comments.

REFERENCES

1. Cray Research Superservers, Inc. *CRAY APP Programmer's Reference*, April 1992. Extended Math. Routines.
2. Krister Dackland, Erik Elmroth, Bo Kågström, and Charles Van Loan. Parallel Block Matrix Factorizations on the Shared-Memory Multiprocessor IBM 3090 vf/600j. *The International Journal of Supercomputer Applications*, 6:69–97, 1992.
3. F.B. Hanson and D.C. Sorensen. The SCHEDULE parallel programming package with recycling job queues and iterated dependency graphs. Technical Report ANL-MCS-P22-0189, Argonne National Laboratory, 1989.
4. J.J.Dongarra, J. Du Croz, I. Duff, and Hammerling S. A set of level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, 1990.
5. J.J.Dongarra, J. Du Croz, S. Hammerling, and R.J. Hanson. An extended set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 14(1):1–32, 1988.
6. S. Lennart Johnsson and Luis F. Ortiz. Local Basic Linear Algebra Subroutines LBLAS for distributed memory architectures and languages with array syntax. *The International Journal of*

Supercomputer Applications, 6:322–350, 1992.

7. C.-H Lai and H.J.J. te Riele. Some experiences of solving 1-D semiconductor device equations on a matrix coprocessor by a domain decomposition method. Technical Report NM-R9304, CWI, February 1993.
8. C.-H Lai, H.J.J. te Riele, and A. Ualit. Parallel experiments with simple linear algebra operations on a Cray S-MP System 500 matrix coprocessor. Technical Report NM-N9301, CWI, June 1993.
9. C.L. Lawson, R.J. Hanson, D. Kincaid, and F.T. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.
10. M. Louter-Nool. Block-Cholesky for parallel processing. *Applied Numerical Mathematics*, 10:37–57, 1992.
11. A. Stewart, M. Louter-Nool, H.J.J. te Riele, and D.T. Winter. An Investigation of Data Reuse on the Cray S-MP System 500. Technical Report NM-R9415, CWI, July 1994.