Appl. Math. and Comp. Sci., 1993, vol.3, No.4, 777–790

NEW ALGORITHMS FOR THE MULTIPLICATION OF SERIES OF RECTANGULAR MATRICES AND THEIR PARALLEL IMPLEMENTATIONS

ROMAN WYRZYKOWSKI*, HENRYK PIECH*
JURI KANEVSKI**, VLADIMIR LEPECHA**

In the paper, two algorithms are proposed for the multiplication of series of rectangular matrices. Using the associative property of matrix multiplications, these algorithms allow us to decrease the complexity of computations in comparison with the direct algorithm. Parallel implementations of all the algorithms are then discussed for a linear array architecture. Finally, the efficiency of these parallel implementations is discussed.

1. Introduction

The methods of linear algebra make a basis for mathematical models in various fields of science and technology (Kung *et al.*, 1986; Rice, 1981). Particularly, in solving some problems from digital signal/image processing (Milovanovic and Stoicev, 1985; Stoicev *et al.*, 1990; Urguhart and Wood, 1984), a problem of practical interest is the multiplication of series of rectangular matrices

$$C = \prod_{i=1}^{s} A_i \tag{1}$$

where A_i is an $N_i \times M_i$ matrix. When multiplying these matrices in the natural order, we will face a large excess of computations. For example, when performing $C = \Big(\big((A_1[5\times 4] * A_2[4\times 6]) * A_3[6\times 3]\big) * A_4[4\times 2]\Big) * A_5[2\times 3]$, where sizes of input matrices are shown in brackets, 310 scalar multiplications have to be employed. However, using the associative property of matrix multiplications and changing the order of multiplications, the number of scalar operations can be considerably decreased. For example, for $C = \Big(A_1 * \big(A_2 * (A_3 * A_4)\big)\Big) * A_5$, only 166 scalar multiplications are sufficient.

Matrix–multiplications are computationally expensive because $O(N^3)$ multiply–add operations are required (Milovanovic and Stoicev, 1985; Urguhart and Wood, 1984) for multiplying two square matrices of order N. As a result, matrix multiplication algorithms require high computation rates to achieve acceptable execution times and to meet the real–time constraints of many signal/image processing applications. To

* Dept. Comput. Sci., Technical University of Częstochowa, Dąbrowskiego 73, 42–200 Częstochowa, Poland
** Dept. Comput. Sci., Kiev Polytechnic Institute, Pr. Pobedy 37, 252 056, Kiev, Ukraine

satisfy these requirements, parallel implementations of matrix multiplications have been studied and developed (Annaratone *et al.*, 1987; Kung, 1982; Kung, 1988; Kung *et al.*, 1986; Moreno and Lang, 1992; Quinton and Robert, 1991; Urguhart and Wood, 1984; Wyrzykowski, 1992; Wyrzykowski *et al.*, 1992).

The drawbacks of general–purpose parallel architectures have led to the development of application–specific architectures (Kung, 1982; Moreno and Lang, 1992) which are tailored to particular applications. Among these architectures there are application–specific processor arrays which can have different degrees of specialization. At one extreme there are algorithm–specific arrays specially designed for one particular algorithm, whereas class–specific arrays can be adapted (programmed) to a variety of algorithms. The choice between these possibilities and other intermediate ones depends on the particular requirements of the applications. Systolic–type arrays (Annaratone *et al.*, 1987; Kung, 1982; Moreno and Lang, 1992; Quinton and Robert, 1991; Wyrzykowski, 1992; Wyrzykowski *et al.*, 1992) are examples of application–specific architectures that have received much attention. Using massive pipelining, these arrays exploit the regularity inherent in many algorithms to achieve high performance while keeping local communications and low I/O requirements.

This paper is organized as follows. In Section 2, after describing the direct algorithm with the natural order of multiplications, two new algorithms are proposed. They use the associative property to decrease the complexity of computations. The next section deals with parallel implementations of all the algorithms. These implementations are oriented for class–specific arrays of systolic type with a fixed number of processing elements and linear structure of connections between them. The efficiency of these implementations is discussed in Section 4.

2. Algorithms for Multiplications of Series of Rectangular Matrices

2.1. Natural Order of Multiplications

The direct algorithm for the multiplication of a series of rectangular matrices in the natural order corresponds to the following program:

```
/* assignment of the first input matrix to an intermediate matrix $AM$ */
$AM := A_1$;
/* step–by–step multiplication of the series */
for $i := 2$ to $s$ do
    $AM := AM * A_i$;
$C := AM$                                                        (2)
```

Assuming that NMK multiply–add operations are required in the traditional algorithm (Rice, 1981) for the multiplication of an $N \times M$ matrix by an $M \times K$ matrix, we can estimate the time complexity W_1 of algorithm (2) in the following way

$$W_1 = N_1 \sum_{i=2}^{s} N_i M_i$$

We will use algorithm (2) as a building block in other algorithms.

2.2. Algorithm with Searching for a Single Minimum

The multiplication of form (1) is now decomposed into the following three steps:

1. Search for the row size N_j which is minimal among all the row sizes of input matrices A_i, where $i = 1, ..., s$. If more than one matrix A_i have the minimal size, then the first among them is chosen.
2. Compute an intermediate matrix
$$B_1 = \prod_{i=1}^{j-1} A_i \tag{3}$$
which is a product of those input matrices which are situated between the first input matrix and the matrix A_{j-1}.
3. To find the product of those input matrices which are situated between the matrix A_j and the last input matrix, compute an intermediate matrix
$$B_2 = \prod_{i=j}^{s} A_i \tag{4}$$
4. Compute the resultant matrix
$$C = B_1 * B_2 \tag{5}$$

As a result, for every multiplication of two adjacent matrices from the series, one of the matrices being multiplied will have the minimal row or column size. Algorithm (3)–(5), in spite of its simplicity, allows us to considerably reduce the amount of multiply–add operations. For $j \neq 1$, W_2 is now estimated by the following expression

$$W_2 = W_{2,1} + W_{2,2} + W_{2,3} = M_{j-1} \sum_{i=1}^{j-2} N_i M_i + N_j \sum_{i=j+1}^{s} N_i M_i + N_1 N_j M_s \tag{6}$$

where components $W_{2,1}$, $W_{2,2}$, $W_{2,3}$ correspond to computations performed in steps 2–4, respectively. Note that in equation (6), searching for a minimum matrix is not taken into account as being negligible. For example, when multiplying a series of such 10 matrices that each matrix A_{2k-1} or A_{2k} $(k = 1, ..., 5)$ has the size of 10×3 or 3×10 elements, respectively, we have $j = 2$, so that the number of required multiply–add operations reduces down to 1020 against 2700 operations in the case when the natural order is preserved. Algorithm (3)–(5) corresponds to the following program:

```
/* searching for the minimum */
min := $N_1$; $j := 1$;
for $i := 2$ to $s$ do if $N_i <$ min, then {min := $N_i$; $j := i$;}
$B_1 := I$; /* I is the unity matrix */
/* multiplication of form (3) to derive $B_1$ */
if $j > 1$, then
  {$AM := A_{j-1}$;
  for $i := j - 2$ downto 1 do
      $AM := A_i * AM$;
```

```
    $B_1 := AM;$}
/* multiplication of form (4) to derive $B_2$ */
$AM := A_j;$
for $i := j+1$ to $s$ do
    $AM := AM * A_i;$
$B_2 := AM;$
/* last step of the algorithm, when $C = B_1 * B_2$ */
$C = B_1 * B_2$                                          (7)
```

In algorithm (6), curly brackets {..} are equivalent to the **begin..end** construction in PASCAL.

2.3. Algorithm Searching for All the Minima

In order to improve the efficiency of algorithm (7), we will search for all the matrices with the minimum number of rows. Note that in the previous algorithm, a single matrix with the minimal row size has been searched for. After such a modification, the algorithm will consists of the following major steps:

1. Search for the size $N_{\min}$ which is minimal among all the row sizes of input matrices A_i, $i = 1, \ldots, s$.
2. Create the list $L_{\min}$ of all the matrices with $N_{\min}$ rows.
3. Create a new list $L_{\text{fin}} = \{t_j : 1 \le j \le r\}$ by removing from the list $L_{\min}$ those matrices which are right–hand adjacent to the matrices with the minimal size.
4. Compute intermediate matrices
$$C_j = \prod_{i=t_j}^{t_{j+1}-1} A_i, \qquad j = 1, \ldots, r-1$$
5. To find the product of all the previously determined intermediate matrices C_j, compute a new intermediate matrix
$$B_2 = \prod_{j=1}^{r-1} C_j.$$
6. Compute an intermediate matrix
$$B_3 = \prod_{i=t_r}^{s} A_i$$
in order to obtain the product of those input matrices which are situated between the end of the list L_{fin} and the last input matrix.
7. Compute an intermediate matrix
$$B_1 = \prod_{i=1}^{t_1-1} A_i$$
in order to find the product of those input matrices which are situated between the first input matrix and the beginning of the list L_{fin}.

8. Compute the resultant matrix $C = (B_1 * B_2) * B_3$.

The above–proposed algorithm corresponds to the following program:

```
/* searching for the size $N_{\min}$ */
min := $N_1$;
for $i := 2$ to $s$ do if $N_i <$ min then min := $N_i$;
/* creation of the list $L_{\min}$ */
$k := 1$;
for $i := 1$ to $s$ do if min $= N_i$ then $\{p_k := i;\ k := k + 1;\}$
/* creation of the list $L_{\text{fin}}$ */
$j := 1$;
for $i := 1$ to $k - 2$ do if $(p_{i+1} - p_i) \neq 1$ then $\{t_j := p_i;\ j := j + 1;\}$
$r := j - 1$;
/* deriving intermediate matrices $C_j$, where $j = 1,..,r - 1$ */
for $j := 1$ to $r - 1$ do {$AM := A_{t_j}$;
                             for $i := t_j$ to $t_{j+1} - 1$ do
                                $AM := AM * A_i$;
                             $C_j := AM$;}
/* multiplication of all the intermediate matrices $C_j$ to derive the matrix $B_2$ */
$AM := C_1$;
for $j := 2$ to $r - 1$ do
    $AM := AM * C_j$;
$B_2 := AM$;
/* deriving the matrix $B_3$ */
$AM := A_{t_r+1}$;
for $i := t_r + 2$ to $s$ do
    $AM := AM * A_i$;
$B_3 := AM$;
/* deriving the matrix $B_1$ */
if $t_1 = 1$ then $B_1 := I$ else {$AM := A_{t_1-1}$ ;
                                   for $i := t_1 - 2$ downto 1 do
                                      $AM := A_i * AM$;
                                   $B_1 := AM$;}
$C := (B_1 * B_2) * B_3$                                              (8)
```

Neglecting the creation of lists $L_{\min}$ and L_{fin}, the time complexity of algorithm (8) can be estimated by the following expression $(t_1 > 1)$

$$W_3 = W_{3,1} + W_{3,2} + W_{3,3} + W_{3,4} + W_{3,5}$$

$$= N_{\min} \sum_{j=1}^{r-1} \sum_{i=t_j+1}^{t_{j+1}-1} N_i M_i + (r-2) N_{\min}^3 + N_{\min} \sum_{i=1}^{t_1-2} N_i M_i \tag{9}$$

$$+ N_{\min} \sum_{i=t_r+1}^{s} N_i M_i + (N_1 N_{\min} N_{\min} + N_1 N_{\min} M_s)$$

where components $W_{3,1}$, $W_{3,2}$, $W_{3,3}$, $W_{3,4}$, $W_{3,5}$ correspond to computations performed in steps 4–8, respectively. Note that if $t_1 = 1$, then $B_1 = 0$, and $W_{3,5} = N_{\min} N_{\min} M_s$.

Algorithm (8) makes it possible to reduce the amount of computations in comparison with algorithm (7). For example, when multiplying the same series of 10 matrices as before, we have $L_{\min} \doteq L_{\text{fin}} = \{2,4,6,8,10\}$. As a result, it is sufficient to perform only 831 multiply–add operations. A drawback of algorithm (7) is the necessity to generate and store the intermediate matrices.

3. Parallel Implementations of the Algorithms

In order to give a workstation or a personal computer a very high performance on special–purpose computation intensive algorithms, the computer must be provided with some kind of hardware accelerator. This can be achieved in different ways. One of them consists in using a class–specific VLSI array which can be adapted (programmed) to a variety of algorithms. Some arrays of this type are already on the market (for example, linear systolic arrays Warp (Kung, 1982) and Matrix–1 (Foulser and Schreiber, 1987). Other arrays like Mismacs (Quinton and Robert, 1991) are being designed.

The general organization of the computer system with a class–specific VLSI array is schematically shown in Figure 1. The kernel of the system is a linear array of systolic type with K processing elements (PEs). Such an array has several important advantages over two–dimensional structure. First, it requires memory bandwidth which is independent of the size of the array. Secondly, a large structure can be constructed simply by concatenation of smaller arrays. Finally, a linear array allows us to minimize the amount of I/O channels because they are connected only with the first or/and the last PE.

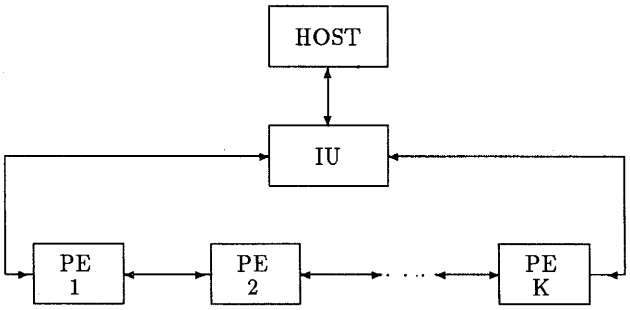

Fig. 1. Computer system with a class–specific VLSI linear array.

An important part of the system is also the interface unit IU, which is in charge of generating data and control to the array. This unit may contain a buffer memory for storing elements of input and intermediate matrices. Alternatively, the host computer can provide this memory.

The internal architecture of each PE is greatly influenced by a class of application algorithms. Since real class–specific VLSI arrays are basically oriented to signal processing and matrix algorithms, each PE of these arrays (Kung, 1982; Quinton and Robert, 1991) usually contains a local memory, some general–purpose and I/O registers, a multiplier, and an arithmetic/logical unit capable of performing elementary operations such as addition, subtraction, etc. All of these components are interconnected through a crossbar switch (Kung, 1982) or a system of buses (Quinton and Robert, 1991). To implement such a PE architecture, either signal processing chips, application specific integrated circuits (ASIC) or general–purpose chips like Transputers can be used.

The above–described organization of both the system as a whole and each PE covers that specific architecture which is required by the algorithms for the multiplication of series of rectangular matrices. The required PE architecture is shown in Figure 2 (without control paths), where R, SW, M and A are register, switch, multiplier and adder, respectively.

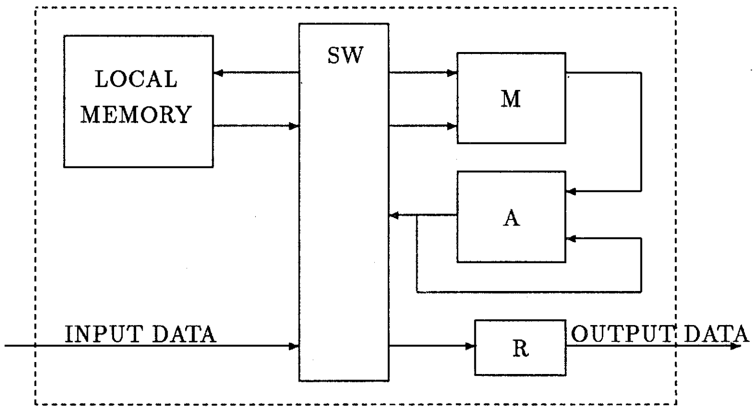

Fig. 2. PE architecture for matrix multiplication algorithms.

3.1. Parallel Implementation for the Natural Order of Multiplications

We will consider two variants of parallel implementation of algorithm (2): with unbounded or bounded parallelism. The first variant corresponds to that case when the number K of PEs is greater or equal to the row size N_1 for the first input matrix. The second variant corresponds to the case when $K \leq N_1$.

Variant 1.

The elements of the matrix A_1 are fed into PEs in such a way that after completing the loading, the i–th PE contains the i–th row of A_1. Then the elements of the matrix A_2 are pipelined in the columnwise order between PEs of the linear array. As a result of this pipelining, the intermediate matrix $A' = A_1 * A_2$ is formed in such a way that the i–th row of A' is obtained in the i–th PE. Next, the elements of A_3 are pipelined between PEs, and the intermediate matrix $A'' = A' * A_3 = A_1 * A_2 * A_3$ is computed, etc. Therefore, not taking into account the unloading of the resultant matrix C from PEs of the array, the execution time T for algorithm (2) is

$$T = N_1 M_1 + \sum_{i=2}^{s} Q_{A_i} = \sum_{i=1}^{s} Q_{A_i} \tag{10}$$

where $Q_{A_i} = N_i M_i$ is the number of elements in the matrix A_i. Moreover, we assume that during one step a single multiply–add operation is performed.

Variant 2.
Having been fed into corresponding PEs, the first K rows of A_1–matrix are multiplied by all the columns of A_2–matrix, which are pipelined between PEs. In this way, the first K rows of the matrix $A' = A_1 * A_2$ are obtained. Next, all the columns of A_3 are pipelined between PEs, and the first K rows of the matrix $A'' = A_1 * A_2 * A_3$ are computed, etc. As a result, we obtain the first K rows of the resultant matrix C. The other rows of matrix C are computed in a similar way. The execution time for this variant can be estimated by the following formula

$$T = Q_{A_1} + L_1 \sum_{i=2}^{s} Q_{A_i} \tag{11}$$

where $L_1 = [N_1/K]$, and $[x]$ is the nearest integer greater or equal to x.

3.2. Parallel Implementation of the Algorithm Searching for a Single Minimum

While implementing this algorithm, the matrix A_{j-1} is first fed into PEs in the columnwise order. Then the rows of the matrix A_{j-2} are pipelined between PEs. As a result of this pipelining, the columns of the matrix $A' = A_{j-2} * A_{j-1}$ are accumulated in PEs of the array. Next, the rows of A_{j-3} are pipelined between PEs in order to provide the accumulation of columns of the matrix $A'' = A_{j-3} * A'$, etc. In this way, the intermediate matrix $B_1 = \prod_{i=1}^{j-1} A_i$ is computed. Its elements are saved in the buffer memory. Next, the first A_j matrix of the series having the minimum row size is fed into PEs in the rowwise order, and then multiplied by columns of the subsequent matrices. These multiplications are performed in the same way as those described in subsection 3.1. As a result, the intermediate matrix $B_2 = \prod_{i=j}^{s} A_i$ is computed, and saved in the buffer. In the last stage, the resultant matrix C is computed by multiplying B_1 and B_2.

As in the case of the natural order of multiplications, we will consider two variants of implementing this algorithm: with unbounded or bounded parallelism. The first case corresponds to $K \geq N_j, N_1$, while the other one corresponds to $K < N_j$ or $K < N_1$. The execution time for both variants can be estimated as follows $(j > 2)$

$$T = T_1 + T_2 + T_3 = \sum_{i=1}^{j-1} Q_{A_i} + \sum_{i=j}^{s} Q_{A_i} + N_j(N_1 + M_s) = \sum_{i=1}^{s} Q_{A_i} + N_j(N_1 + M_s) \tag{12}$$

$$T = T_1 + T_2 + T_3 = \Big(Q_{A_{j-1}} + L_j \sum_{i=1}^{j-2} Q_{A_i}\Big) + \Big(Q_{A_j} + L_j \sum_{i=j+1}^{s} Q_{A_i}\Big)$$

$$+ N_j(N_1 + L_1 M_s) = Q_{A_j} + Q_{A_{j-1}} + L_j \Big(\sum_{i=1}^{j-2} Q_{A_i} + \sum_{i=j+1}^{s} Q_{A_i}\Big) + N_j(N_1 + L_1 M_s) \tag{13}$$

where $L_j = [N_j/K]$, and components T_1, T_2, T_3 have the same meaning as those in equation (6). Note that if $j = 2$, then $T_1 = 0$ in equations (12)–(13).

3.3. Parallel Implementation of the Algorithm with Searching for all the Minima

For the given linear architecture with a fixed number of PEs, algorithms (8) should be modified at the stage of searching for the "minimal" matrices. Rather than to a matrix having the minimal row size, this definition is now referred to such a matrix for which the time required to multiply it by an arbitrary matrix is minimal. In searching for such matrices, it is necessary to determine the interval of sizes N_i such that $L_{\min}K < N_i \leq (L_{\min}+1)K$, where $L_{\min} = [N_{\min}/K]$. Indices i of all matrices whose size N_i falls into the above interval compose the list $L_{\min}$ of minimal matrices, which is formed in step 2 of the algorithm. The remainder of the computations closely corresponds to algorithm (8). Its execution time for the variant with unbounded parallelism can be estimated by the following formula ($t_1 > 2$ and $t_r < s$)

$$T = T_1 + T_2 + T_3 + T_4 + T_5 = \sum_{i=t_1}^{t_r-1} Q_{A_i} + (r-1)N_{\min}^2 + \sum_{i=1}^{t_1-1} Q_{A_i} + \sum_{i=t_r}^{s} Q_{A_i}$$

$$+(N_1 N_{\min} + N_{\min}N_{\min} + N_{\min}M_s) = \sum_{i=1}^{s} Q_{A_i} + N_{\min}(rN_{\min} + N_1 + N_s) \qquad (14)$$

where components T_1–T_5 have the same meaning as those in equation (9), and r is the number of "minimal" matrices in the list $L_{\text{fin}} = \{t_j : j = 1,..,r\}$. For the variant with bounded parallelism, when $K < N_{\min}$ or $K < N_1$, we have the following:

$$T = \sum_{i=1}^{5} T_i = \Big(\sum_{j=1}^{r-1} Q_{A_{t_j}} + L_{\min} \sum_{j=1}^{r-1}\sum_{i=t_j+1}^{t_{j+1}-1} Q_{A_i}\Big) + \big(N_{\min}^2 + N_{\min}N_{\min}^2(r-2)\big)$$

$$+\Big(Q_{A_{t_1-1}} + L_{\min}\sum_{i=1}^{t_1-2} Q_{A_i}\Big) + \Big(Q_{A_{t_r}} + L_{\min}\sum_{i=t_r+1}^{s} Q_{A_i}\Big) \qquad (15)$$

$$+\Big(N_1 N_{\min} + L_1(N_{\min}N_{\min} + N_{\min}M_s)\Big)$$

Note that if $t_r = s$, then in formula (15) we have $T_4 = 0$, and $T_5 = N_1 N_{\min} + L_1 N_{\min} N_{\min}$. Also, if $t_1 = 1$, then $T_3 = 0$, and $T_5 = N_1 N_{\min} + L_1 N_{\min} M_s$. Finally, if $t_1 = 2$, then $T_3 = 0$.

4. Comparison of the Algorithms

To compare the efficiency of parallel implementation of algorithms (2), (7), and (8) we will base on the estimations from the derived formulae (10)–(15) for the algorithm execution time. They allow us to conclude first that in the case of unbounded parallelism, the direct algorithm (2) is more efficient than the proposed ones. This surprising conclusion can be explained by the necessity of computing some intermediate matrices in algorithms (7), (8). However, this case appears to be important only from the theoretical point of view. Practically, we usually deal (Kung, 1988; Kung *et al.*, 1986) with bounded parallelism. In this case, the advantage of the proposed algorithm (7) over the direct algorithm (2) results from the fact that the majority of matrices are pipelined through the array not $[N_1/K]$ times but only $[N_{\min}/K]$ times. At the same time, the advantage of algorithm (7) can be explained by the fact that not only two but all the "minimal" matrices are pipelined through the array only once. However, the necessity to compute all the intermediate matrices decreases this advantage.

Considering the case of bounded parallelism, in Tables 1–4 we estimate the execution time required to perform these algorithms on a linear array with K PEs for a series of 15 matrices. The sizes of the matrices being multiplied are shown in the upper part of each table, while figures placed in the lower part have been obtained using formulae (11), (13), (15). These results are illustrated graphically in Figures 3–6. However, for better understanding, we show on the graphs not the algorithm execution time, but a much more reasonable criterion which is the efficiency E of a parallel algorithm with respect to the best sequential algorithm. Following Ortega (1988), we have

$$E = W_B/(KT)$$

where $W_B = \min\{W_1, W_2, W_3\}$, and T is the execution time for an algorithm implemented on K PEs.

These tables and graphs allow us to conclude that the natural order of multiplications becomes justified when the row size N_1 of the first matrix is minimal in comparison with the matrices of a series (see Tab. 1 and Fig. 3), or N_1 is close to $N_{\min}$ (see Tab. 2 and Fig. 4). In the second case, however, the number K of PEs should not be too small in comparison with N_1. In other cases, it is advisable to use one of the proposed algorithms searching for "minimal" matrices. The algorithm searching for all the "minimal" matrices substantially increases the efficiency of computations only in the case when a series features a sufficiently large number of "minimal" matrices and the minimal $N_{\min}$ size of matrices considerably exceeds the number K of PEs (see Tab. 4 and Fig. 6). If these conditions are not satisfied, then algorithm (8) might even lead to a certain decrease in the efficiency of computations as compared with the algorithm based on searching for a single "minimal" matrix (see Tab. 3 and Fig. 5). Taking into account that the above-described linear architecture allows us to perform any of these three algorithms, we can always choose the most efficient algorithm depending on the set of matrices being multiplied.

Acknowledgment

The authors gratefully acknowledge the valuable comments of the anonymous reviewer in improving the first version of the paper.

References

Annaratone M., Arnould E., Gross T., Kung H.T., Lam M., Menzilcioglu O. and Webb J.A. (1987): *The Warp computer: architecture, implementation, and performance.* — IEEE Trans. Comput., v.C–36, No.12, pp.1523–1537.

Foulser D.E. and Schreiber R. (1987): *The Saxpy Matrix–1: A general purpose systolic computer.* — Computer, v.20, No.7, pp.35–43.

Kung H.T. (1982): *Why systolic architectures ?.* — Computer, v.15, No.1, pp.37–46.

Kung S.Y. (1988): *VLSI Array Processors.* — Englewood Cliffs: Prentice Hall.

Kung S.Y., Whitehouse H.J. and Kailath T. (Eds.) (1986): *VLSI and Modern Signal Processing.* — New York: Prentice–Hall.

Milovanovic I.Z. and Stoicev M.K. (1985): *Matrix multiplication in computer graphics.* — Proc. 30–th Intern. Wiss. Coll., TH Ilmenau, (Germany), pp.151–154.

Moreno J.H. and Lang T. (1992): *Matrix Computations on Systolic–Type Arrays.* — Boston: Kluwer Academic Publishers.

Ortega J.M. (1988): *Introduction to Parallel and Vector Solution of Linear Systems.* — New York: Plenum Press.

Quinton P. and Robert Y. (1991): *Systolic Algorithms and Architectures.* — Englewood Cliffs: Prentice Hall.

Rice J.R. (1981): *Matrix Computations and Mathematical Software.* — New York: McGraw–Hill Book Comp.

Stoicev M.K., Milovanovic E.I. and Milovanovic I.Z. (1990): *An algorithm for multiplication of concatenated matrices.* — Parallel Computing, v.13, p.211–223.

Urguhart R.B. and Wood D. (1984): *Systolic matrix and vector multiplication methods for signal processing.* — IEE Proc., Part E, v.131, No.6, pp.623–631.

Wyrzykowski R. (1992): *Processor arrays for matrix triangularisation with partial pivoting.* — IEE Proc., Pt.E, v.139, No.2, pp.165–169.

Wyrzykowski R., Kanevski Ju.S. and Ovramenko S.G. (1992): *Dependence graph transformations in the design of processor arrays for matrix multiplications.* — Microprocessing and Microprogramming, v.35, pp.539–544.

Tab. 1.

N_i	20	40	40	60	80	20	100	60	20	50	40	20	60	40	60
M_i	40	40	60	80	20	100	60	20	50	40	20	60	40	60	50

K	natural order	a single minimum	all the minima
2	324 800	324 800	309 800
5	130 400	130 400	127 400
10	65 600	65 600	66 600
15	65 600	65 600	66 600
20	33 200	33 200	36 200

Tab. 2.

N_i	60	80	80	50	80	60	80	50	80	100	50	100	50	100	80
M_i	80	80	50	80	60	80	50	80	100	50	100	50	100	80	100

K	natural order	a single minimum	all the minima
2	2 284 800	1 981 000	1 847 500
5	916 800	799 000	755 500
10	460 800	405 000	391 500
20	232 800	244 400	250 800
40	156 800	166 600	170 100
60	80 800	88 800	98 800

Tab. 3.

N_i	100	100	100	100	40	80	20	60	20	80	20	40	100	20	100
M_i	100	100	100	40	80	20	60	20	80	20	40	100	20	100	100

K	natural order	a single minimum	all the minima
2	2 670 000	708 800	697 600
5	1 074 000	286 400	284 800
10	542 000	145 600	147 200
15	382 400	139 600	140 000
20	276 000	75 200	78 400
25	222 800	73 200	76 000
40	169 600	71 200	73 600

Tab. 4.

N_i	200	200	200	200	50	100	50	800	60	500	50	600	60	800	50
M_i	200	200	200	50	100	50	800	60	500	50	600	60	800	50	400

K	natural order	a single minimum	all the minima
2	41 740 000	13 075 000	11 292 500
5	16 720 000	5 245 000	4 587 500
10	8 380 000	2 635 000	2 352 500
15	5 878 000	2 073 000	1 654 500
20	4 210 000	1 551 000	1 278 500
25	3 376 000	1 069 000	1 011 500
40	2 125 000	1 009 000	880 000

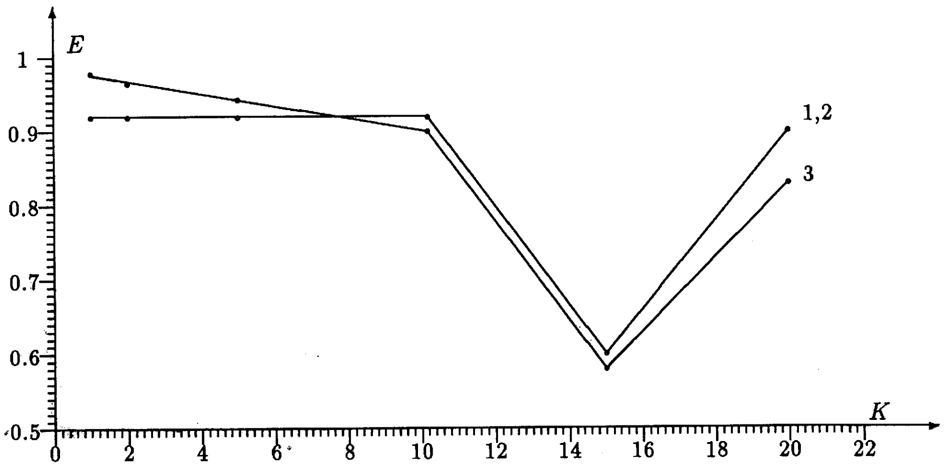
E
1
0.9
0.8
0.7
0.6
0.5
0
2
4
6
8
10
12
14
16
18
20
22
K
1,2
3

Fig. 3.

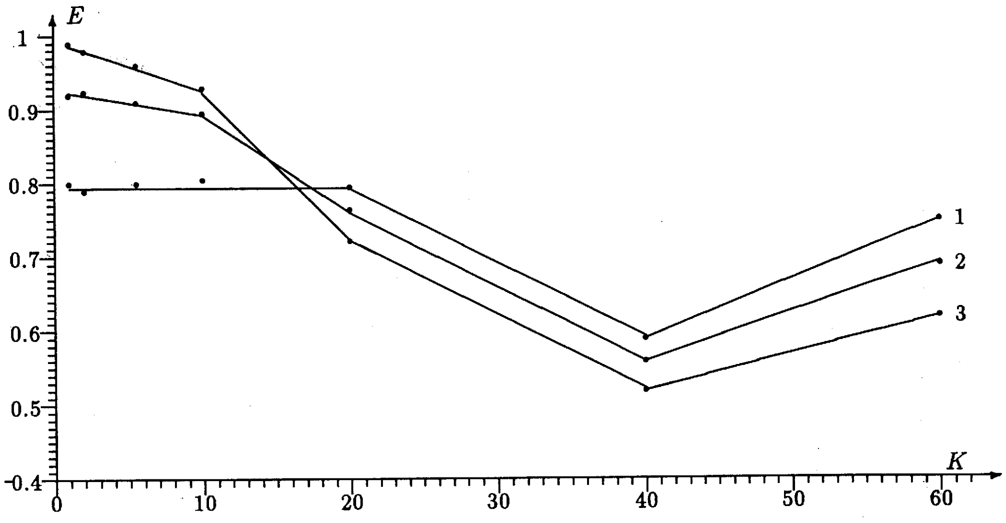
E
1
0.9
0.8
0.7
0.6
0.5
-0.4
0
10
20
30
40
50
60
K
1
2
3

Fig. 4.

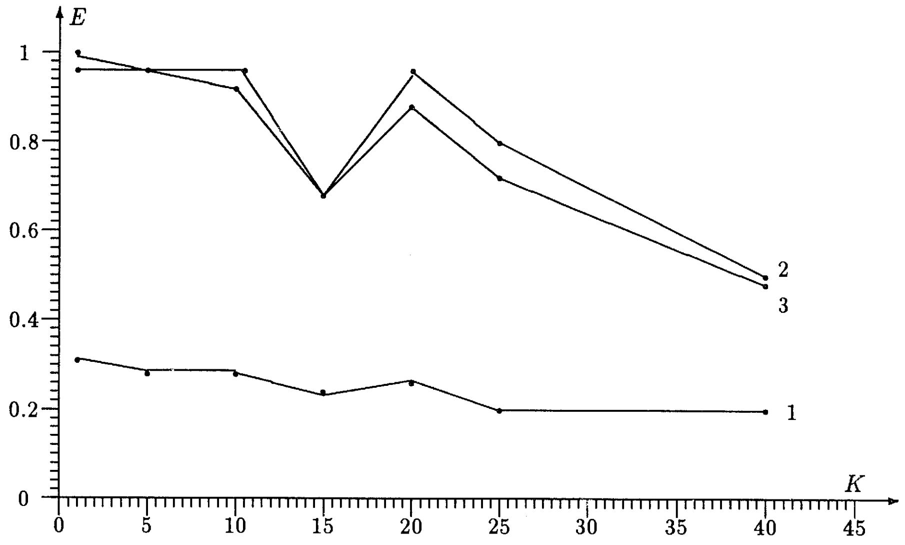

Fig. 5.

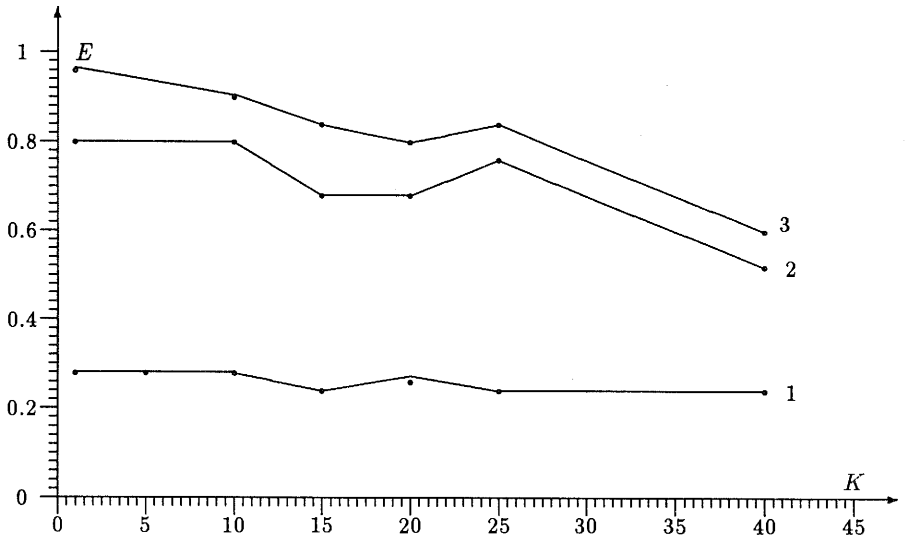

Fig. 6.

Received March 18, 1993
Revised December 17, 1993