Parallel experiments with simple linear algebra operations on a Cray S-MP System 500 matrix coprocessor

C.-H. Lai, H.J.J. te Riele, A. Ualit

Department of Numerical Mathematics

Note NM-N9301 June 1993

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.

Copyright © Stichting Mathematisch Centrum
P.O. Box 4079, 1009 AB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Parallel experiments with simple linear algebra operations on a Cray S-MP System 500 matrix coprocessor

C.-H. Lai, H.J.J. te Riele and A. Ualit

CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

The main characteristics of the Cray S-MP System 500 matrix coprocessor are described and the results are presented of parallel experiments with matrix-vector and matrix-matrix operations on a coprocessor configuration consisting of twenty-eight processing elements. The performance results are compared with a theoretical model involving computing and communication time, and cache size characteristics.

1991 Mathematics Subject Classification: 65Y05, 69C12.
Keywords & Phrases: MIMD computer, parallel matrix coprocessor, linear algebra operations.

Note: This research was done while the first author was visiting CWI as an ERCIM fellow. His current address is: School of Mathematics, Statistics and Computing, University of Greenwich, Wellington Street, Woolwich, London SE18 6PF, UK. Email address: c.h.lai@greenwich.ac.uk.

Note NM-N9301
ISSN 0169-0388
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1 Introduction

In 1991 CWI acquired an FPS System 500 64-bit distributed memory multiprocessor system involving one 72 MIPS SPARC scalar processor and a matrix coprocessor with twenty-eight 40 MIPS i860 processors, configured in seven buses, each consisting of four processing elements. In December 1991, Cray took over the FPS System 500 production and maintenance, and since then the machine is called the Cray S-MP System 500. Figure 1 gives a schematic picture of the system, taken from [FPS91b], where the structure of the matrix coprocessor is enlarged. The vector coprocessor is not part of the CWI configuration. More information about this system is given in Appendix A to this paper.

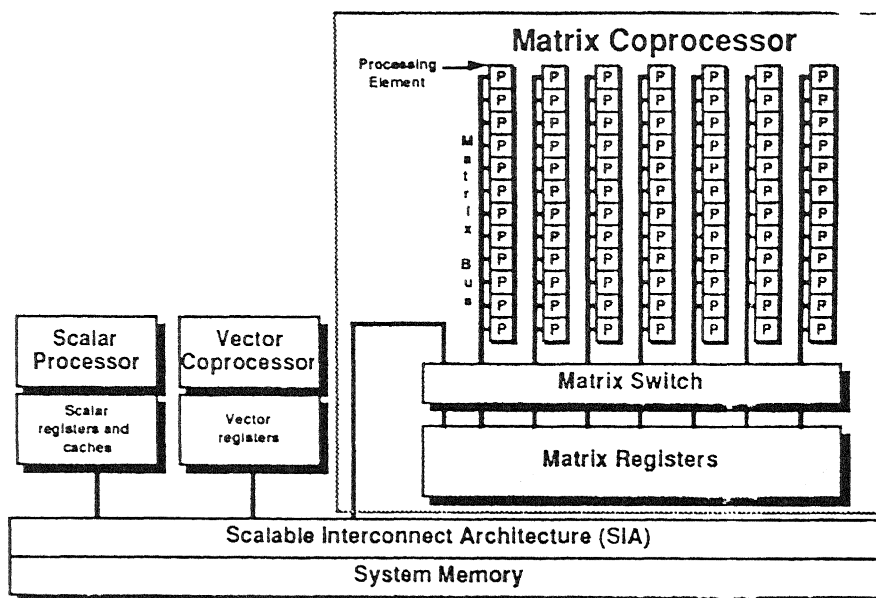


Figure 1: Cray S-MP System 500 Matrix Coprocessor Architecture

In order to get acquainted with this machine, we have carried out several experiments with simple linear algebra operations (matrix-vector and matrix-matrix). Based on the vendor's information we have designed a theoretical performance model and compared this with the actual performance figures. As can be expected, algorithms with high data locality, i.e., with many floating-point operations for each data access, perform best on the matrix coprocessor. There is one restrictive characteristic of the System 500 which should be mentioned here, namely that at a given time, only one processing element on each bus can access the matrix registers for data transport. So while one processing element on a bus is transporting data, the others should spend their cycles on computing, otherwise they have to wait. This means that low locality algorithms can only run efficiently on a one processing element per bus configuration.

The organization of the paper is as follows. In Section 2 we describe, by means of

an example of a vector dot-product computation, how programs can be parallelized on the matrix coprocessor [FPS91b]. In Section 3 we present the results of experiments with matrix-vector multiplication, and in Section 4 those for matrix-matrix multiplication. In both sections the results of varying the number of buses, and the number of processors per bus are discussed. Furthermore, we discuss how the performance is influenced by varying the processor's cache block size. In Section 5 we present a simple performance model and compare this with the results of our experiments. Finally, some conclusions are drawn. In Appendix B we give the listings of our parallel Fortran block subroutines for matrix-vector and matrix-matrix multiplication.

2 An example of parallel programming on the System 500 matrix coprocessor

By means of an example of the computation of the vector dot-product we illustrate how a subroutine can be parallelized on the System 500 matrix coprocessor. To that end, one has to insert the so-called **mpp** and **pfp** comment directives. Their meanings are explained shortly in additional comment lines. Details about **mpp** and **pfp** directives are given in Appendix A.1.

The dot-product loop is executed in parallel on the available processing elements (the *number of* available processing elements is set in the calling program), and after completion of the loop the partial sums are collected in the output parameter c.

```
C
C Parallel dotproduct
C
C Four mpp directive lines follow which indicate that this subroutine
C should be executed on the matrix coprocessor, and which specify the
C use of the subroutine's parameters
CMCP subroutine dotproduct (a, b, c, n)
CMCP input   real*8      a(n), b(n)
CMCP output  real*8      c
CMCP input   integer*4   n
      subroutine dotproduct (a, b, c, n)
      real*8      a(*), b(*), c
      integer*4   n
      real*8      psum
C
C Initialize  c
      c = 0.0D0
C
C pfp directive: start of parallel region in this subroutine
CPCF PARALLEL
C
C pfp directive: psum is local to each process
CPCF PRIVATE psum
C
C Initialize psum on each processor
      psum = 0.0D0
C
C pfp directive: iterations of the following loop are to be executed in
C                parallel, where the iterations of the do-loop are split
C                into equal blocks among all available processors.
```

```
C                 There is an implicit barrier (synchronization point)
C                 at the end of the loop: this option can be switched off
C                 by means of the NOWAIT pfp directive.
CPCF PDO BLOCKED
      do 20 i = 1, n
         psum = psum + a(i)*b(i)
   20 continue
C
C Compute total result from partial results in each processor
C
C pfp directive: the code which follows is processed by one processor
C                 at a time
CPCF CRITICAL SECTION
      c = c + psum
C
C pfp directive: end of section of code that was started by the
C                 CRITICAL SECTION directive
CPCF END CRITICAL SECTION
C
C pfp directive: end of section of code that was started by the
C                 PARALLEL directive
CPCF END PARALLEL
      return
      end
```

3 Some experimental results for the matrix-vector product

Suppose the matrix A is of size $n_1 \times n_2$, partitioned in blocks of size $\alpha_1 \times \alpha_2$, and b and c are vectors of size n_2, n_1 and partitioned in segments of size α_2, α_1 respectively. Furthermore, we assume that $n_1 = \alpha_1 N_1$, $n_2 = \alpha_2 N_2$. Consider the *matrix-vector product* $c = Ab$, where $A = (A_{ik})$, $b = (b_k)$ $(k = 1, N_2)$ and $c = (c_i) = \sum_{k=1}^{N_2} A_{ik} b_k$ $(i = 1, N_1)$. We assume that α_1, α_2 are chosen so that the block matrix A_{ik}, and the vector segments b_k and c_i can be stored completely in the cache (each processing element has a data cache of 8 KBytes), i.e. $\alpha_1\alpha_2 + \alpha_2 + \alpha_1 \leq S^L$, where $S^L = 2048$ for single precision arithmetic and $S^L = 1024$ for double precision arithmetic.
We discuss three different implementations for the *matrix-vector multiplication*.
The first implementation is a standard *block-dot product* approach and the second implementation is a standard *block-saxpy product* approach for *matrix-vector multiplication*. The third implementation is a standard *non-blocked dot product* approach.
(Note: in the sequel, the superscripts and the subscripts d or s indicate the *block-dot* and the *block-saxpy* algorithm, respectively. In what follows, A^L, b^L and c^L are cache *work-space* arrays and the superscript L means local.)

First implementation:
- *Block-dot algorithm: ik-version*

(A_{ik} is a block matrix, b_k and c_i are segments of the vectors b and c)

```
cpcf   parallel
cpcf   pdo
         For i=1:N1
             c^L(1 : α1) = 0.0d0
             For k=1:N2
cpcf   critical section bus
                 load A_ik and b_k into the cache arrays A^L and b^L, respectively.
cpcf   end critical section
                 c^L = c^L + A^L * b^L
             End
cpcf   critical section bus
             store array c^L = c_i into the main memory.
cpcf   end critical section
         End
cpcf   end parallel
```

The algorithm is parallelized over the *i-loop*. So at a given time, each processor is computing independently on different segments c_i, $i = 1, \ldots, N_1$, of c (see appendix B, locality example 1 *matrix-vector multiplication*). In the inner *k-loop* of the algorithm, the segment

c^L is kept in cache. So the total number of operations (additions and multiplications) in this algorithm is $2n_1n_2$, and the total number T_d of reads and stores in this algorithm is given by

$$T_d = N_1\{N_2(\alpha_1\alpha_2 + \alpha_2) + \alpha_1\} = n_1 + n_1n_2(1 + \frac{1}{\alpha_1}) \tag{1}$$

Note that the matrix A is read once, the vector b is read N_1 times and the vector c is stored once. T_d is independent of α_2 for fixed order of the matrix A.

Second implementation:

- *Block-saxpy algorithm: ki-version*

(A_{ik} is a block matrix, b_k and c_i are segments of vectors b and c)

```
        c(1 : n1) = 0.0d0
cpcf    parallel
cpcf    pdo
        For k=1:N2
cpcf    critical section bus
              load b_k into the cache array b^L.
cpcf    end critical section
              For i=1:N1
cpcf    critical section bus
                  load A_ik into the cache array A^L.
cpcf    end critical section
                  c^L = A^L * b^L
cpcf    critical section
                  c_i = c_i + c^L
cpcf    end critical section
              End
        End
cpcf    end parallel
```

This algorithm is parallelized over the *k-loop*. The segment b^L is kept in cache during execution of the *i-loop*. In this loop, the segments c_i, $i = 1, \ldots, N_1$, of the global variable c are updated. The pfp directive **cpcf critical section** ensures that no two processors can do this update at the same time (i.e. the directive specifies that only a single process will execute this section of code at a time). So at a given time, each processor is computing dependently on segment c_i (see appendix B, locality example 2 *matrix-vector multiplication*). In a similar way, as in the previous algorithm, the total number T_s of reads and stores between the main memory and the cache in the *block-saxpy* algorithm is approximately given by

$$T_s \approx N_2\{\alpha_2 + N_1(\alpha_1\alpha_2 + 2\alpha_1)\} = n_2 + n_1n_2(1 + \frac{2}{\alpha_2}) \tag{2}$$

We notice that T_s is independent of the value of α_1 for fixed order of the matrix A. The matrix A and the vector b are read once and the vector c is loaded N_2 times and stored N_2 times.

We have carried out various experiments in double precision with these implementations. We measured the *MFLOP-rates* of the first and the second implementation for $\alpha_1 = \alpha_2 = 30$ and $n = i * 30$, $i = 1, ..., 30$, on the following configurations of the matrix coprocessor: $p = n_b \times n_p$ processors (where n_b denotes the number of buses and n_p the number of processors per bus), for $n_b = 7$, $n_p = 1, ..., 4$, and for $n_p = 4$, $n_b = 1, ..., 7$. The results are displayed in figures 3.1-3.4. The peak performance of both implementations is reached when $n = 840$ and also the performance increases with the number of processors in use.

We measured the total computing time varying n_b and keeping fixed n_p and the total computing time varying n_p and keeping fixed n_b. The results are given in tables 3.1 and 3.2. We also have computed the speedups for the various configurations given above. These are given in figures 3.5 and 3.6.

If a certain task for one processor requires t_c seconds computing time and t_m seconds communication time, then the parallel execution of N such tasks on a configuration of p processors (with n_b buses and n_p processors per bus) requires $\lceil \frac{N}{p} \rceil t_c$ seconds computing time, and $\lceil \frac{N}{n_b} \rceil t_m$ seconds communication time (since processors on the same bus can not communicate with the matrix registers concurrently). The total execution (computing and communication) time depends on the ratio between t_c and t_m. If $t_c \gg t_m$ then most of the processor communication on a single bus will be done while the other processors on that bus are busy with computing. If $t_c \ll t_m$ then the computing on the bus processors will be done while one of them is busy with communication. In all other cases the total execution time depends on the extent to which the computing and communication parts can be overlapped, and this in turn depends on the precise place(s) in the algorithm where communication has to be carried out.

For the parallel execution of our *matrix-vector* algorithm, let C_d and C_s denote the computing times for the *block-dot* and the *block-saxpy* versions, respectively, and let M_d and M_s denote the corresponding communication times. If R is the number of floating point operations per second and r the rate, in words per seconds, by which a block matrix or a segment vector can be read or stored then we have the following (optimistic) estimates ($\lceil x \rceil$ is the smallest integer $\geq x$, $\lfloor x \rfloor$ is the largest integers $\leq x$):

(3) $$C_d \approx \lceil N_1/p \rceil 2\alpha_1 n_2/R$$

(4) $$M_d \approx \lceil N_1/n_b \rceil \{(1+\alpha_1)n_2 + \alpha_1\}/r$$

(5) $$C_s \approx \lceil N_2/p \rceil 2\alpha_2 n_1/R + N_2 n_1/R$$

(6) $$M_s \approx \lceil N_2/n_b \rceil \alpha_2(1+n_1)/r + 2N_2n_1/r$$

The terms N_2n_1/R in C_s and $2N_2n_1/r$ in M_s are due to the fact that no two processors are allowed to execute the update statement $c_i = c_i + c^L$ concurrently. From (3)-(6) it follows that if we increase the value of n_b, while keeping all other parameters (including n_p) fixed, then the computing and the communication times will decrease. However, if we increase n_p, we see that the computing times will decrease, but not the communication times.

We have carried out a number of experiments in double precision to verify these properties. We measured the total computing times and the corresponding *MFLOP-rates* of the two implementations varying the block sizes of the matrix A. For the *block-dot* algorithm, we took rectangular blocks of size: $\alpha_1 = 6$, $\alpha_2 = 36$; $\alpha_1 = 18$, $\alpha_2 = 36$; $\alpha_1 = 18$, $\alpha_2 = 42$; $\alpha_1 = 36$, $\alpha_2 = 18$; $\alpha_1 = 36$, $\alpha_2 = 24$. For the *block-saxpy* algorithm, we took similar blocks, but with α_1 and α_2 interchanged. We fixed the configuration of the matrix coprocessor at $p = 28$ processors, i.e. $n_b = 7$, and $n_p = 4$. The results are displayed in figures 3.7 and 3.8.

Third implementation:

The *matrix-vector* algorithm is a standard dot product approach, so that $c = Ab$, where $A = (A_{ik})$ is a matrix of size $n_1 \times n_2$, $b = (b_k)$, $c = (c_i)$ are two vectors of size n_2, n_1 respectively, and $c_i = \sum_{k=1}^{n_2} A_{ik}b_k$, $i = 1, ..., n_1$. A is stored by means of a row storage scheme. This algorithm is parallelized over the *i-loop* in such a way that different processors will treat different iterations of the loop (see appendix B, example 3 *matrix-vector multiplication*). We measured the *MFLOP-rates* of this implementation for $n_1 = n_2 = 200 * i$, $i = 1, ..., 9$ by varying the values of n_p and keeping n_b fixed, and by varying the values of n_b and keeping n_p fixed. The results are displayed in figures 3.9 and 3.10.

Table 3.1: The total computing time for the *block-dot* algorithm (in seconds) with $n_1 = n_2 = n$.

n	$n_b \times n_p$							
n	1×1	2×1	3×1	4×1	5×1	6×1	7×1	
210	1.200	0.695	0.528	0.362	0.362	0.363	0.195	$\times 10^{-2}$
420	4.718	2.378	1.706	1.374	1.041	1.042	0.709	$\times 10^{-2}$
630	10.600	5.585	3.561	3.056	2.572	2.063	1.563	$\times 10^{-2}$
840	18.820	9.428	6.756	4.751	4.09	3.427	2.746	$\times 10^{-2}$
n	1×2	2×2	3×2	4×2	5×2	6×2	7×2	
210	0.729	0.384	0.375	0.208	0.208	0.210	0.201	$\times 10^{-2}$
420	2.447	1.413	1.056	0.732	0.715	0.715	0.379	$\times 10^{-2}$
630	5.694	3.116	2.094	1.604	1.569	1.088	1.070	$\times 10^{-2}$
840	9.602	4.836	3.455	2.769	2.101	2.094	1.433	$\times 10^{-2}$
n	1×3	2×3	3×3	4×3	5×3	6×3	7×3	
210	0.577	0.391	0.223	0.213	0.213	0.214	0.206	$\times 10^{-2}$
420	1.792	1.088	0.733	0.726	0.391	0.405	0.382	$\times 10^{-2}$
630	3.690	2.128	1.592	1.088	1.085	1.080	0.583	$\times 10^{-2}$
840	6.919	3.487	2.791	2.139	1.440	1.439	1.448	$\times 10^{-2}$
n	1×4	2×4	3×4	4×4	5×4	6×4	7×4	
210	0.453	0.272	0.2171	0.208	0.209	0.208	0.201	$\times 10^{-2}$
420	1.741	0.851	0.837	0.503	0.395	0.395	0.385	$\times 10^{-2}$
630	3.896	1.921	1.249	1.267	1.246	0.747	0.600	$\times 10^{-2}$
840	6.276	3.416	2.544	1.665	1.666	1.664	1.008	$\times 10^{-2}$

Table 3.2: The total computing time for the *block-saxpy* algorithm (in seconds) with $n_1 = n_2 = n$.

n	$n_b \times n_p$							
n	1×1	2×1	3×1	4×1	5×1	6×1	7×1	
210	1.215	0.699	0.535	0.368	0.368	0.365	0.204	$\times 10^{-2}$
420	4.667	2.354	1.692	1.363	1.035	1.044	0.707	$\times 10^{-2}$
630	10.434	5.481	3.521	3.013	2.516	2.022	1.537	$\times 10^{-2}$
840	18.469	9.255	6.647	4.658	4.006	3.347	2.694	$\times 10^{-2}$
n	1×2	2×2	3×2	4×2	5×2	6×2	7×2	
210	0.843	0.458	0.417	0.251	0.249	0.260	0.211	$\times 10^{-2}$
420	2.949	1.618	1.209	0.876	0.802	0.808	0.487	$\times 10^{-2}$
630	6.715	3.631	2.406	1.915	1.807	1.307	1.198	$\times 10^{-2}$
840	11.538	5.799	4.176	3.197	2.558	2.408	1.757	$\times 10^{-2}$
n	1×3	2×3	3×3	4×3	5×3	6×3	7×3	
210	0.723	0.466	0.297	0.257	0.254	0.252	0.217	$\times 10^{-2}$
420	2.459	1.375	0.967	0.883	0.558	0.559	0.488	$\times 10^{-2}$
630	5.238	2.888	2.037	1.547	1.429	1.355	0.858	$\times 10^{-2}$
840	9.561	4.814	3.668	2.704	2.058	1.922	1.765	$\times 10^{-2}$
n	1×4	2×4	3×4	4×4	5×4	6×4	7×4	
210	0.614	0.357	0.305	0.260	0.258	0.259	0.219	$\times 10^{-2}$
420	2.278	1.164	0.995	0.665	0.568	0.568	0.491	$\times 10^{-2}$
630	5.011	2.594	1.719	1.593	1.483	0.988	0.865	$\times 10^{-2}$
840	8.339	4.453	3.312	2.277	2.139	2.028	1.402	$\times 10^{-2}$

Figure 3.1: The speed of the parallel block dot algorithm (matvec)

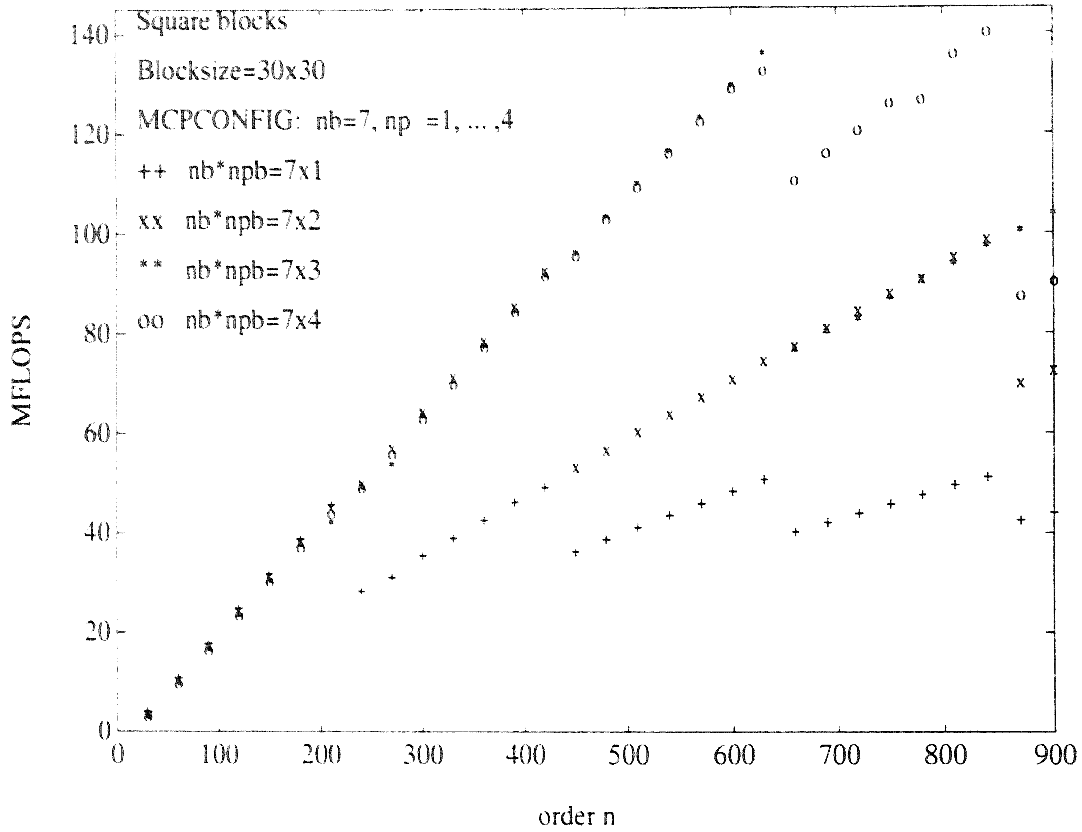


Figure 3.2: The speed of the parallel block dot algorithm (matvec)

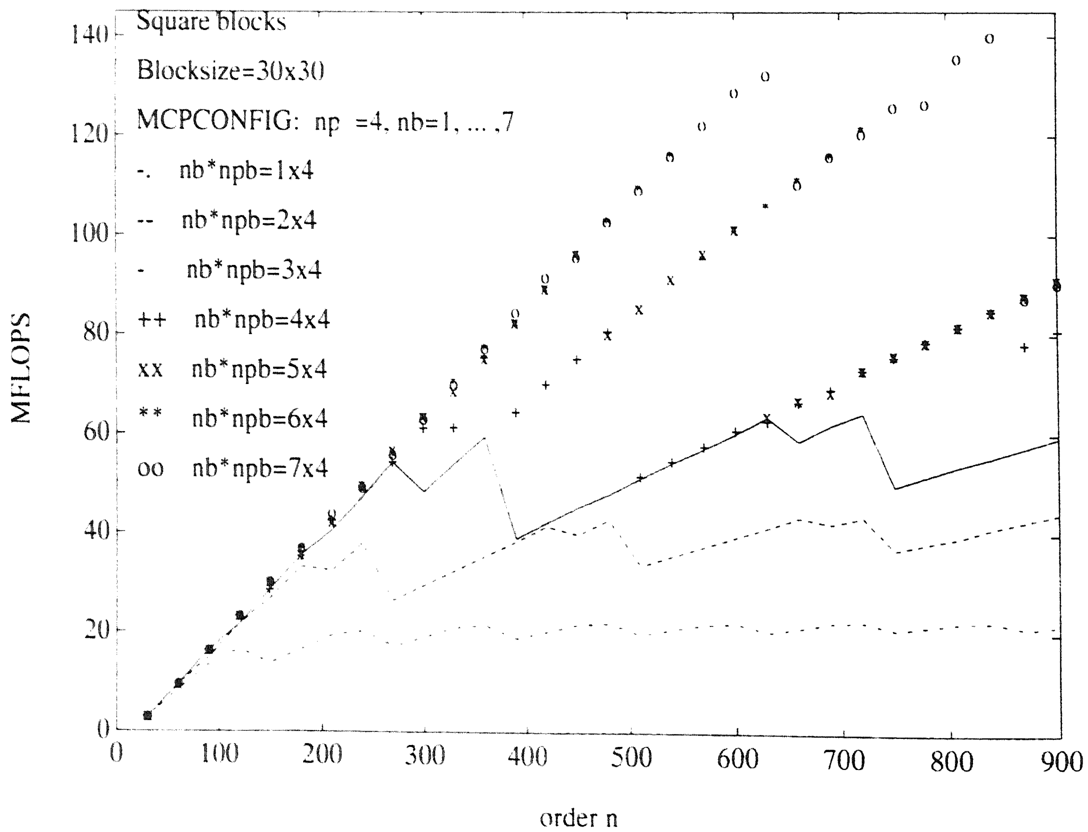


Figure 3.3: The speed of the parallel block saxpy algorithm (matvec)

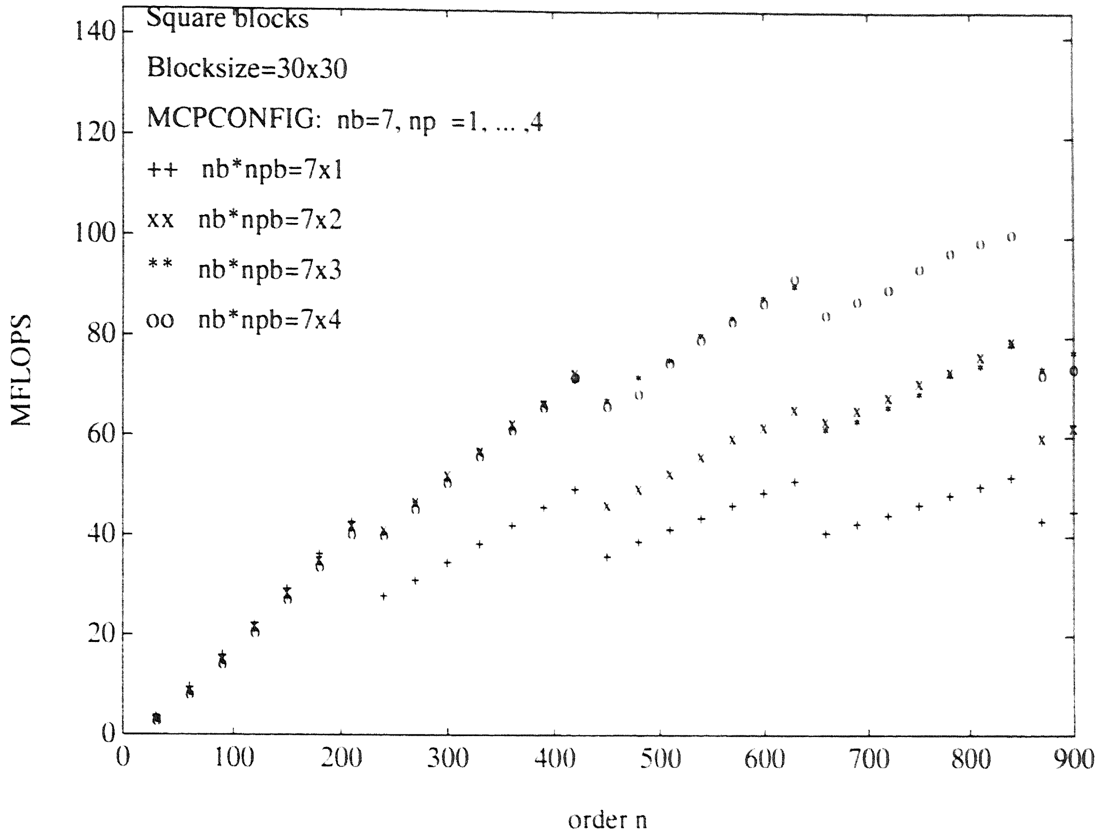
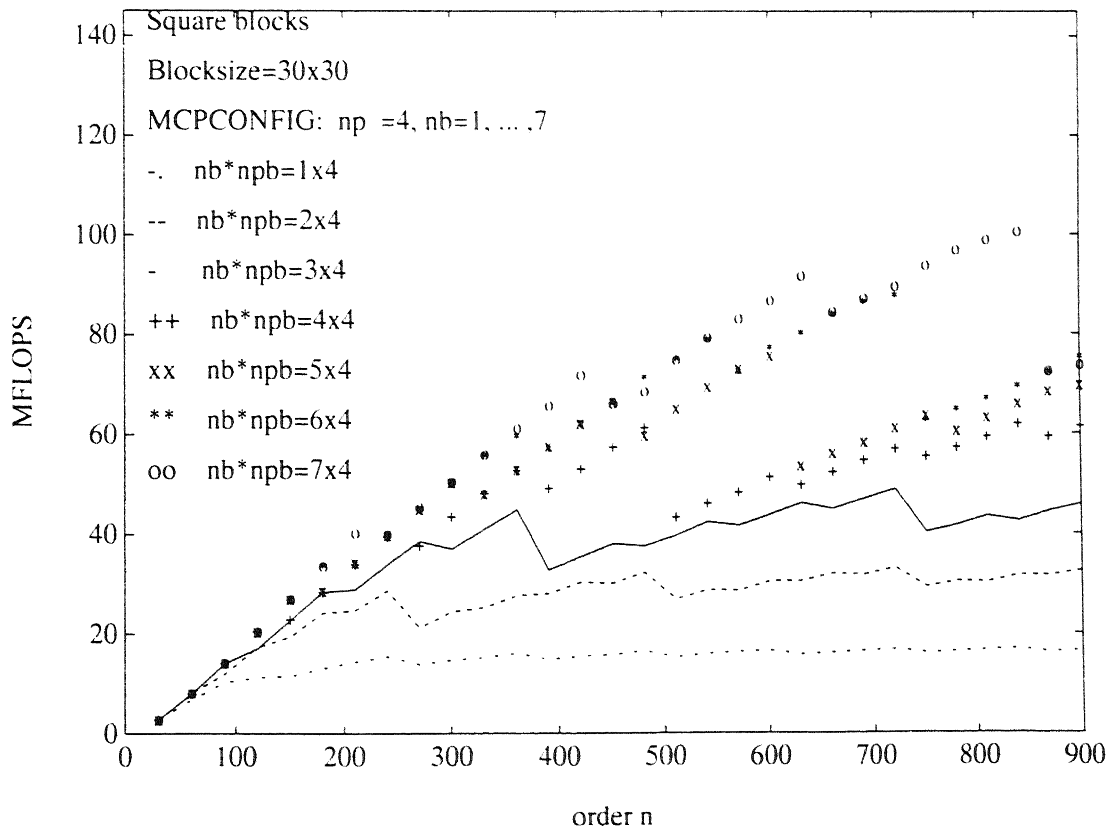


Figure 3.4: The speed of the parallel block saxpy algorithm (matvec)



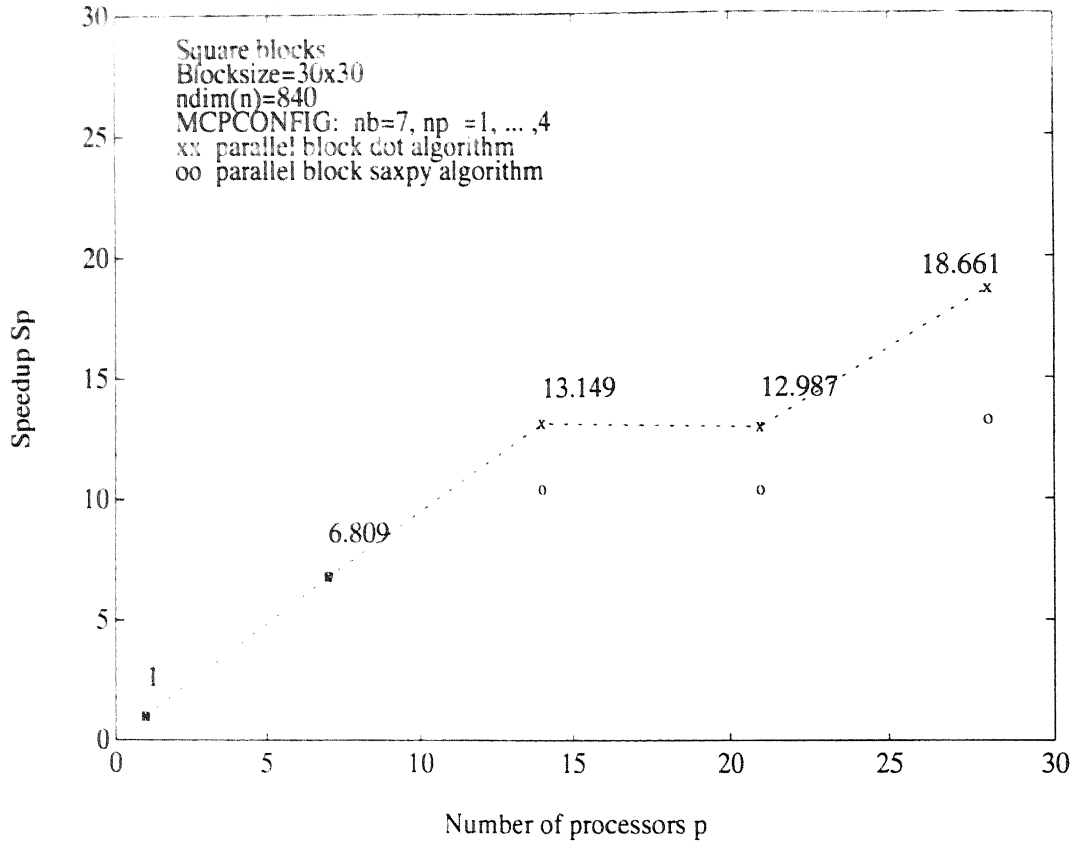


Figure 3.5: The speedups Sp of the parallel block algorithms

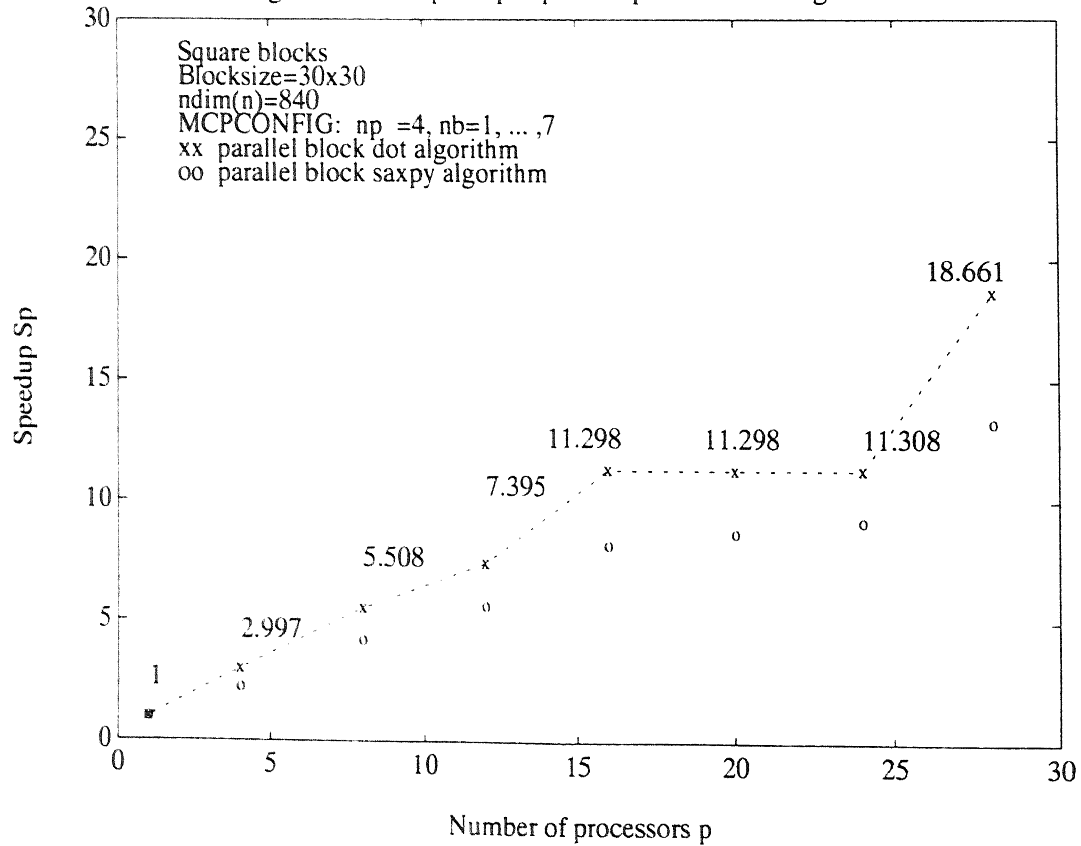


Figure 3.6: The speedups Sp of the parallel block algorithms

Figure 3.7: The speed of the parallel block dot algorithm

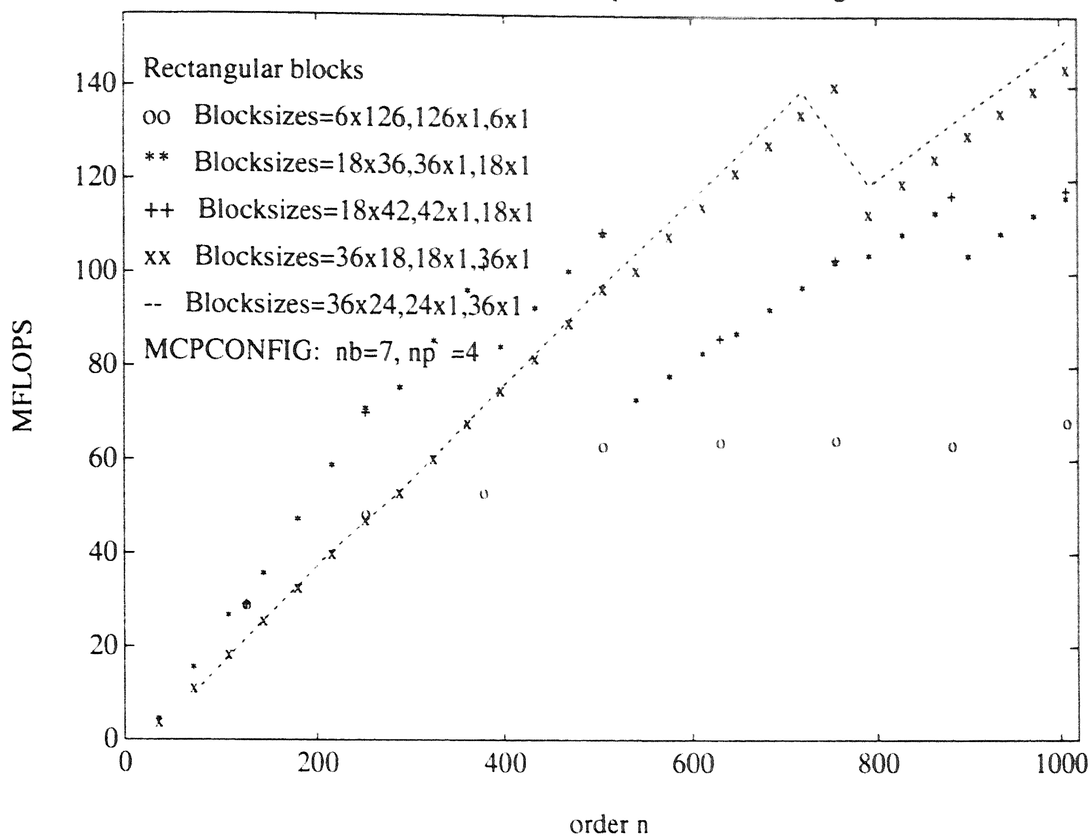


Figure 3.8: The speed of the parallel block saxpy algorithm

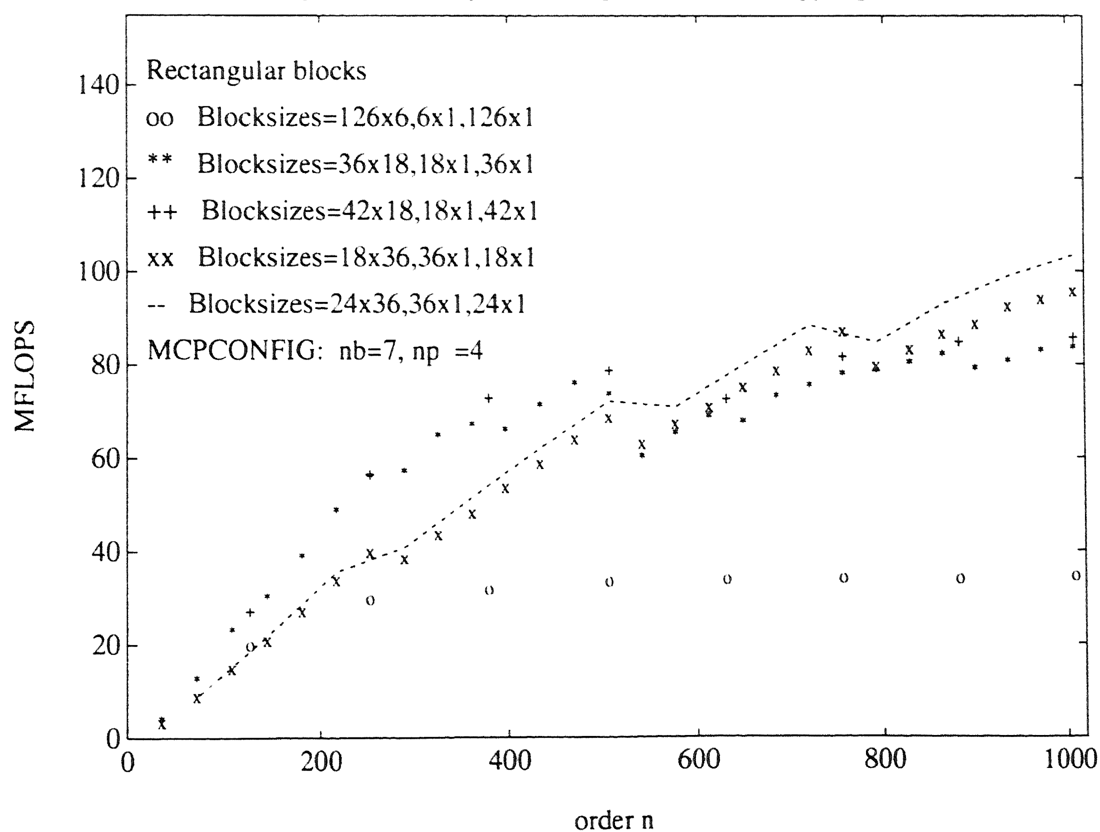


Figure 3.9: **matvec3**

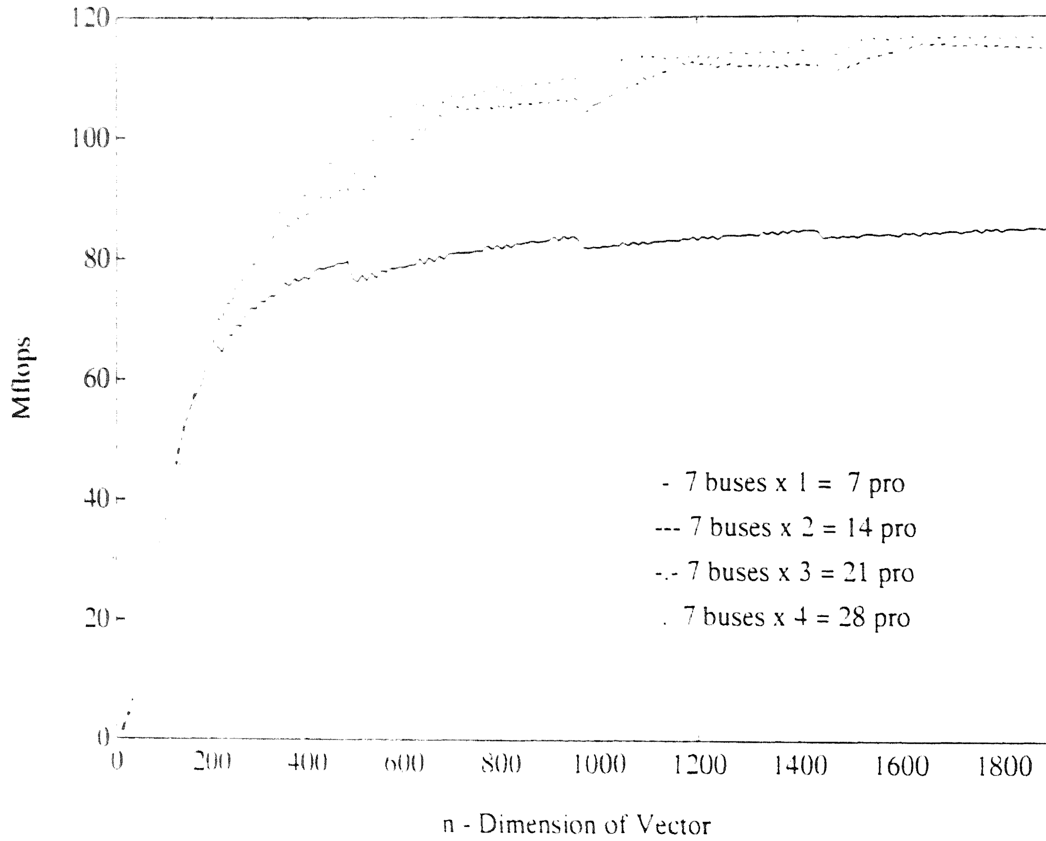
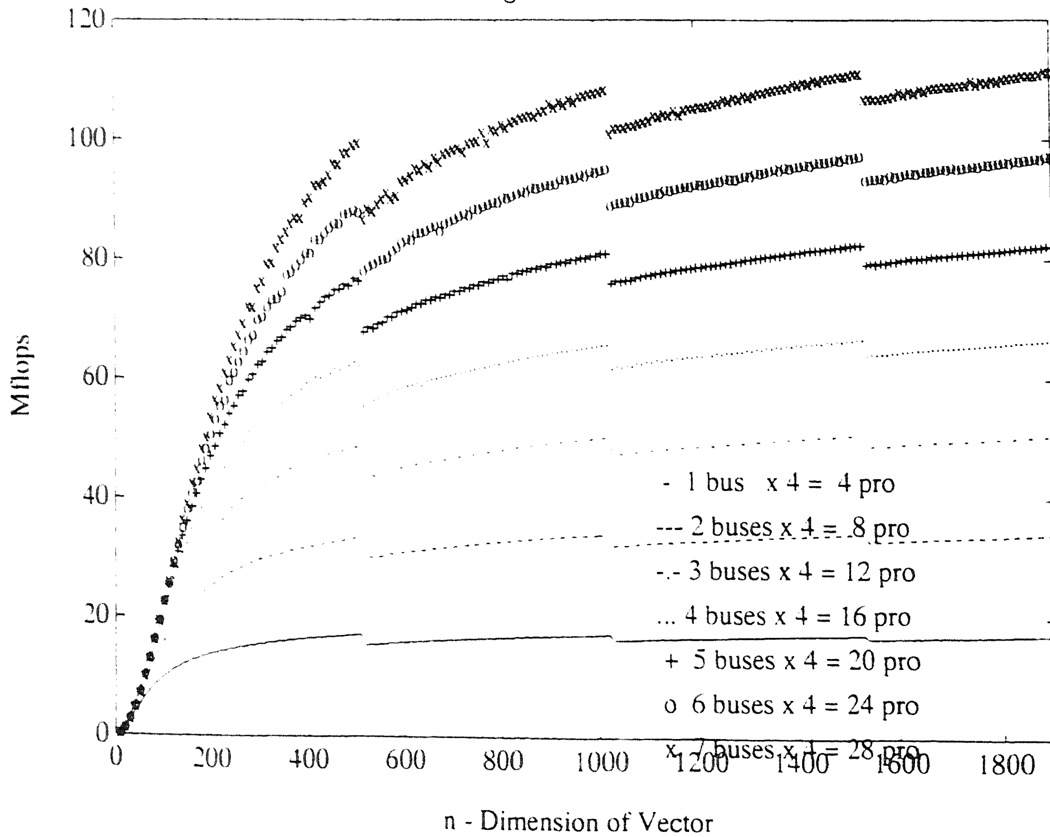


Figure 3.10: **matvec3**.



4 Some experimental results for the matrix-matrix product

In this section we discuss three different implementations for the *matrix-matrix multiplication*. We assume that the matrices A,B and C are of sizes $n_1 \times n_2$, $n_2 \times n_3$, $n_1 \times n_3$ and are partitioned in blocks of size $\alpha_1 \times \alpha_2$, $\alpha_2 \times \alpha_3$, $\alpha_1 \times \alpha_3$ respectively and we assume that $n_i = \alpha_i N_i$. Consider the *matrix-matrix product* $C = AB$, where $A = (A_{ik})$, $B = (B_{kj})$, $C = (C_{ij})$ and $C_{ij} = \sum_{k=1}^{N_2} A_{ik}B_{kj}$, $i = 1, ..., N_1$; $j = 1, ..., N_3$. We also assume that α_1, α_2, α_3 are chosen so that one block from A, one block from B, and one block from C can be stored completely in the cache (each processing element has a data cache of 8 KBytes), i.e. $\alpha_1\alpha_2 + \alpha_2\alpha_3 + \alpha_1\alpha_3 \leq S^L$ where $S^L = 2048$ for single precision arithmetic and $S^L = 1024$ for double precision arithmetic. We proceed as follows (here A^L, B^L and C^L are the cache *work-space* arrays and the superscript L means local):

First implementation:
• *Block-dot algorithm: ijk-version*
(A_{ik}, B_{kj}, and C_{ij} are block matrices)

```
cpcf   parallel
cpcf   pdo
     For i=1:N_1
        For j=1:N_3
           C^L(1 : α_1, 1 : α_3) = 0.0d0
           For k=1:N_2
cpcf   critical section bus
                 load A_ik and B_kj into the cache arrays A^L and B^L, respectively.
cpcf   end critical section
                 C^L = C^L + A^L * B^L
           End
cpcf   critical section bus
           store array C^L = C_ij into the main memory.
cpcf   end critical section
        End
     End
cpcf   end parallel
```

The algorithm operates on blocks. The matrices are stored columwise (ordinary Fortran way) in *two-dimensional* arrays. In the innermost *k-loop*, the array C^L is kept in cache. The algorithm is parallelized over the *i-loop* in such a way that different processors will treat different iterations of the loop. So at a given time, each processor is computing independently on different rows of blocks C_{ij}, $j = 1, ..., N_3$, of C (see appendix B, locality example 1 *matrix-matrix multiplication*). The computation of each block C_{ij} of C requires

a row of blocks A_{ik}, $k = 1, ..., N_2$ and a column of blocks B_{kj}, $k = 1, ..., N_2$. If we compute the amount of traffic between main memory and cache, we find that the matrix A is read N_3 times and the matrix B is read N_1 times and the matrix C is stored once, so that the total number T_d of reads and stores in the *block-dot* algorithm is given by

$$(7) \qquad T_d = N_1 N_3 (N_2(\alpha_1\alpha_2 + \alpha_2\alpha_3) + \alpha_1\alpha_3) = n_1 n_3 + n_1 n_2 n_3 (\frac{1}{\alpha_3} + \frac{1}{\alpha_1}).$$

The total number of operations (additions and multiplications) in the *block-dot* algorithm is $2n_1n_2n_3$. Further, we notice that T_d is independent of α_2 (for fixed n_1, n_2 and n_3).

Second implementation:
• *Block-saxpy algorithm: jki-version*
(A_{ik}, B_{kj}, and C_{ij} are block matrices)

```
        C(1:n1,1:n3) = 0.0d0
cpcf    parallel
cpcf    pdo
        For j=1:N3
          For k=1:N2
cpcf    critical section bus
              load B_kj into the cache array B^L.
cpcf    end critical section
              For i=1:N1
cpcf    critical section bus
                 load A_ik and C_ij into the cache arrays A^L and C^L, respectively.
cpcf    end critical section
                 C^L = C^L + A^L * B^L
cpcf    critical section bus
                 store array C^L=C_ij into the main memory.
cpcf    end critical section
              End
          End
        End
cpcf    end parallel
```

In the inner loop over k, the block B_{kj} is kept in the cache. The algorithm is parallelized over the *j-loop*. So at a given time, each processor is computing independently on different columns of blocks C_{ij}, $i = 1, ..., N_1$, of C. We have regrouped the load and store instructions inside the *i-loop* of the algorithm in one **critical section bus**. This eliminates the overhead caused by having more than one **critical section bus** in the *i-loop* (see appendix B, locality example 2 *matrix-matrix multiplication*). In a similar way, as in the previous algorithm, the total number T_s of reads and stores between the main memory and the

cache in the *block-saxpy* algorithm is given by

$$T_s = N_3N_2(N_1(\alpha_1\alpha_2 + 2\alpha_1\alpha_3) + \alpha_2\alpha_3) = n_2n_3 + n_1n_2n_3(\frac{1}{\alpha_3} + \frac{2}{\alpha_2}). \tag{8}$$

The total number of operations in this algorithm is $2n_1n_2n_3$, and T_s is independent of α_2 for fixed n_1, n_2 and n_3. The matrix A is read N_3 times, the matrix B is read once and the matrix C is loaded N_2 times and stored N_2 times.

Remark:
The choice made in the earlier described algorithms on the loop parallelized is somewhat arbitrary because the three loops are entirely interchangeable and offer similar opportunities for parallelization.

We have carried out various experiments in double precision with the *block-dot* and *block-saxpy* algorithms. First, we took $\alpha_1 = \alpha_2 = \alpha_3 = 18$, and $n_1 = n_2 = n_3 = n$. We measured the MFLOP-rates of our implementations for $n = i * 18$, $i = 1, ..., 30$, on the following configurations of the matrix coprocessor: $p = n_b \times n_p$ processors, for $n_b = 7$, $n_p = 1, ..., 4$, and for $n_p = 4$, $n_b = 1,..., 7$. The results are displayed in figures 4.1-4.4. We see for example that the maximal performance is reached when $n = 28 \times 18 = 504$, and for both algorithms, the performance increases with the number of processors in use.

We measured the total computing time varying n_b and keeping fixed n_p and the total computing varying n_p and keeping fixed n_b. The results are given in tables 4.1 and 4.2. For a matrix of size $n = 504$, we have computed the speedups (with respect to the wall clock time, on one processor) for the various configurations given above. There are given in figures 4.5-4.6. Since the block size is 18×18, there are $504/18 = 28$ iterations of the outermost loop of both algorithms, so we have 28 independent tasks for the available number p of processors. This means that if 28 is divisible by p, we expect a speedup by a factor of about p. Otherwise, this speedup factor will be smaller. The results in figures 4.5 and 4.6 confirm this.

So far we have considered (square) matrices of order that are multiples of the (square) block size. If now the block matrices have different sizes $\alpha_1 \times \alpha_2$, $\alpha_2 \times \alpha_3$ and $\alpha_1 \times \alpha_3$ respectively, then the computing time C_d, the communication time M_d, and their quotient for the *block-dot* algorithm can be estimated by

$$C_d \approx \lceil N_1/p \rceil 2\alpha_1 n_2 n_3 / R \tag{9}$$

$$M_d \approx \lceil N_1/n_b \rceil \{n_2n_3(1 + \frac{\alpha_1}{\alpha_3}) + \alpha_1 n_3\}/r \tag{10}$$

$$\frac{M_d}{C_d} \approx \frac{\lceil N_1/n_b \rceil}{\lceil N_1/p \rceil}\frac{1}{2}\{\frac{1}{\alpha_3} + \frac{1}{\alpha_1} + \frac{1}{n_2}\}R/r \tag{11}$$

For the *block-saxpy* algorithm, the computing time C_s, the communication time M_s, and their quotient can be estimated by

$$C_s \approx \lceil N_3/p \rceil 2\alpha_3 n_1 n_2 / R \tag{12}$$

$$M_s \approx \lceil N_3/n_b \rceil \{n_1 n_2 (1 + 2\frac{\alpha_3}{\alpha_2}) + \alpha_3 n_2\}/r \tag{13}$$

$$\frac{M_s}{C_s} \approx \frac{\lceil N_3/n_b \rceil}{\lceil N_3/p \rceil} \frac{1}{2} \{\frac{1}{\alpha_3} + \frac{2}{\alpha_2} + \frac{1}{n_1}\} R/r \tag{14}$$

Similar to the *matrix-vector case* discussed in Section 3, we see that the computing time can be decreased by increasing n_b or n_p, but that the communication time can be decreased only by increasing n_b (and not by increasing n_p).
From (11), if α_1 and α_3 increase for fixed order of the matrices A and B then the ratio $\frac{M_d}{C_d}$ decreases. Similarly, from (14) if α_2 and α_3 increase for fixed order of the matrices A and B then the ratio $\frac{M_s}{C_s}$ decreases and this leads of course to a better performance.
We have performed tests to demonstrate these properties, which are discussed more in details in Section 5. So we measured the total computing time and the *MFLOP-rates* of our implementations, varying the processor's cache block sizes.

For the first implementation, we took square blocks of size $\alpha_1 = 18$, $\alpha_2 = 18$, $\alpha_3 = 18$ and rectangular blocks of size $\alpha_1 = 18$, $\alpha_2 = 36$, $\alpha_3 = 6$; $\alpha_1 = 18$, $\alpha_2 = 6$, $\alpha_3 = 36$; $\alpha_1 = 6$, $\alpha_2 = 36$, $\alpha_3 = 18$ and $\alpha_1 = 6$, $\alpha_2 = 18$, $\alpha_3 = 36$ respectively, for $n_1 = n_2 = n_3 = n$, where $n = i * 36$, for $i = 1$,..., 15. For the second implementation, we took the same square blocks and rectangular blocks of size $\alpha_1 = 18$, $\alpha_2 = 36$, $\alpha_3 = 6$; $\alpha_1 = 6$, $\alpha_2 = 36$, $\alpha_3 = 18$; $\alpha_1 = 6$, $\alpha_2 = 18$, $\alpha_3 = 36$; $\alpha_1 = 36$, $\alpha_2 = 18$, $\alpha_3 = 6$ respectively. For both algorithms, we fixed the configuration of the matrix coprocessor at $p = 28$ processors, i.e, $n_b = 7$, and $n_p = 4$. The results are displayed in figures 4.7 and 4.8.

Third implementation:
The algorithm used here, is implemented in terms of a call to level 3 BLAS on the matrix coprocessor. The level 3 BLAS incorporates *matrix-matrix operations*. The level 3 BLAS used here is: **RGMMUL** for multiplying two matrices (see Appendix B, locality example 3 *matrix-matrix multiplication*). We measured the *MFLOP-rates* of this implementation for $n = i * 100$, $i = 1$, ..., 10, on the following configurations of the matrix coprocessor: for $n_b = 7$, $n_p = 1$, ..., 4, and for $n_p = 4$, $n_b = 1$, ..., 7. The results are displayed in figures 4.9 and 4.10. The maximal performance is reached round $n = 500$, and increases with the number of processors in use.

Table 4.1: The total computing time for the *block-dot* algorithm (in seconds) with $n_1 = n_2 = n_3 = n$.

n	$n_b \times n_p$						
n	1×1	2×1	3×1	4×1	5×1	6×1	7×1
126	0.439	0.251	0.188	0.125	0.125	0.125	0.0632
252	3.501	1.750	1.250	1.001	0.750	0.750	0.500
378	11.799	6.181	3.933	3.372	2.810	2.248	1.686
504	27.946	13.973	9.982	6.988	5.989	4.991	3.993
n	1×2	2×2	3×2	4×2	5×2	6×2	7×2
126	0.253	0.127	0.126	0.064	0.064	0.064	0.062
252	1.767	1.008	0.755	0.505	0.503	0.503	0.253
378	6.228	3.396	2.262	1.700	1.696	1.134	1.129
504	14.085	7.044	5.031	4.018	3.019	3.011	2.014
n	1×3	2×3	3×3	4×3	5×3	6×3	7×3
126	0.191	0.127	0.064	0.064	0.064	0.064	0.063
252	1.269	0.759	0.505	0.504	0.254	0.254	0.253
378	3.987	2.276	1.702	1.140	1.137	1.132	0.571
504	10.096	5.051	4.032	3.021	2.023	2.018	2.010
n	1×4	2×4	3×4	4×4	5×4	6×4	7×4
126	0.129	0.065	0.064	0.064	0.064	0.064	0.063
252	1.020	0.510	0.506	0.256	0.254	0.254	0.253
378	3.424	1.714	1.142	1.140	1.136	0.572	0.570
504	7.104	4.051	3.037	2.026	2.022	2.014	1.016

Table 4.2: The total computing time for the *block-saxpy* algorithm (in seconds) with $n_1 = n_2 = n_3 = n$.

n	$n_b \times n_p$						
n	1×1	2×1	3×1	4×1	5×1	6×1	7×1
126	0.443	0.253	0.189	0.126	0.126	0.126	0.063
252	3.525	1.763	1.259	1.007	0.756	0.755	0.504
378	11.882	6.224	3.962	3.395	2.829	2.264	1.698
504	28.143	14.072	10.053	7.036	6.032	5.027	4.021
n	1×2	2×2	3×2	4×2	5×2	6×2	7×2
126	0.258	0.129	0.128	0.0653	0.065	0.065	0.063
252	1.792	1.020	0.764	0.512	0.509	0.508	0.257
378	6.300	3.434	2.288	1.721	1.714	1.147	1.141
504	14.239	7.121	5.089	4.060	3.054	3.040	2.038
n	1×3	2×3	3×3	4×3	5×3	6×3	7×3
126	0.195	0.129	0.066	0.065	0.065	0.065	0.064
252	1.291	0.769	0.516	0.511	0.259	0.259	0.257
378	4.046	2.308	1.723	1.156	1.151	1.146	0.579
504	10.223	5.113	4.078	3.054	2.048	2.044	2.031
n	1×4	2×4	3×4	4×4	5×4	6×4	7×4
126	0.133	0.067	0.066	0.065	0.065	0.065	0.063
252	1.040	0.520	0.514	0.262	0.259	0.259	0.253
378	3.478	1.743	1.163	1.155	1.149	0.583	0.570
504	7.207	4.107	3.078	2.055	2.048	2.038	1.016

So far we have studied different block algorithm techniques applied to the simple *matrix-vector* and *matrix-matrix* operations. If now we examine the behavior of the numerical results of our implementations, which are given in the figures 3.1-3.6 and 4.1-4.6, it can be seen that the total computing time follows the same pattern in all the given figures. For the *matrix-matrix product* the measured *MFLOP-rates* for the *block-dot* (resp. *block-saxpy*) algorithm are linear with the order of the matrices if the processor configuration is used once i.e., $N_1/p \leq 1$ (respectively $N_3/p \leq 1$). If $N_1/p > 1$ (respectively $N_3/p > 1$) then the *MFLOP-rates* become *non-linear* as a function of the order n and drop after each use of the processor configuration, and this happens $\lfloor N_1/p \rfloor$ times (respectively $\lfloor N_3/p \rfloor$ times).
For fixed n_1, n_2 and n_3, we generally expect for all algorithms that the total computing time decreases as n_b increases for fixed value of n_p and decreases slightly as n_p increases for fixed value of n_b. The information of the following tables support this. The tables 3.1, 3.2, 4.1 and 4.2 show the computing time obtained by varying n_b and keeping fixed n_p and the total computing time obtained by varying n_p and keeping fixed n_b. For the tables 3.1 and 3.2, we took $n = 30 * i$, $i = 7$, 14, 21, 28 and blocks of size 30×30. For the tables 4.1

and 4.2, we took $n = 18 * i$, $i = 7$, 14, 21, 28 and blocks of size 18×18.

It is observed in view of the given tables that the total computing time on the matrix coprocessor configuration with $n_b = a$ and $n_p = b$ where $a > b$ is slightly less than that on a configuration with $n_b = b$ and $n_p = a$ (for a given order n). This again illustrates the communication problem if more than one processors are configured on the same bus. We conclude that we have to choose the value of n_b as close as possible to the maximum of the matrix coprocessor bus configuration in order to minimize the total computing time.

Figure 4.1: The speed of the parallel block dot algorithm

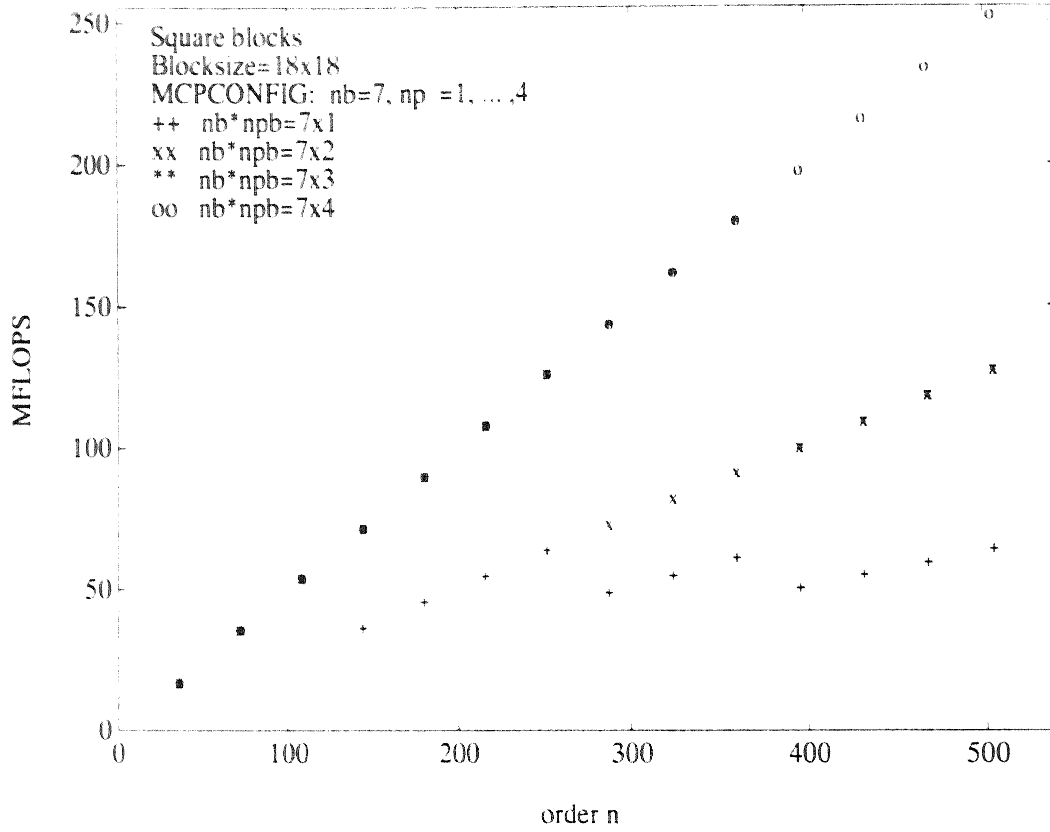


Figure 4.2: The speed of the parallel block dot algorithm

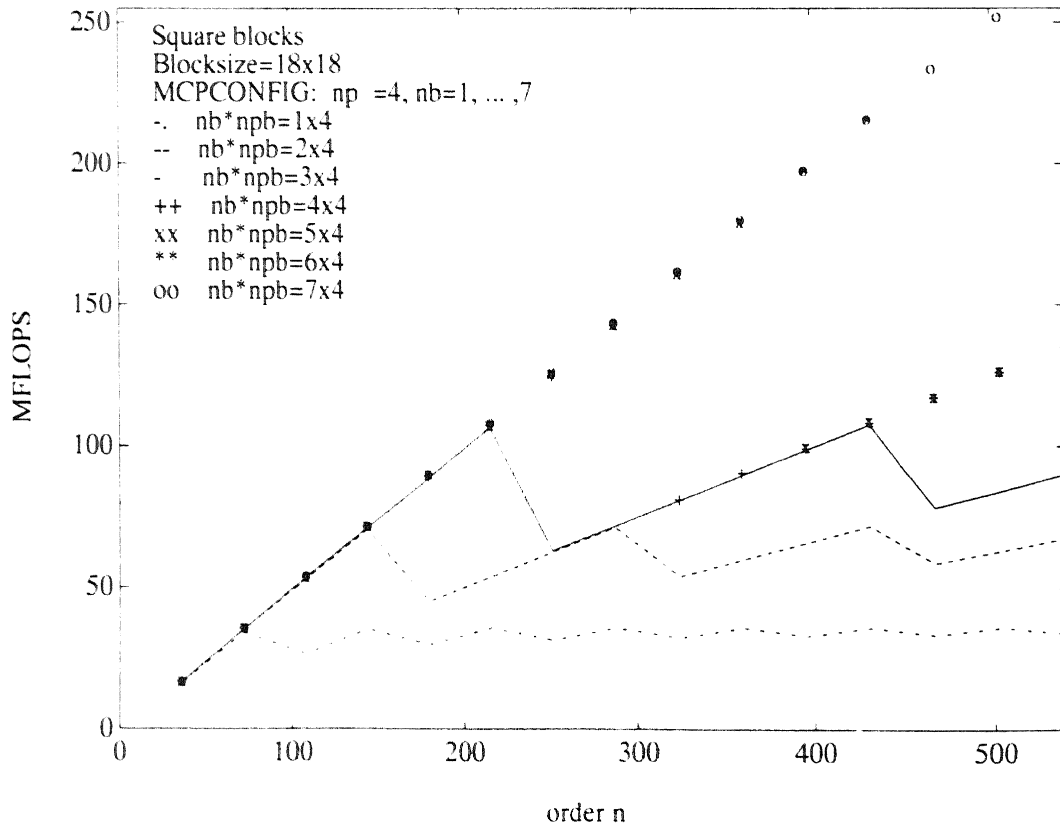


Figure 4.3: The speed of the parallel block saxpy algorithm

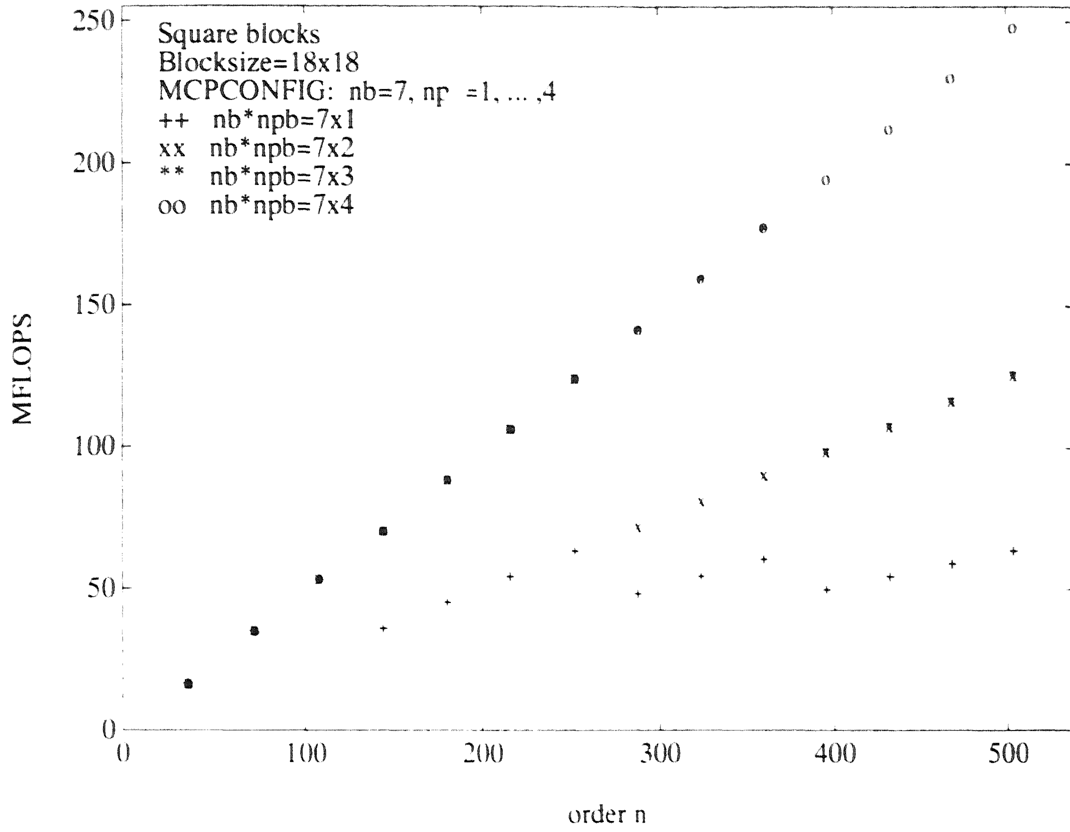
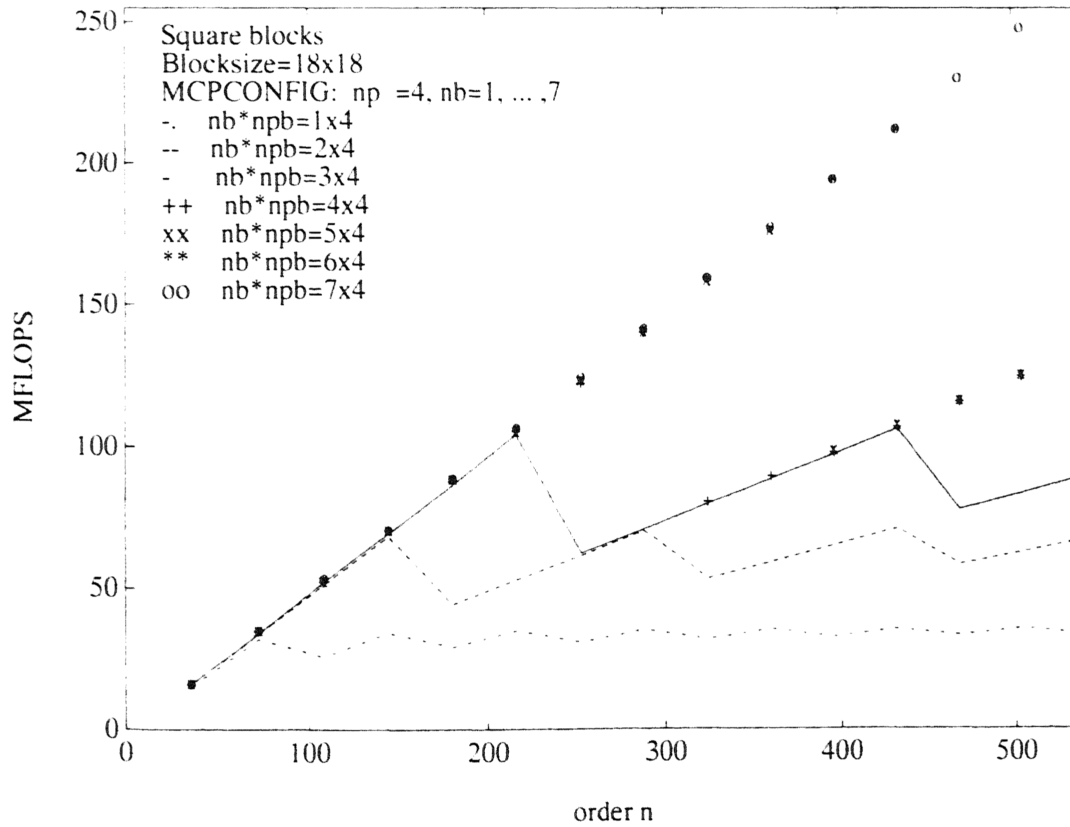


Figure 4.4: The speed of the parallel block saxpy algorithm



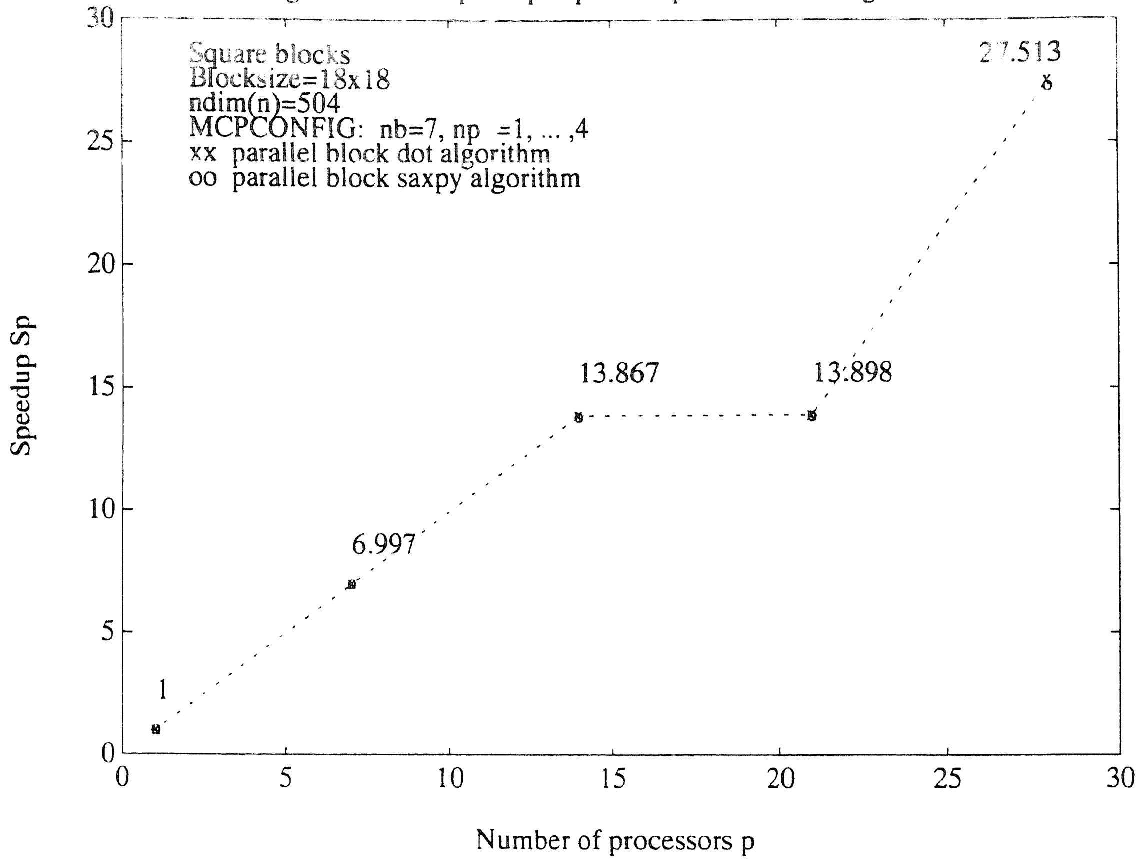


Figure 4.5: The speedups Sp of the parallel block algorithms
Square blocks
Blocksize=18x18
ndim(n)=504
MCPCONFIG: nb=7, np =1, ... ,4
xx parallel block dot algorithm
oo parallel block saxpy algorithm
27.513
13.867
13.898
6.997
1
Speedup Sp
Number of processors p
0
5
10
15
20
25
30

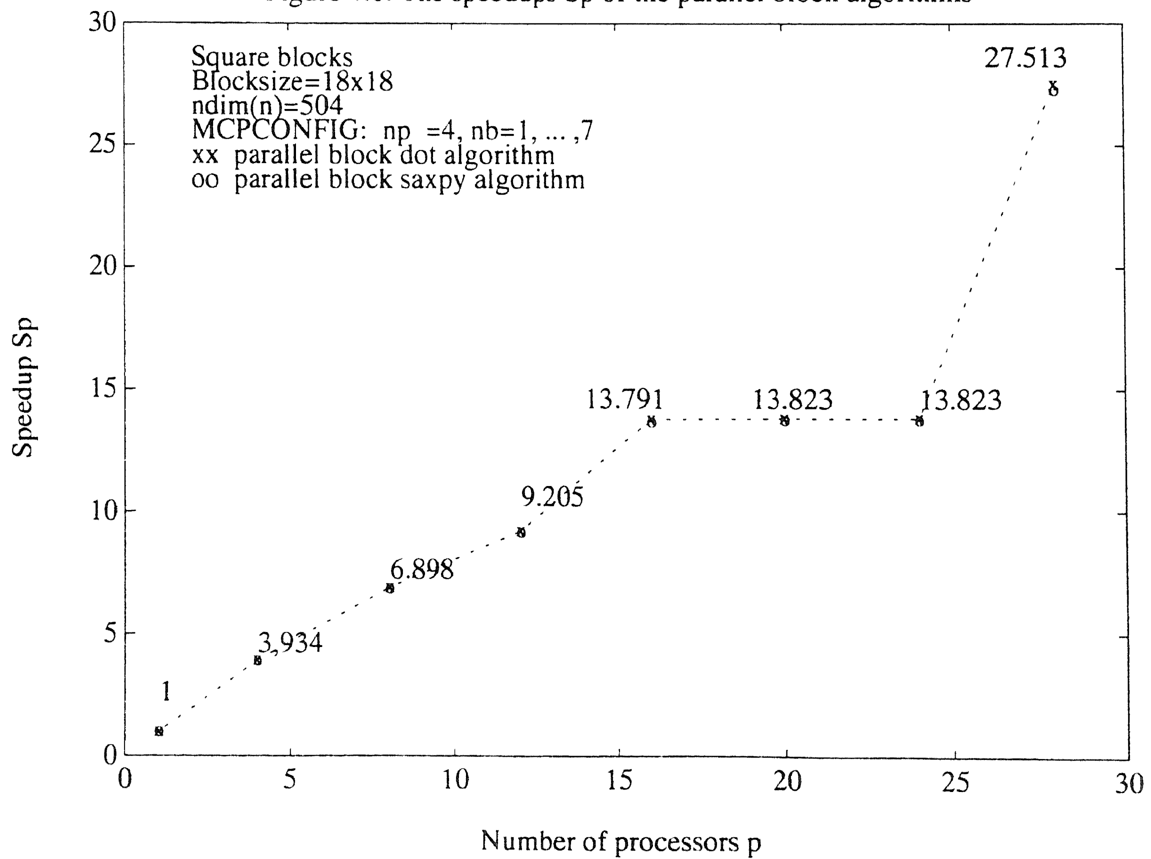


Figure 4.6: The speedups Sp of the parallel block algorithms
Square blocks
Blocksize=18x18
ndim(n)=504
MCPCONFIG: np =4, nb=1, ... ,7
xx parallel block dot algorithm
oo parallel block saxpy algorithm
27.513
13.791
13.823
13.823
9.205
6.898
3,934
1
Speedup Sp
Number of processors p
0
5
10
15
20
25
30

Figure 4.7: The speed of the parallel block dot algorithm

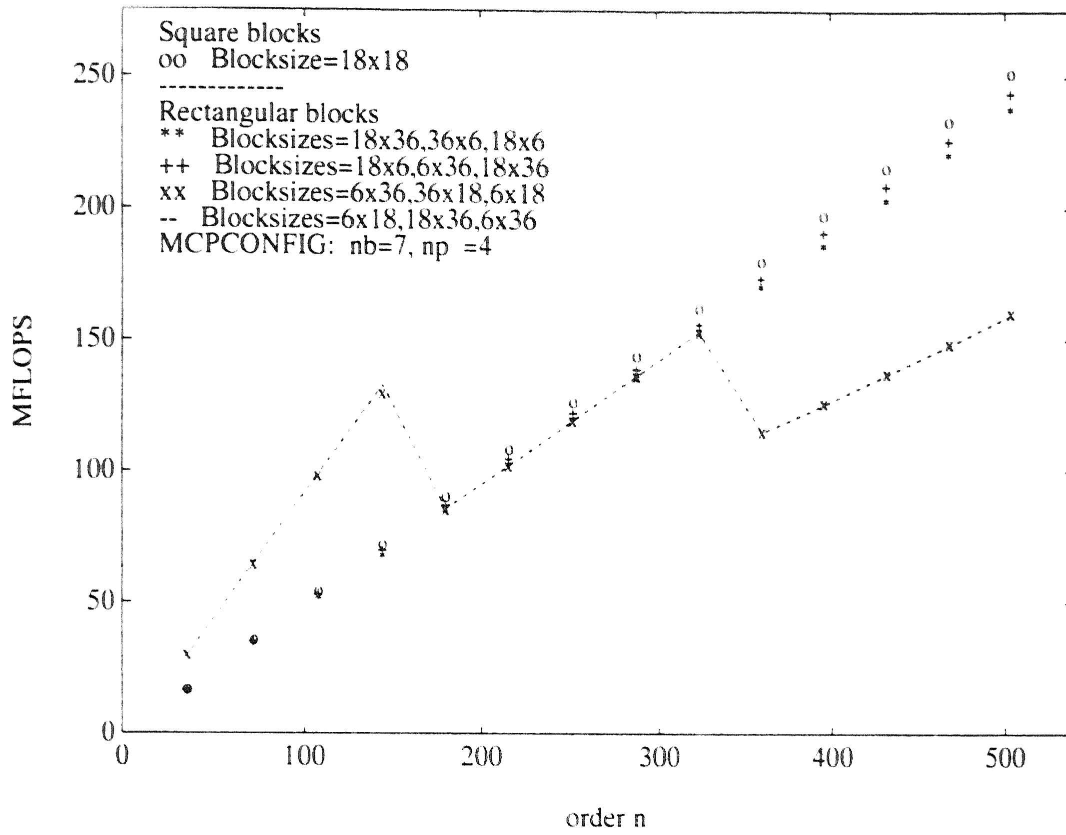
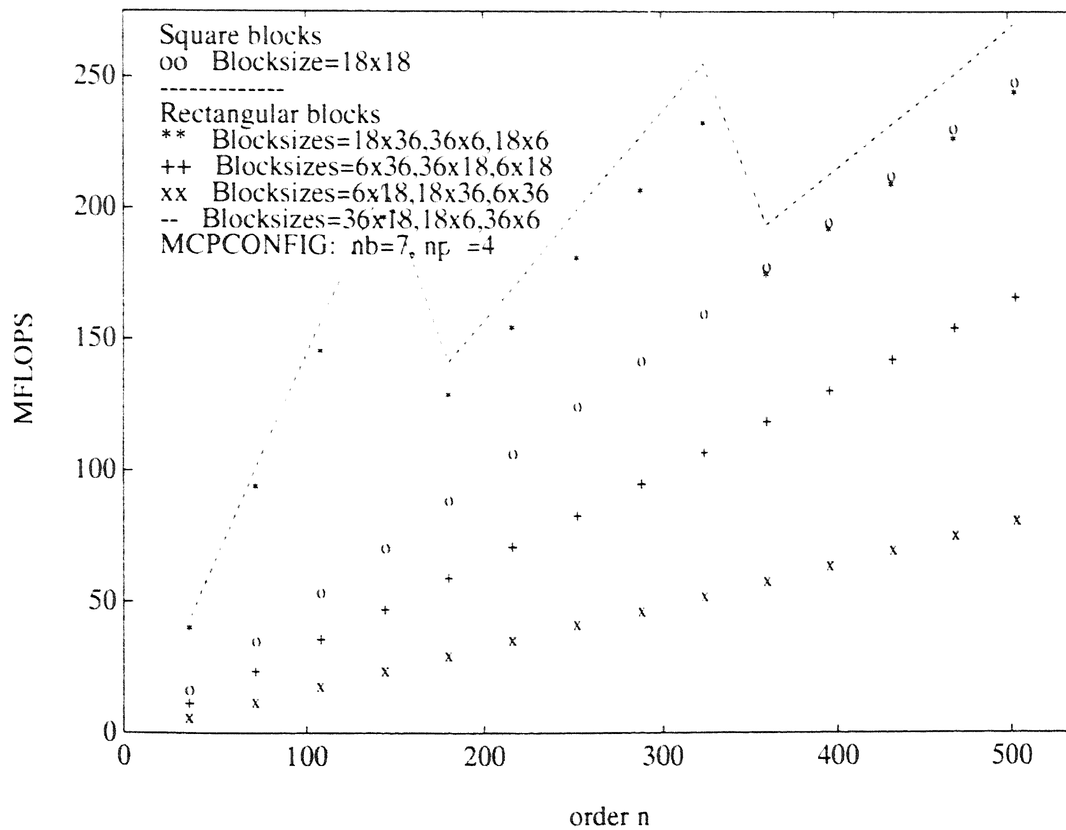


Figure 4.8: The speed of the parallel block saxpy algorithm



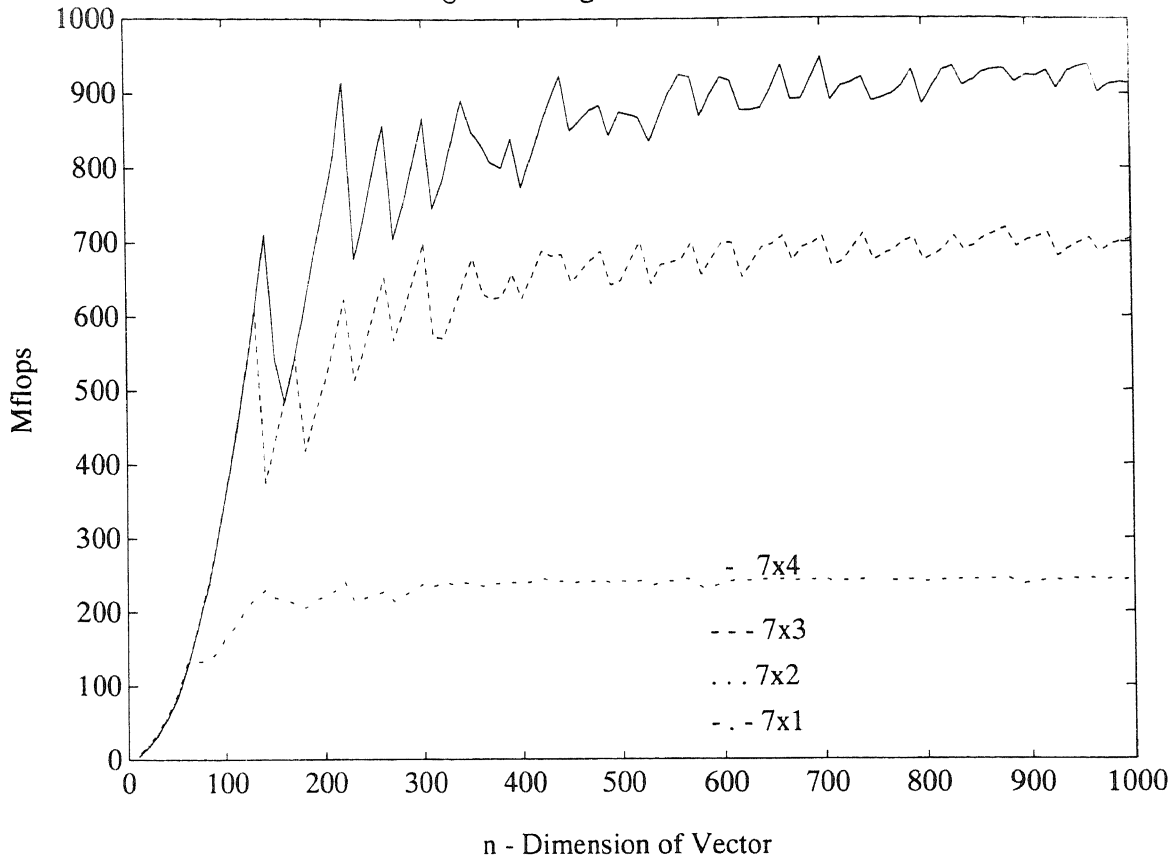


Figure 4.9: rgmmul
Mflops
n - Dimension of Vector
0
100
200
300
400
500
600
700
800
900
1000
- 7x4
- - - 7x3
. . . 7x2
- . - 7x1

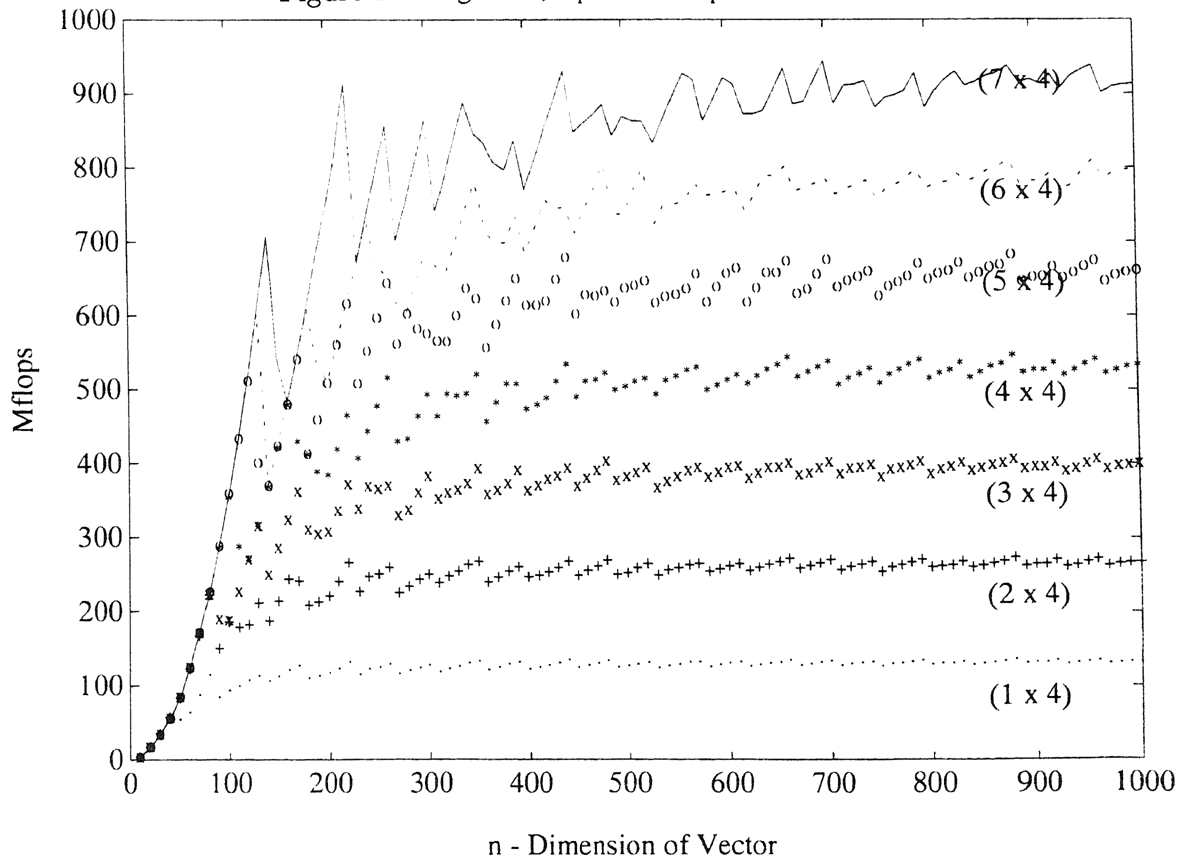


Figure 4.10: rgmmul, 4 processors per bus
Mflops
n - Dimension of Vector
0
100
200
300
400
500
600
700
800
900
1000
(7 x 4)
(6 x 4)
(5 x 4)
(4 x 4)
(3 x 4)
(2 x 4)
(1 x 4)

5 A performance model for matrix-vector and matrix-matrix product

In this section we model the total cost of communication, i.e. the cost of sending or receiving messages (like load, store instructions) and the time spent in computing. We assume that the sending or receiving of n floating point numbers between one processor and the global memory takes $\frac{n}{r}$ seconds, where r is the rate by which a message can be transferred, expressed in words per second.
We assume that a reasonable estimate of the time T_{pa} required to execute a program of *N-equal processes* on a shared parallel MIMD computer of matrix configuration $p = n_b \times n_p$ processors (if we assume that the algorithm uses only the pfp directives **cpcf critical section bus**) is as follows:

$$T_{pa} \approx \lceil \frac{N}{p} \rceil T_{comp} + \lceil \frac{N}{n_b} \rceil T_{comm} \quad seconds, \tag{15}$$

where, T_{comp} and T_{comm} are respectively the computing time and the communication overhead required to execute one process on **one processor**. We neglect the idle time of a processing element waiting when another one is accessing the memory, since this very much depends on the ratio between T_{comp} and T_{comm}, and on the places in the algorithm where communication is performed. The *MFLOP-rate*, the *speedup* and the *efficiency* are defined, as usual, by

$$MFLOP - rate = \frac{the\ total\ number\ of\ operations}{T_{pa}} \ ; \tag{16}$$

the speed-up by

$$S_p = \frac{T_1}{T_{pa}} \ ; \tag{17}$$

where T_1 is the time on one processor, and the efficiency by

$$E_p = \frac{S_p}{p} \times 100\%. \tag{18}$$

Furthermore, since each matrix bus has a peak bandwidth of 160 MBytes per second, we have $r = 2 \times 10^7$ words (of 8 bytes) per second. At any time only one processing element along a matrix bus is allowed to access the shared storage area, and each processing element produces up to 40 MFLOPS of double precision multiply and add performance. So we have $R = 4 \times 10^7$ flops per second.

The performance model for matrix-vector product:
For each iteration step of the *k-loop* of the *block-dot algorithm*, one block A_{ik} from matrix A and one segment from vector b are sent to each processor; after k iterations one array c^L is sent to the main memory. Thus, we have $\alpha_1 * \alpha_2 + \alpha_2$ data transfers and $2\alpha_1 * \alpha_2$ flops

performed on different iterations k. If the computation proceeds at R flops per second and the communication proceeds at r words per seconds then the cost of each step of the *i-loop* of the *block- dot* algorithm requires a communication time ${T_{comm}}^d$ and a computing time ${T_{comp}}^d$ given by

$$(19) \qquad {T_{comm}}^d \approx N_2(\alpha_1\alpha_2 + \alpha_2)/r + \alpha_1/r \;\; and \;\; {T_{comp}}^d = 2\alpha_1\alpha_2 N_2/R.$$

For the *block-saxpy* algorithm, the cost of each step of the *k-loop* requires a communication time ${T_{comm}}^s$ and a computing time ${T_{comp}}^s$ given by

$$(20) \qquad {T_{comm}}^s \approx \alpha_2/r + N_1(\alpha_1\alpha_2 + 2\alpha_1)/r \;\; and \;\; {T_{comp}}^s = N_1(2\alpha_1\alpha_2 + \alpha_1)/R.$$

In particular, if $n_1 = n_2 = n$, and $\alpha_1 = \alpha_2 = \alpha$, with $n = \alpha N$, then each step of *i-loop* of the *block-dot* algorithm requires: $2N\alpha^2/R + \{N\alpha^2 + (N+1)\alpha\}/r$ *seconds* and each step of *j-loop* of the *block-saxpy* algorithm requires: $N(2\alpha^2+\alpha)/R + \{N\alpha^2 + (2N+1)\alpha\}/r$ *seconds.* Thus, the ratio of the computing time to the communication time on one processor for the *block-dot* algorithm is: $\frac{2}{\{1+(\frac{1}{\alpha}+\frac{1}{n})\}} r/R \approx 1$.
The ratio of the computing time to the communication time on one processor for the *block-saxpy* algorithm is: $\frac{2+\frac{1}{\alpha}}{\{1+(\frac{2}{\alpha}+\frac{1}{n})\}} r/R \approx 1$.
If now the computing time and the communication time are modeled as given above, then an estimate of the time ${T_1}^d$ (resp. ${T_1}^s$) required to execute the program on one processor and the time ${T_{pa}}^d$ (resp. ${T_{pa}}^s$) required to execute the program on p processors for the *block-dot* (resp. *block-saxpy*) algorithm looks as follows:

$$(21) \qquad {T_1}^d \approx 2n_1n_2/R + \{n_1 + n_1n_2(1 + \frac{1}{\alpha_1})\}/r$$

$$(22) \qquad {T_1}^s \approx n_1n_2(2 + \frac{1}{\alpha_2})/R + \{n_2 + n_1n_2(1 + \frac{2}{\alpha_2})\}/r$$

$$(23) \qquad {T_{pa}}^d \approx C_d + M_d \;\; and \;\; {T_{pa}}^s \approx C_s + M_s$$

where C_d, M_d, C_s and M_s are given in section 3, formulas (3)-(6). The *MFLOP-rates* are given by

$$(24) \qquad MFLOP_d \approx \frac{2n_1n_2}{{T_{pa}}^d} \;\; and \;\; MFLOP_s \approx \frac{2n_1n_2}{{T_{pa}}^s}$$

In tables 5.1 and 5.2 we present the model and observed values of T_1, T_{pa}, S_p and MFLOP, for the *block-dot* and the *block-saxpy* version of the *matrix-vector* algorithm, respectively.

Table 5.1: Theoretical and observed times (in seconds), speedups and *MFLOP-rates* for the *block-dot version* of the *matrix-vector* algorithm with $n_1 = n_2 = n = 1008$ and $p = 28$.

α_1, α_2	$\frac{(n/\alpha_1)}{p}$	C_d	M_d	$\frac{M_d}{C_d}$	$T_1{}^d$ *model*	$T_1{}^d$ *exp*	$T_{pa}{}^d$ *model*	$T_{pa}{}^d$ *exp*	$S_p{}^d$ *model*	$S_p{}^d$ *exp*	$MFLOP_d$ *model*	$MFLOP_d$ *exp*
6,126	6	0.181×10^{-2}	0.847×10^{-2}	4.670	0.110	0.486	1.028×10^{-2}	0.298×10^{-1}	10.700	16.308	197.510	068.160
18,36	2	0.181×10^{-2}	0.766×10^{-2}	4.226	0.104	0.311	0.948×10^{-2}	0.175×10^{-1}	10.970	17.771	214.307	116.140
18,42	2	0.181×10^{-2}	0.766×10^{-2}	4.226	0.104	0.309	0.948×10^{-2}	0.172×10^{-1}	10.970	17.965	214.307	117.820
36,18	1	0.181×10^{-2}	0.746×10^{-2}	4.115	0.103	0.265	0.928×10^{-2}	0.141×10^{-1}	11.099	18.794	218.962	143.800
36,24	1	0.181×10^{-2}	0.746×10^{-2}	4.115	0.103	0.261	0.928×10^{-2}	0.135×10^{-1}	11.099	19.333	218.962	150.070
42,18	$\frac{24}{28}$	0.211×10^{-2}	0.867×10^{-2}	4.099	0.1028	0.256	1.079×10^{-2}	0.153×10^{-1}	09.527	16.732	188.264	132.540
126,6	$\frac{8}{28}$	0.635×10^{-2}	1.281×10^{-2}	2.017	0.102	0.229	1.916×10^{-2}	0.294×10^{-1}	05.323	07.789	106.035	069.020

Table 5.2: Theoretical and observed times (in seconds), speedups and *MFLOP-rates* for the *block-saxpy version* of the *matrix-vector* algorithm with $n_1 = n_2 = n = 1008$ and $p = 28$.

α_1, α_2	$\frac{(n/\alpha_2)}{p}$	C_s	M_s	$\frac{M_s}{C_s}$
126,6	6	0.604×10^{-2}	2.419×10^{-2}	4.005
36,18	2	0.322×10^{-2}	1.291×10^{-2}	4.009
42,18	2	0.322×10^{-2}	1.291×10^{-2}	4.009
18,36	1	0.252×10^{-2}	1.008×10^{-2}	4.000
24,36	1	0.252×10^{-2}	1.008×10^{-2}	4.000
18,42	$\frac{24}{28}$	0.272×10^{-2}	1.089×10^{-2}	4.003
6,126	$\frac{8}{28}$	0.655×10^{-2}	1.352×10^{-2}	2.064

T_1^s		T_{pa}^s		S_p^s		$MFLOP_s$	
model	*exp*	*model*	*exp*	*model*	*exp*	*model*	*exp*
0.122	0.289	3.023×10^{-2}	0.589×10^{-1}	4.035	04.906	067,222	034.476
0.108	0.272	1.613×10^{-2}	0.243×10^{-1}	6.695	11.193	125,984	083.619
0.108	0.265	1.613×10^{-2}	0.237×10^{-1}	6.695	11.181	125,984	085.642
0.105	0.295	1.260×10^{-2}	0.212×10^{-1}	8.333	13.915	161.280	095.502
0.105	0.276	1.260×10^{-2}	0.196×10^{-1}	8.333	14.081	161.280	103.410
0.104	0.293	1.361×10^{-2}	0.231×10^{-1}	7.641	12.683	149.311	087.998
0.102	0.470	2.007×10^{-2}	0.788×10^{-1}	5.082	05.964	101.250	025.769

In general, as long as $\frac{N_1}{p} \geq 1$ the observed total computing times decrease (respectively MFLOP-rates increase) for the *block-dot version* as α_1 increases. Similarly, as long as $\frac{N_2}{p} \geq 1$ the observed total computing times decrease for the *block-saxpy version* as α_2 increases. Moreover, the *block-dot* algorithm has a better performance than the *block-saxpy* algorithm. This is due to the fact that the first algorithm uses **critical section bus** directives, while the second algorithm uses **critical section** and **critical section bus** directives.

The performance model for matrix-matrix product:

We proceed as in the previous performance model for *matrix-vector product.* We consider firstly the *block-dot* algorithm. For each iteration step of the *k-loop*, two blocks A_{ik} and B_{kj} are sent to each processor; after k iterations one array C_L is sent to main memory. Thus, we have $\alpha_1 * \alpha_2 + \alpha_2 * \alpha_3$ data transfers and $2\alpha_1 * \alpha_2 * \alpha_3$ flops are performed on different iterations k. If the computation proceeds at R flops per second and the communication proceeds at r words per second then the cost of each step of the *i-loop* requires a communication time ${T_{comm}}^d$ and a computing time ${T_{comp}}^d$ given by

$$(25) \qquad {T_{comm}}^d \approx N_3(N_2(\alpha_1\alpha_2 + \alpha_2\alpha_3) + \alpha_1\alpha_3)/r \;\; and \;\; {T_{comp}}^d = 2\alpha_1\alpha_2\alpha_3 N_2 N_3/R$$

In the similar way, for the *block-saxpy* algorithm, the cost of each step of the *j-loop* requires a communication time ${T_{comm}}^s$ and a computing time ${T_{comp}}^s$ given by

$$(26)\qquad {T_{comm}}^s \approx N_2(N_1(\alpha_1\alpha_2 + 2\alpha_1\alpha_3) + \alpha_2\alpha_3)/r \ \ and \ \ {T_{comp}}^s = 2\alpha_1\alpha_2\alpha_3 N_1 N_2/R$$

In particular, if $n_1 = n_2 = n_3 = n$, and $\alpha_1 = \alpha_2 = \alpha_3 = \alpha$, with $n_i = N_i\alpha_i$, $i = 1, 2, 3$ then the cost of each step of the *i-loop* of the *block-dot* algorithm requires: $2\alpha^3 N^2/R + N(1 + 2N)(\alpha^2/r)$ *seconds* and the cost of each step of the *j-loop* of the *block-saxpy* algorithm requires: $2\alpha^3 N^2/R + N(1+3N)(\alpha^2/r)$ *seconds*. Thus, the ratio of the computing time to the communication time on one processor for the *block-dot* algorithm is: $\frac{2}{\frac{1}{n}+\frac{2}{\alpha}} r/R$, and the ratio of the computing time to the communication time on one processor for the *block-saxpy* algorithm is given by $\frac{2}{\frac{1}{n}+\frac{3}{\alpha}} r/R$.
If now the computing time and the communication time are modeled as given above, then a reasonable estimate of ${T_1}^d$, (resp. ${T_1}^s$), ${T_{pa}}^d$ (resp. ${T_{pa}}^s$) and the MFLOP-rates $MFLOP_d$, (resp. $MFLOP_s$) for the *block-dot* (resp. *block-saxpy*) algorithm looks as follows:

$$(27)\qquad {T_1}^d \approx \{n_1 n_3 + n_1 n_2 n_3(\frac{1}{\alpha_3} + \frac{1}{\alpha_1})\}/r + 2n_1 n_2 n_3/R$$

$$(28)\qquad {T_1}^s \approx \{n_2 n_3 + n_1 n_2 n_3(\frac{1}{\alpha_3} + \frac{2}{\alpha_2})\}/r + 2n_1 n_2 n_3/R$$

$$(29)\qquad {T_{pa}}^d \approx C_d + M_d \ \ and \ \ {T_{pa}}^s \approx C_s + M_s$$

where C_d, M_d, C_s and M_s are given in section 4, formulas (9)-(14).

$$(30)\qquad MFLOP_d \approx \frac{2n_1 n_2 n_3}{{T_{pa}}^d} \ \ and \ \ MFLOP_s \approx \frac{2n_1 n_2 n_3}{{T_{pa}}^s}$$

The model and observed values of T_1, T_{pa}, S_p and MFLOP are given in Tables 5.3 and 5.4, for the *block-dot* and the *block-saxpy* version of the *matrix-matrix* algorithm, respectively.

Table 5.3: Theoretical and observed times (in seconds), speedups and *MFLOP-rates* for the *block-dot version* of the *matrix-matrix* algorithm with $n_1 = n_2 = n_3 = n = 504$ and $p = 28$.

$\alpha_1, \alpha_2, \alpha_3$	$\frac{(n/\alpha_1)}{p}$	C_d	M_d	$\frac{M_d}{C_d}$	${T_1}^d$		${T_{pa}}^d$		${S_p}^d$		$MFLOP_d$	
					model	*exp*	*model*	*exp*	*model*	*exp*	*model*	*exp*
6,36,18	3	0.228	0.205	0.896	7.836	43.463	0.433	1.597	18.097	27.215	590.460	160.010
6,18,36	3	0.228	0.179	0.785	7.658	43.663	0.408	1.600	18.769	27.289	627.200	160.270
18,36, 6	1	0.228	0.205	0.896	7.836	29.108	0.433	1.075	18.097	27.077	590.460	238.020
18,18,18	1	0.228	0.103	0.452	7.125	27.946	0.332	1.018	21.460	27.451	771.148	251.510
18, 6,36	1	0.228	0.078	0.341	6.947	28.581	0.306	1.049	22.702	27.246	835.030	243.970
36,6,18	$\frac{14}{28}$	0.457	0.078	0.170	6.947	24.429	0.535	1.771	12.985	13.793	478.372	144.580
36,18,6	$\frac{14}{28}$	0.457	0.179	0.392	7.658	25.015	0.636	1.807	12.041	13.843	402.051	141.700

Table 5.4: Theoretical and observed times (in seconds), speedups and *MFLOP-rates* for the *block-saxpy version* of the *matrix-matrix* algorithm with $n_1 = n_2 = n_3 = n = 504$ and $p = 28$.

$\alpha_1, \alpha_2, \alpha_3$	$\frac{(n/\alpha_3)}{p}$	C_s	M_s	$\frac{M_s}{C_s}$
36,18, 6	3	0.228	0.255	1.563
18,36, 6	3	0.228	0.068	0.896
36, 6,18	1	0.228	0.357	1.563
18,18,18	1	0.228	0.154	0.674
6,36,18	1	0.228	0.103	0.452
18,6,36	$\frac{14}{28}$	0.457	0.332	0.726
6,18,36	$\frac{14}{28}$	0.457	0.128	0.281

$T_1{}^s$		$T_{pa}{}^s$			$S_p{}^s$			$MFLOP_s$		
model	*exp*	*model*	*exp*1	*exp*2	*model*	*exp*1	*exp*2	*model*	*exp*1	*exp*2
7.303	25.399	0.484	1.144	0.964	15.08	22.20	26.34	528.540	223.820	270.520
7.125	28.607	0.296	1.234	1.048	24.07	23.18	27.29	862.241	207.370	244.130
8.903	26.521	0.586	1.208	1.010	15.19	21.95	26.25	436.904	211.370	253.480
7.480	28.143	0.382	1.140	1.032	19.58	24.68	27.27	668.815	224.520	247.960
7.836	42.389	0.332	1.665	1.543	23.60	25.45	27.47	771.147	153.720	165.880
8.903	30.501	0.789	2.357	2.257	11.28	12.94	13.51	324.413	108.620	113.450
8.192	43.203	0.586	3.199	3.156	13.97	13.50	13.69	436.904	080.025	081.111

Note: expl (resp. exp2) denotes the numerical experiment performed without (resp. with) **critical section bus regrouping technique**, see Section 4.

In the same way as in the previous model, we expect that as long as $\frac{N_1}{p} \geq 1$ the observed total computing time will decrease for *block-dot* algorithm if α_1, α_3 increase. This is supported by the information of Table 5.3. We expect also that as long as $\frac{N_3}{p} \geq 1$ the observed total computing time decreases for *block-saxpy* if α_2, α_3 increase. This is supported by the information of Table 5.4. So the property given in Section 4 is verified now for the *block-dot* algorithm (except for blocks of size $\alpha_1 = 18$, $\alpha_1 = 6$ and $\alpha_3 = 36$). Furthermore, the property given in Section 4 is not verified for the *block-saxpy* algorithm. What this suggests is that the total idle time becomes more significant. Therefore, the total computing time is affected more and more, particularly if the number of processes is increased, and this causes a degraded performance.
For **RGMMUL**, the subroutine has been programmed to achieve as close as possible to the theoretical values of the *MFLOP-rates*. In our implementations, we have reached the theoretical results. The figures 4.9 and 4.10 confirm this.

Remark: The choice made on the processor's cache block sizes is not arbitrary. The block sizes have to satisfy some constraints, due to some cache characteristics. Firstly, the block sizes have to satisfy: $\alpha_1 \equiv 0$ [mod 2], $\alpha_2 \equiv 0$ [mod 2] (with α_3 arbitrary) if we compute with double precision arithmetic. Secondly, the block sizes must be chosen so that the block matrices can be stored completely in the cache. Finally, the block sizes have to satisfy: $n_i = \alpha_i \times N_i$, $i = 1, 2, 3$. The two last conditions are also mentioned in the previous sections.

Conclusions and remarks:

We have carried out several experiments with simple linear algebra operations on the Cray S-MP System 500 matrix coprocessor. In particular, we have studied different block algorithm techniques applied to the *matrix-vector* and *matrix-matrix operations*. A number of characteristics related to the matrix coprocessor configurations and the block size influence on the *matrix-vector* and *matrix-matrix product* have been studied. Furthermore, we have presented a performance model for both operations concerning the total computing time and the *MFLOP-rates*. We have compared this with our experiments. It turned out that the numerical results are worse than what is predicted by the performance model. This can partly be explained by the fact that

- The block algorithms are not implemented in terms of calls to *Optimal Hand Coded Math Routines* (Matrix Coprocessor's vector primitives), like useful routines as _ **dvmv** and _ **dvmm** [FPS91a]. These routines are designed to operate on data that have been put in cache. The subroutine _ **dvmv** (resp. _ **dvmm**) multiplies the elements of a matrix and a vector (resp. a matrix) in the cache.
- Processing elements on the same bus can compute in parallel, but can **not** communicate with the main memory at the same time. Our performance model only roughly accounts for the idle time induced by this bottleneck. (This idle time is a complicated function of the number of processing elements per bus, of the ratio of computing to communication time, of the places in the algorithm where communication has to be carried out, and of the synchronization points in the algorithm.)
- Our performance model does not account for overhead caused by loops, the use of mpp and pfp directives, and data initialization.

APPENDIX A: Some hardware and software characteristics of the System 500 matrix coprocessor

A.1 mpp and pfp directives

mpp and **pfp** comment directives have to be used for parallelizing programs. For either type a preprocessor is invoked which interprets these directives and creates special Fortran code for the matrix coprocessor, to be compiled and executed subsequently by the Fortran compiler.

- **mpp** *directives*
  The scope of an **mpp** (**m**atrix **p**rocedure **p**reprocessor) directive is a subroutine or a function. The directive informs the preprocessor that the subroutine or function that follows has to be executed on the matrix coprocessor. It specifies the type and use (e.g., **INPUT**, **OUTPUT**, or **INOUT**) of the parameters of the subroutine or the function, and of common blocks, if appropriate. An **mpp** directive has the general form:

  CMCP *directive* [*modifiers*]

  where *directive* may be one of the following:

  SUBROUTINE
  FUNCTION
  INPUT
  OUTPUT
  INOUT
  CLIENT
  SERVER

  modifiers supplies one or more additional arguments to the directive. For details see [FPS91a, pp. 6-1/6-9].

- **pfp** *directives*
  The scope of a **pfp** (**p**arallel **f**ortran **p**reprocessor) directive is the code that immediately follows the directive. It specifies which loops in a Fortran program have to be executed in parallel. A **pfp** directive has the following general form:

  CPCF *directive* [*modifiers*]

  where *directive* may be one of the following:

  PARALLEL
  END PARALLEL
  SINGLE PROCESS

END SINGLE PROCESS
PDO
CRITICAL SECTION
CRITICAL SECTION BUS
END CRITICAL SECTION
PRIVATE
BARRIER

modifiers supplies one or more additional arguments to the directive. For details see [FPS91a, pp. 7-1/18].

APPENDIX B: The listings of our parallel Fortran block subroutines for matrix-vector and matrix-matrix multiplication.

c * * * Locality example 1 matrix-vector multiplication

```
c  Fortran matrix-vector multiply executed in cache
c
CMCP  SUBROUTINE Block-dot(A,B,C,n,alpha1,alpha2,lda)
CMCP  INPUT    REAL*8    A(lda,n),B(n)
CMCP  OUTPUT   REAL*8    C(n)
CMCP  INPUT    INTEGER*4 n,alpha1,alpha2,lda
c
c     Parallel Fortran matrix-vector multiplication
c
c***************************************************************************
      Subroutine Block-dot(A,B,C,n,alpha1,alpha2,lda)
c***************************************************************************
ccccc PURPOSE:
c---------------------------------------------------------------------------
c     This subroutine determines matrix-vector multiplication with block dot
c     approach ik-version, Ci=Ci+Aik*Bk. Aik is a block matrix of size
c     alpha1*alpha2, Bk and Ci are segment vectors of size alpha2, alpha1
c     respectively, where
c     Aik = A((i-1)*alpha1+1:i*alpha1,(k-1)*alpha2+1:k*alpha2),
c     Bk = B((k-1)*alpha2+1:k*alpha2) and  Ci= C((i-1)*alpha1+1:i*alpha1).
c     The subroutine is parallelized over the i-loop in a such way that
c     different processors will perform different iterations of the loop.
c     Each processor, at a given time will compute independently on different
c     segments  Ci, i=1,..., N1, of C.
c----------------------------------------------------------------------------
ccccc VARIABLES IDENTIFICATION:
c
c     On entry:
c
c     A                      real*8(lda,*) the matrix A.
c     B                      real*8(*)     the vector B.
c     n                      integer*4     the order of A, B; n must be
c                                          less than lda.
c     alpha1,alpha2          integer*4     the sizes of the block
c                                          matrices Aik,Bk and Ci.
c     lda                    integer*4     the row dimension of array A
c     CA, CB                 real*8        storage allocations for the
c                                          transferred block matrices Aik,
```

```
c                                             Bk from the matrix registers
c                                             into the cache.
c     On exit:
c
c     CC                    real*8            storage allocation for the
c                                             multiplication of the vectors
c                                             CA by CB in the cache.
c     C                     real*8(*)         the matrix A by B multiplication.
c
c     Local variables:
c
c     ib, kb                integer*4         specify the number of block matrices.
c     i,k                   integer*4
c     bound1,bound2         integer*4
c     KVL                   integer*4         specifies the cache size.
c     volume                integer*4         specifies the total number of
c                                             data transferred into the cache.
c
ccccc TYPE DECLARATION AND STORAGE ALLOCATION:
c
      integer*4     n,alpha1,alpha2,lda
      real*8        A(lda,*),B(*),C(*)
      real*8        CA(1),CB(1),CC(1)
      integer*4     ib,kb,i,k,bound1,bound2
      integer*4     KVL,volume
c-----------------------------------------------------------------
ccccc EQUIVALENCES:
c     A non-constant expression is not allowed as an index in the array
c     MCP_DREG of the EQUIVALENCE-statements given below where a constant
c     expression is required. For this reason, we illustrate in this algorithm
c     by means of an example how the cache can be divided into three cache
c     arrays of 64-bit elements.
c     Suppose the block matrix Aik is of size 18x36 and the segment vectors
c     Bk, Ci are of size 36, 18 respectively.
ccccc
      INCLUDE       'mcpreg.h'
      PARAMETER     (KVL=((MCP_DREG_SIZE)/2)*2)
c
c     CA is a cache array of length KVL-(18+36)=970.
      EQUIVALENCE   (CA(1),MCP_DREG(1))
c
c     CB is a cache array of length alpha2=36.
      EQUIVALENCE   (CB(1),MCP_DREG(1+KVL-54))
```

```
c
c     CC is a cache array of length alpha1=18.
      EQUIVALENCE  (CC(1),MCP_DREG(1+KVL-18))
c-----------------------------------------------------------------------
      bound1=n/alpha1
      bound2=n/alpha2
      volume=alpha1*alpha2+alpha2+alpha1
      if ((n.NE.alpha1*bound1).OR.
     $   (n.NE.alpha2*bound2).OR.
     $   (volume.GT.1024))           then
         print *, 'alpha1 or alpha2 is not a divisor of n'
         print *, 'alpha1 =', alpha1,'alpha2 =', alpha2,
     $              ' n =', n
         print *, 'or data does not fit in the cache'
         stop
      endif
c
CPCF  PARALLEL
CPCF  PRIVATE i,k,kb
CPCF  PDO
c
      do 120 ib=1,bound1
c
c     Initialize the matrix CC in the cache:
c
      do 10 i=1,alpha1
         CC(i)=0.0d0
   10 continue
c
             do 100 kb=1,bound2
c
CPCF CRITICAL SECTION BUS
c
c     Read the strip Bk using utility routine;
c                    Bk= B((kb-1)*alpha2+1:kb*alpha2)
c
         call _DVLOAD(B((kb-1)*alpha2+1),
     $                   1,CB,1,alpha2)
c
c     Read the block Aik using utility routine;
c                Aik= A((ib-1)*alpha1+1:ib*alpha1,(kb-1)*alpha2+1:kb*alpha2)
c
                call _DMLOAD (A((ib-1)*alpha1+1,(kb-1)*alpha2+1),
```

```
     $                      1,lda,CA,1,alpha1,alpha1,alpha2)
c
CPCF  END CRITICAL SECTION
c
c      Compute the result: Ci=Ci+Aik*Bk;
c                     Ci= C((ib-1)*alpha1+1:ib*alpha1)
c
         do 80 k=1,alpha2
            do 70 i=1,alpha1
               CC(i)=CA((k-1)*alpha1+i)*CB(k)
     $                       +CC(i)
   70       continue
   80    continue
c
  100       continue
c
CPCF  CRITICAL SECTION BUS
c
c     Store the output Ci strip into memory using utility routine;
c                     Ci=C((ib-1)*alpha1+1:ib*alpha1)
c
                call _DVSTOR (CC,1,C((ib-1)*alpha1+1),
     $                      1,alpha1)
c
CPCF  END CRITICAL SECTION
c
  120 continue
c
CPCF  END PARALLEL
c
      return
      end
```

c * * * Locality example 2 matrix-vector multiplication

```
c  Fortran matrix multiply executed in cache
c
CMCP  SUBROUTINE Block-saxpy(A,B,C,n,alpha1,alpha2,lda)
CMCP  INPUT    REAL*8     A(lda,n),B(n)
CMCP  OUTPUT   REAL*8     C(n)
CMCP  INPUT    INTEGER*4 n,alpha1,alpha2,lda
c
c     Parallel Fortran matrix-vector multiplication
c*************************************************************************
      Subroutine Block-saxpy(A,B,C,n,alpha1,alpha2,lda)
c*************************************************************************
ccccc PURPOSE:
c------------------------------------------------------------------------
c     This subroutine determines matrix-vector multiplication with block saxpy
c     approach ki-version, Ci=Ci+Aik*Bk. Aik is a block matrix of size
c     alpha1*alpha2, Bk and Ci are segment vectors of size alpha2, alpha1
c     respectively, where
c     Aik = A((i-1)*alpha1+1:i*alpha1,(k-1)*alpha2+1:k*alpha2),
c     Bk  = B((k-1)*alpha2+1:k*alpha2) and  Ci= C((i-1)*alpha1+1:i*alpha1).
c     The subroutine is parallelized over the k-loop in a such way that
c     different processors will perform different iterations of the loop.
c     For each iterations step of the i-loop the segment Ci (i=1,..., N1)
c     of C is updated, and this is processed by one processor at a time.
c-------------------------------------------------------------------------
ccccc VARIABLES IDENTIFICATION:
c
c     On entry:
c
c     A                     real*8(lda,*) the matrix A.
c     B                     real*8(*)     the vector B.
c     n                     integer*4     the order of A, B; n must be
c                                         less than lda.
c     alpha1,alpha2         integer*4     the sizes of the block
c                                         matrices Aik,Bk and Ci.
c     lda                   integer*4     the row dimension of array A
c     CA, CB                real*8        storage allocations for the
c                                         transferred block matrices Aik,
c                                         Bk from the matrix registers
c                                         into the cache.
c     On exit:
c
```

```
c     CC                    real*8       storage allocation for the
c                                        multiplication of the vectors
c                                        CA by CB in the cache.
c     C                     real*8(*)    the matrix A by B multiplication.
c
c     Local variables:
c
c     kb,ib                 integer*4    specify the number of blocks.
c     i,k                   integer*4
c     bound1,bound2         integer*4
c     KVL                   integer*4    specifies the cache size.
c     volume                integer*4    specifies the total number of
c                                        data transferred into the cache.
c
ccccc TYPE DECLARATION AND STORAGE ALLOCATION:
c
      integer*4    n,alpha1,alpha2,lda
      real*8       A(lda,*),B(*),C(*)
      real*8       CA(1),CB(1),CC(1)
      integer*4    ib,kb,i,k,bound1,bound2
      integer*4    KVL,volume
c-------------------------------------------------------------------
ccccc EQUIVALENCES:
c     A non-constant expression is not allowed as an index in the array
c     MCP_DREG of the EQUIVALENCE-statements given below where a constant
c     expression is required. For this reason, we illustrate in this algorithm
c     by means of an example how the cache can be divided into three cache
c     arrays of 64-bit elements.
c     Suppose the block matrix Aik is of size 18x36 and the segment vectors
c     Bk, Ci are of size 36, 18 respectively.
ccccc
      INCLUDE      'mcpreg.h'
      PARAMETER    (KVL=((MCP_DREG_SIZE)/2)*2)
c
c     CA is a cache array of length KVL-(18+36)=970.
      EQUIVALENCE  (CA(1),MCP_DREG(1))
c
c     CB is a cache array of length alpha2=36.
      EQUIVALENCE  (CB(1),MCP_DREG(1+KVL-54))
c
c     CC is a cache array of length alpha1=18.
      EQUIVALENCE  (CC(1),MCP_DREG(1+KVL-18))
c-------------------------------------------------------------------
```

```
      bound1=n/alpha1
      bound2=n/alpha2
      volume=alpha1*alpha2+alpha2+alpha1
      if ((n.NE.alpha1*bound1).OR.
     $   (n.NE.alpha2*bound2).OR.
     $   (volume.GT.1024))            then
         print *, 'alpha1 or alpha2 is not a divisor of n'
         print *, 'alpha1 =', alpha1,'alpha2 =', alpha2,
     $             ' n =', n
         print *, 'or data does not fit in the cache'
         stop
      endif
CPCF  PARALLEL
CPCF  PDO
c
c     Initialize the vector C:
c
      do 11 i=1,n
            C(i)=0.0d0
   11 continue
c
CPCF  END PARALLEL
c
CPCF  PARALLEL
CPCF  PRIVATE i,k,ib
CPCF  PDO
      do 110 kb=1,bound2
c
CPCF  CRITICAL SECTION BUS
c
c     Read the strip Bk using utility routine;
c                       Bk=B((kb-1)*alpha2+1:kb*alpha2)
c
         call _DVLOAD(B((kb-1)*alpha2+1),
     $                    1,CB,1,alpha2)
c
CPCF  END CRITICAL SECTION
c
            do 100 ib=1,bound1
c
CPCF CRITICAL SECTION BUS
c
c     Read the block Aik using utility routine;
```

```
c                    Aik=A((ib-1)*alpha1+1:ib*alpha1,(kb-1)*alpha2+1:kb*alpha2)
c
                call _DMLOAD(A((ib-1)*alpha1+1,(kb-1)*alpha2+1),
     $                       1,lda,CA,1,alpha1,alpha1,alpha2)
c
CPCF  END CRITICAL SECTION
c
c     Compute the result: Ci=Ci+Aik*Bk;
c                         Ci=C((ib-1)*alpha1+1:ib*alpha1)
c
      do 10 i=1,alpha1
            CC(i)=0.0d0
   10 continue
         do 80 k=1,alpha2
            do 70 i=1,alpha1
               CC(i)=CA((k-1)*alpha1+i)*CB(k)
     $                         +CC(i)
   70       continue
   80    continue
CPCF  CRITICAL SECTION
c
c     add the result to the ib-th segment of C.
c
      do 90 i=1, alpha1
         C((ib-1)*alpha1+i)=C((ib-1)*alpha1+i)+CC(i)
   90 continue
c
CPCF  END CRITICAL SECTION
c
  100       continue
  110 continue
c
CPCF  END PARALLEL
c
      return
      end
```

c * * * Example 3 matrix-vector multiplication

```
c       Matrix data structure (row storage scheme) :
c       a1,1    a1,2    a1,3    ...     a1,n    a2,1    a2,2    a2,3
c       ...     a2,n    a3,1    a3,2    ...
c
      CMCP SUBROUTINE MATVEC3(A, B, C, N)
      CMCP INPUT  REAL*8      A(N*N), B(N)
      CMCP OUTPUT REAL*8      C(N)
      CMCP INPUT  INTEGER*4   N
C
C*****************************************************************************
      SUBROUTINE MATVEC3(A, B, C, N)
C*****************************************************************************
c
ccccc TYPE DECLARATION AND STORAGE ALLOCATION:
c
      REAL*8       A(*), B(*), C(*)
      INTEGER*4    N
      INTEGER*4    i, j
C
-------------------------------------------------------------------
CPCF PARALLEL
CPCF PDO
      do 10 i = 1, n
         c(i) = 0.0d0
         do 20 j = 1, n
         c(i) = c(i) + A((i-1)*n+j)*B(j)
20       continue
10    continue
CPCF END PARALLEL
C
      RETURN
      END
```

c * * * **Locality example 1 matrix-matrix multiplication**

```
c Fortran Block matrix multiply executed in cache
c
CMCP  SUBROUTINE Block-dot(A,B,C,n,alpha1,alpha2,alpha3,lda,ldb,ldc)
CMCP  INPUT   REAL*8    A(lda,n),B(ldb,n)
CMCP  OUTPUT  REAL*8    C(ldc,n)
CMCP  INPUT   INTEGER*4 n,alpha1,alpha2,alpha3,lda,ldb,ldc
c
c     Parallel Fortran Block matrix-matrix multiplication
c*****************************************************************************
      Subroutine Block-dot(A,B,C,n,alpha1,alpha2,alpha3,lda,ldb,ldc)
c*****************************************************************************
ccccc PURPOSE:
c----------------------------------------------------------------------------
c     The subroutine determines matrix-matrix multiplication with block dot
c     approach ijk-version, Cij=Cij+Aik*Bkj. Aik is a block matrix of size
c     alpha1*alpha2, Bkj is a block matrix of size alpha2*alpha3 and Cij is
c     a block matrix of size alpha1*alpha3, where
c     Aik = A((i-1)*alpha1+1:i*alpha1,(k-1)*alpha2+1:k*alpha2),
c     Bkj = B((k-1)*alpha2+1:k*alpha2,(j-1)*alpha3+1:j*alpha3) and
c     Cij = C((i-1)*alpha1+1:i*alpha1,(j-1)*alpha3+1:j*alpha3).
c     The subroutine is parallelized over the i-loop in a such way that
c     different processors will perform different iterations of the loop.
c     Each processor, at the same time  will compute independently on
c     different rows of blocks Cij, j=1,..., N3, of C.
c-----------------------------------------------------------------------------
ccccc VARIABLES IDENTIFICATION:
c
c     On entry:
c
c     A                     real*8(lda,*) the matrix A.
c     B                     real*8(ldb,*) the matrix B.
c     n                     integer*4     the order of A, B; n must be
c                                         less than lda, ldb and ldc.
c     alpha1,alpha2,alpha3  integer*4     the sizes of the block
c                                         matrices Aik,Bkj and Cij.
c     lda,ldb,ldc           integer*4     the row dimensions of arrays A, B
c                                         and C.
c     CA, CB                real*8        storage allocations for the
c                                         transferred block matrices Aik,
c                                         Bkj from the matrix registers
c                                         into the cache.
```

```
c     On exit:
c
c     CC                    real*8          storage allocation for the
c                                           multiplication of the vectors
c                                           CA by CB in the cache.
c     C                     real*8(ldc,*)   the matrix A by B multiplication.
c
c     Local variables:
c
c     ib,jb,kb              integer*4       specify the number of blocks.
c     i,j,k                 integer*4
c     bound1,bound2,bound3  integer*4
c     KVL                   integer*4       specifies the cache size.
c     volume                integer*4       specifies the total number of
c                                           data transferred into the cache.
c
ccccc TYPE DECLARATION AND STORAGE ALLOCATION:
c
      integer*4     n,alpha1,alpha2,alpha3,lda,ldb,ldc
      real*8        A(lda,*),B(ldb,*),C(ldc,*)
      real*8        CA(1), CB(1),CC(1)
      integer*4     ib,jb,kb,i,j,k,bound1,bound2,bound3
      integer*4     KVL,volume
c-----------------------------------------------------------------------------
ccccc EQUIVALENCES:
c     A non-constant expression is not allowed as an index in the array
c     MCP_DREG of the EQUIVALENCE-statements given below where a constant
c     expression is required. For this reason, we illustrate in this algorithm
c     by means of an example how the cache can be divided into three cache
c     arrays of 64-bit elements.
c     Suppose the block matrices Aik, Bkj, Cij are of size 18x36, 36x6 and
c     18x6 respectively.
ccccc
      INCLUDE       'mcpreg.h'
      PARAMETER     (KVL=((MCP_DREG_SIZE)/2)*2)
c
c     CA is a cache array of length KVL-(36x6+18x6)=700.
      EQUIVALENCE  (CA(1),MCP_DREG(1))
c
c     CB is a cache array of length alpha2xalpha3=36x6.
      EQUIVALENCE  (CB(1),MCP_DREG(1+KVL-324))
c
c     CC is a cache array of length alpha1xalpha3=18x6.
```

```
      EQUIVALENCE  (CC(1),MCP_DREG(1+KVL-108))
c-----------------------------------------------------------------------
      bound1=n/alpha1
      bound2=n/alpha2
      bound3=n/alpha3
      volume=alpha1*alpha2+alpha2*alpha3+alpha1*alpha3
      if ((n.NE.alpha1*bound1).OR.
     $   (n.NE.alpha2*bound2).OR.
     $   (n.NE.alpha3*bound3).OR.
     $   (volume.GT.1024))          then
         print *, 'alpha1 or alpha2 or alpha3 is not a divisor of n'
         print *, 'alpha1 =', alpha1,'alpha2 =', alpha2,'alpha3 =',
     $            alpha3, ' n =', n
         print *, 'or data does not fit in the cache'
         stop
      endif
c
CPCF  PARALLEL
CPCF  PRIVATE i,j,k,jb,kb
CPCF  PDO
c
      do 120 ib=1,bound1
         do 110 jb=1,bound3
c
c     Initialize the matrix CC in the cache:
c
      do 10 j=1,alpha3
         do 10 i=1,alpha1
               CC((j-1)*alpha1+i)=0.0d0
   10 continue
c
            do 100 kb=1,bound2
c
CPCF CRITICAL SECTION BUS
c
c     Read the block Aik using utility routine;
c                   Aik=A((ib-1)*alpha1+1:ib*alpha1,(kb-1)*alpha2+1:kb*alpha2)
c
                 call _DMLOAD(A((ib-1)*alpha1+1,(kb-1)*alpha2+1),
     $                     1,lda,CA,1,alpha1,alpha1,alpha2)
c
c     Read the block Bkj using utility routine;
c                   Bkj=B((kb-1)*alpha2+1:kb*alpha2,(jb-1)*alpha3+1:jb*alpha3)
```

```
c
        call _DMLOAD(B((kb-1)*alpha2+1,(jb-1)*alpha3+1),
     $                   1,ldb,CB,1,alpha2,alpha2,alpha3)
c
CPCF  END CRITICAL SECTION
c
c     Compute the result: Cij=Cij+Aik*Bkj;
c                      Cij=C((ib-1)*alpha1+1:ib*alpha1,(jb-1)*alpha3+1:jb*alpha3)
c
       do 90 j=1,alpha3
        do 80 k=1,alpha2
           do 70 i=1,alpha1
              CC((j-1)*alpha1+i)=CA((k-1)*alpha1+i)*CB((j-1)*alpha2+k)
     $                               +CC((j-1)*alpha1+i)
   70         continue
   80      continue
   90   continue
c
  100         continue
c
CPCF  CRITICAL SECTION BUS
c
c    Store the output Cij block into memory using utility routine;
c                      Cij=C((ib-1)*alpha1+1:ib*alpha1,(jb-1)*alpha3+1:jb*alpha3)
c
               call _DMSTOR(CC,1,alpha1,C((ib-1)*alpha1+1,
     $                         (jb-1)*alpha3+1),1,ldc,alpha1,alpha3)
c
CPCF  END CRITICAL SECTION
c
  110      continue
  120 continue
c
CPCF  END PARALLEL
c
      return
      end
```

c *** **Locality example 2 matrix-matrix multiplication**

```
c Fortran Block matrix multiply executed in cache
c
CMCP  SUBROUTINE Block-saxpy(A,B,C,n,alpha1,alpha2,alpha3,lda,ldb,ldc)
CMCP  INPUT   REAL*8    A(lda,n),B(ldb,n)
CMCP  OUTPUT  REAL*8    C(ldc,n)
CMCP  INPUT   INTEGER*4 n,alpha1,alpha2,alpha3,lda,ldb,ldc
c
c     Parallel Fortran Block matrix-matrix multiplication
c
c*****************************************************************************
      Subroutine Block-saxpy(A,B,C,n,alpha1,alpha2,alpha3,lda,ldb,ldc)
c*****************************************************************************
ccccc PURPOSE:
c-----------------------------------------------------------------------------
c     The subroutine determines matrix-matrix multiplication
c     with block saxpy approach jki-version, Cij=Cij+Aik*Bkj,Aik is a block
c     matrix of size alpha1*alpha2, Bkj is a block matrix of size alpha2*alpha3
c     and Cij is a block matrix of size alpha1*alpha3, where
c     Aik = A((i-1)*alpha1+1:i*alpha1,(k-1)*alpha2+1:k*alpha2),
c     Bkj = B((k-1)*alpha2+1:k*alpha2,(j-1)*alpha3+1:j*alpha3) and
c     Cij = C((i-1)*alpha1+1:i*alpha1,(j-1)*alpha3+1:j*alpha3).
c     The subroutine is parallelized over the j-loop in a such way that
c     different processors will perform different iterations of the loop.
c     Each processor, at the same time  will compute independently on
c     different columns of blocks Cij, i=1,..., N1, of C.
c-----------------------------------------------------------------------------
ccccc VARIABLES IDENTIFICATION:
c
c     On entry:
c
c     A                      real*8(lda,*) the matrix A.
c     B                      real*8(ldb,*) the matrix B.
c     n                      integer*4     the order of A, B; n must be
c                                          less than lda, ldb and ldc.
c     alpha1,alpha2,alpha3   integer*4     the sizes of the block
c                                          matrices Aik,Bkj and Cij.
c     lda,ldb,ldc            integer*4     the row dimensions of arrays A, B
c                                          and C.
c     CA, CB                 real*8        storage allocations for the
c                                          transferred block matrices Aik,
c                                          Bkj from the matrix registers
```

```
c                                            from the matrix registers
c                                            into the cache.
c      On exit:
c
c      CC                    real*8          storage allocation for the
c                                            multiplication of the vectors
c                                            CA by CB in the cache.
c      C                     real*8(ldc,*)   the matrix A by B multiplication.
c
c      Local variables:
c
c      ib,jb,kb              integer*4       specify the number of blocks.
c      i,j,k                 integer*4
c      bound1,bound2,bound3  integer*4
c      KVL, lastib           integer*4       specifies the cache size.
c      volume                integer*4       specifies the total number of
c                                            data transferred into the cache.
c
ccccc TYPE DECLARATION AND STORAGE ALLOCATION:
c
      integer*4    n,alpha1,alpha2,alpha3,lda,ldb,ldc
      real*8       A(lda,*),B(ldb,*),C(ldc,*)
      REAL*8       CA(1), CB(1),CC(1)
      integer*4    ib,jb,kb,i,j,k,bound1,bound2,bound3,lastib
      integer*4    KVL,volume
c------------------------------------------------------------------------------
ccccc EQUIVALENCES:
c     A non-constant expression is not allowed as an index in the array
c     MCP_DREG of the EQUIVALENCE-statements given below where a constant
c     expression is required. For this reason, we illustrate in this algorithm
c     by means of an example how the cache can be divided into three cache
c     arrays of 64-bit elements.
c     Suppose the block matrices Aik, Bkj, Cij are of size 18x36, 36x6 and
c     18x6 respectively.
ccccc
      INCLUDE        'mcpreg.h'
      PARAMETER      (KVL=((MCP_DREG_SIZE)/2)*2)
c
c     CA is a cache array of length KVL-(36x6+18x6)=700.
      EQUIVALENCE  (CA(1),MCP_DREG(1))
c
c     CB is a cache array of length alpha2xalpha3=36x6.
      EQUIVALENCE  (CB(1),MCP_DREG(1+KVL-324))
```

```
c
c     CC is a cache array of length alpha1xalpha3=18x6.
      EQUIVALENCE  (CC(1),MCP_DREG(1+KVL-108))
c--------------------------------------------------------------------------------
      bound1=n/alpha1
      bound2=n/alpha2
      bound3=n/alpha3
      volume=alpha1*alpha2+alpha2*alpha3+alpha1*alpha3
      if ((n.NE.alpha1*bound1).OR.
     $   (n.NE.alpha2*bound2).OR.
     $   (n.NE.alpha3*bound3).OR.
     $   (volume.GT.1024))           then
         print *, 'alpha1 or alpha2 or alpha3 is not a divisor of n'
         print *, 'alpha1 =', alpha1,'alpha2 =', alpha2,'alpha3 =',
     $             alpha3, ' n =', n
         print *, 'data does not fit in the cache'
         stop
      endif
c     Initialize the matrix C:
c
CPCF  PARALLEL
CPCF  PRIVATE i
CPCF  PDO
c
      do 10 j=1,n
         do 10 i=1,n
            c(i,j)=0.0d0
   10 continue
c
CPCF  END PARALLEL
c
CPCF  PARALLEL
CPCF  PRIVATE i,k,j,kb,ib,lastib
CPCF  PDO
c
      do 120 jb=1,bound3
         do 110 kb=1,bound2
c
CPCF  CRITICAL SECTION BUS
c
c     Read the block Bkj using utility routine;
c                    Bkj=B((kb-1)*alpha2+1:kb*alpha2,(jb-1)*alpha3+1:jb*alpha3)
c
```

```
        call _DMLOAD(B((kb-1)*alpha2+1,(jb-1)*alpha3+1),
     $                 1,ldb,CB,1,alpha2,alpha2,alpha3)
c
CPCF  END CRITICAL SECTION
c
           lastib=-1
           do 100 ib=1,bound1
c
CPCF CRITICAL SECTION BUS
c
              IF (lastib.EQ.-1)         THEN
c
c     Read the block Aik using utility routine;
c                    Aik=A((ib-1)*alpha1+1:ib*alpha1,(kb-1)*alpha2+1:kb*alpha2)
c
                call _DMLOAD(A((ib-1)*alpha1+1,(kb-1)*alpha2+1),
     $                    1,lda,CA,1,alpha1,alpha1,alpha2)
c
c     Read the block Cij using utility routine;
c                    Cij=C((ib-1)*alpha1+1:ib*alpha1,(jb-1)*alpha3+1:jb*alpha3)

                call _DMLOAD(C((ib-1)*alpha1+1,(jb-1)*alpha3+1),
     $                      1,ldc,CC,1,alpha1,alpha1,alpha3)
              ELSE
c
c    Store output Cij block  into memory using utility routine;
c                 Cij=C((ib-1)*alpha1+1:ib*alpha1,(jb-1)*alpha3+1:jb*alpha3)
c
                call _DMSTOR(CC,1,alpha1,C((lastib-1)*alpha1+1,
     $                         (jb-1)*alpha3+1),1,ldc,alpha1,alpha3)
                call _DMLOAD(A((ib-1)*alpha1+1,(kb-1)*alpha2+1),
     $                    1,lda,CA,1,alpha1,alpha1,alpha2)
                call _DMLOAD(C((ib-1)*alpha1+1,(jb-1)*alpha3+1),
     $                      1,ldc,CC,1,alpha1,alpha1,alpha3)
              ENDIF
c
CPCF  END CRITICAL SECTION
c
c     Compute the result: Cij=Cij+Aik*Bkj;
c                    Cij= C((ib-1)*alpha1+1:ib*alpha1,(jb-1)*alpha3+1:jb*alpha3)
c
        do 90 j=1,alpha3
         do 80 k=1,alpha2
```

```
            do 70 i=1,alpha1
               CC((j-1)*alpha1+i)=CA((k-1)*alpha1+i)*CB((j-1)*alpha2+k)
     $                                +CC((j-1)*alpha1+i)
   70       continue
   80    continue
   90  continue
c
          lastib=ib
c
  100       continue
c
               IF (lastib.NE.-1)      THEN
c
CPCF  CRITICAL SECTION BUS
c
c    Store the last output Cij block into memory using utility routine;
c                  Cij=C((ib-1)*alpha1+1:ib*alpha1,(jb-1)*alpha3+1:jb*alpha3)

                call _DMSTOR(CC,1,alpha1,C((lastib-1)*alpha1+1,
     $                     (jb-1)*alpha3+1),1,ldc,alpha1,alpha3)
c
CPCF  END CRITICAL SECTION
c
               ENDIF
c
  110    continue
  120 continue
c
CPCF  END PARALLEL
c
      return
      end
```

c * * * Locality example 3 matrix-matrix multiplication

```
C
C     Locality of the user written routine "rgmmul'.
C
      PROGRAM MMtime
C
      EXTERNAL DTIME64
      REAL*8 DTIME64
      REAL*8 DTARRAY(2)
      REAL*8 DT
C
      INTEGER*4 VEC_LEN
      PARAMETER (VEC_LEN = 1000)
      REAL*8        A(VEC_LEN*VEC_LEN), B(VEC_LEN*VEC_LEN)
      REAL*8        C(VEC_LEN*VEC_LEN)
      INTEGER*4     N
      REAL*8        CX, mflops
      INTEGER*4     NBUS, NCPUBUS
      INTEGER*4     NOPS, NWORDS
      REAL*4        COMPI
C
C-------------------------------------------------------------------
      write(6,601)
601   format(/1x,'**************************************************',
     *       /1x,'*                                                *',
     *       /1x,'*  Benchmark Study for User Routine rgmmul       *',
     *       /1x,'*                                                *',
     *       /1x,'**************************************************')
      NERRORS = 0
C
C Get the current MCP configuration
C
      NBUS = MCP_NBUS()
      NCPUBUS = MCP_NCPUBUS()
      write(6,611) nbus,ncpubus,nbus*ncpubus
611   format(//1x,'MCP Configuration :',
     *        /6x,'Number of Processors = ',i1,
     *        '(buses) * ',i2,'(processors/bus) = ',i2,
     *      //6x,'Double precision: DTIME64, 50 nanosec resolution',
     *     ///1x,' Dim',3x,3x,'Timings(sec.)',3x,10x,'Mflops',3x,
     *        2x,'Num of Op',3x,3x,'Locality')
C
```

```
C Initialize the input data
C
      DO 100 I = 1, VEC_LEN
      DO 100 J = 1, VEC_LEN
      A(I+(J-1)*vec_len) = 2.0d0 * dfloat(I + J)
      B((I-1)*vec_len+J) = 3.0d0*dfloat(J) + 4.0d0*dfloat(I)
100   CONTINUE
C
      open(unit=2,file='tmp.mat')
      do 400 n = 10, VEC_LEN, 10
c      n = 1000
C
C Calculate the compute intensity for the DOTP operation
C
C     rgmmul:          2* N**3      operations
C
C     A:              1 *  N*N  words
C     B:              1 *  N*N  words
C     C:              2 *  N*N  words
C
      NOPS = 2*(N**3)
      NWORDS = (1+1+2) * N*N
      COMPI = FLOAT(NOPS) / FLOAT(NWORDS)
C
C Call the routine.
C
      DT = DTIME64 (DTARRAY)
      do 170 iloop = 1, 5
      CALL rgmmul(0,n,n,n,a,1,n,b,n,1,c,n,1)
170   continue
   DT = DTIME64 (DTARRAY)
      dt = dt / 5.0d0
      mflops = dfloat(NOPS)/dt * 1.0d-06
      write(6,615) n,dt,mflops,float(nops),compi
615   format(1x,i4,3x,d16.10,3x,d16.10,3x,e11.5,3x,e11.5)
      write(2,290) n, mflops
290   format(1x,i4,1x,e11.4)
      RETURN
      END
```

References

[FPS91a] FPS Computing. *Matrix Coprocessor Programmer's Guide*, April 1991. FPS Computing has been taken over in December 1991 by Cray Research Superservers, Inc.

[FPS91b] FPS Computing. *System 500 Matrix Coprocessor Overview*, February 1991. FPS Computing has been taken over in December 1991 by Cray Research Superservers, Inc.