FAST MATRIX MULTIPLICATION ALGORITHMS ON MIMD ARCHITECTURES

B.DUMITRESCU [*][†] , J.L.ROCH [*] AND D.TRYSTRAM [*]

Abstract. Sequential fast matrix multiplication algorithms of Strassen and Winograd are studied; the complexity bound given by Strassen is improved. These algorithms are parallelized on MIMD distributed memory architectures of ring and torus topologies; a generalization to a hyper-torus is also given. Complexity and efficiency are analyzed and good asymptotic behaviour is proved. These new parallel algorithms are compared with standard algorithms on a 128-processor parallel computer; experiments confirm the theoretical results.

Key Words. matrix multiplication, Strassen's algorithm, Winograd's algorithm, parallel algorithms.

AMS(MOS) subject classification. 65-04, 65F30, 65Y05.

1. Introduction.

Matrix multiplication is an operation that often occurs in linear algebra algorithms; moreover, the trend to express problems in terms of block of matrices (level-3 operations, see [11]), due especially to the memory organization of supercomputers, has instigated new studies on matrix multiplication problem.

In 1969, Strassen proposed a new way to multiply two n-dimensional matrices [19]. This method is based on a substitution of multiplications by additions in the usual formulae (inner products), which reduces the number of multiplications from $O(n^3)$ to $O(n^{\log_2 7})$. Improvements have been developed from Strassen's algorithm [20] and many other algorithms have been discovered [17]. At this time, the best upper bound known is $O(n^{2.37})$ [7].

Among all these methods, Strassen's [19] and Winograd's [20] have been shown to be of some practical interest for sequential implementations [4, 6], but disregarded because their numerical properties are weaker than those of the standard algorithm.

Fast matrix multiplication methods lead to theoretical optimal parallel algorithms with a polynomial bounded number of processors [7]. However, from a practical point of view, only few implementations have been proposed [2, 3, 16]; all of these are for SIMD architectures. One of the main reasons is that the basic formulae used in these methods are not symmetric or regular and then cannot easily, and thus efficiently, be parallelized, while the standard algorithm has a structure very appropriate for parallelism. But the cited papers have proved that efficiency can be obtained and that, from numerical point of view, the fast algorithms, implemented with care, may be of value. Of course, fast algorithms will be better only for matrix dimensions of the order of hundreds.

We propose in this paper parallelizations which minimize the depth of the precedence graph using 7 processors on a ring architecture or 49 on a torus. This study leads to practical efficient implementations: on one hand, fast methods are used locally as the basic sequential routine on block submatrices, and, on the other hand, it leads to a parallel task repartition on a ring or on a torus.

From this study, we give asymptotically optimal algorithms, whose complexity is $O((\frac{n}{2^k})^{\log_2 7})$ on a k hyper-torus with 7^k processors. We present some experiments on a 128 Transputer network.

[*] LMC-IMAG, 46, Av. F. Viallet, 38031 Grenoble, France.

[†] On leave from the Polytechnical Institute of Bucarest, Romania; supported by a TEMPUS grant.

This paper is structured as follows. In section 2 fast sequential algorithms of Strassen and Winograd are analyzed; Strassen's bound of $4.7n^{\log_2 7}$ is improved. Section 3 and 4 deal respectively with the precedence task graph of these algorithms and task allocation on ring and hyper-torus topologies; complexity formulae are derived and asymptotic efficiency equal to 1 is proved. In section 5 a short review of standard algorithms on ring and torus is given; a mixed algorithm is obtained from Strassen's and standard algorithms. Experimental results are presented in section 6, that show the good behaviour of the new parallel algorithms, compared to standard ones, for matrices of size greater than 200×200.

2. Fast sequential methods for matrix multiplication. Let A and B be two square matrices of size $n \times n$. If not explicitly specified, we assume that n is a power of 2. We denote by $a_{i,j}$ the element of row i, column j, of matrix A. Denote by $MM(n)$ the problem of computing the matrix multiplication $C = A \cdot B$. We call the explicit use of the formula $c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}$ the *standard method* for solving $MM(n)$.

For any matrix X of size $n \times n$, we define the partition $X = \begin{bmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{bmatrix}$, where $X_{i,j}$ are square blocks of size $\frac{n}{2} \times \frac{n}{2}$.

2.1. Strassen's method. The first fast method for matrix multiplication was discovered by Strassen in 1969 [19]. It is based on the following formulae:

$$C = A \cdot B = \begin{bmatrix} M_0 + M_1 + M_2 - M_3 & M_3 + M_5 \\ M_2 + M_4 & M_0 - M_4 + M_5 + M_6 \end{bmatrix} \tag{1}$$

$$\begin{array}{rr} M_0 = (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2}) & M_3 = (A_{1,1} + A_{1,2}) \cdot B_{2,2} \\ M_1 = (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2}) & M_4 = (A_{2,1} + A_{2,2}) \cdot B_{1,1} \\ M_2 = A_{2,2} \cdot (B_{2,1} - B_{1,1}) & M_5 = A_{1,1} \cdot (B_{1,2} - B_{2,2}) \\ \multicolumn{2}{c}{M_6 = (A_{2,1} - A_{1,1}) \cdot (B_{1,1} + B_{1,2})} \end{array}$$

The algorithm above is recursively applied for each of the 7 new $MM(\frac{n}{2})$ subproblems, halving each time the dimension of the blocks. Moreover, 18 additions of matrices of dimension $\frac{n}{2}$ are involved. The arithmetic complexity is:

$$t_m(n) = n^{\log 7} \qquad t_a(n) = 6n^{\log 7} - 6n^2 \tag{2}$$

where $t_m(n)$ and $t_a(n)$ respectively denote the number of multiplications and the number of additions; all logarithms in this paper are in basis 2, so we will simply write log instead of $\log_2$.

2.2. Winograd's method. In 1973, Winograd [20] improves Strassen's method, reducing the number of additions to only 15; it is also proved that this is the minimum, for any algorithm of rank 7 (using 7 multiplications instead of 8, as the standard) for $MM(2)$ [17]. The following formulae describe the algorithm:

$$C = A \cdot B = \begin{bmatrix} M_0 + M_1 & M_0 + M_4 + M_5 - M_6 \\ M_0 - M_2 + M_3 - M_6 & M_0 + M_3 + M_4 - M_6 \end{bmatrix} \tag{3}$$

$$\begin{array}{rr} M_0 = A_{1,1}.B_{1,1} & M_3 = (A_{1,1} - A_{2,1}).(B_{2,2} - B_{1,2}) \\ M_1 = A_{1,2}.B_{2,1} & M_4 = (A_{2,1} + A_{2,2}).(B_{1,2} - B_{1,1}) \\ M_2 = A_{2,2}.(B_{1,1} - B_{1,2} - B_{2,1} + B_{2,2}) & M_5 = (A_{1,1} + A_{1,2} - A_{2,1} - A_{2,2}).B_{2,2} \\ \multicolumn{2}{c}{M_6 = (A_{1,1} - A_{2,1} - A_{2,2}).(B_{1,1} + B_{2,2} - B_{1,2})} \end{array}$$

The complexity is similar with (2):

$$t_m(n) = n^{\log 7} \qquad t_a(n) = 5n^{\log 7} - 5n^2 \tag{4}$$

2.3. Designing a good sequential algorithm. In this section, we wish to derive an efficient algorithm for matrix multiplication. As the target processors are usual microprocessors, on which floating-point addition and multiplication have approximately equal execution time, we may consider the complexity of each of the three algorithms (standard, Strassen and Winograd) as the sum of the number of multiplications $t_m(n)$ and the number of additions $t_a(n)$:

$$T_{Stand}(n) = 2n^3 - n^2 \qquad T_{Strassen}(n) = 7n^{\log 7} - 6n^2 \qquad T_{Wino}(n) = 6n^{\log 7} - 5n^2$$

The idea is to determine a good tradeoff between Strassen's or Winograd's algorithms and the standard one; the recursion will be performed until a cutoff dimension n_0 is obtained, after which the standard algorithm is used; such an algorithm will be called *mixed.*

If n and n_0 are powers of 2, the best cutoff dimension is $n_0 = 8$, both for Strassen's and Winograd's algorithms (see also [12]).

In the general case where matrix size is not a power of 2, recursions take place down to a dimension q less or equal to n_0, i.e. in the interval $[[\frac{n_0}{2}] + 1,\ n_0]$. In the original paper of Strassen [19] it is proved that there exists an n_0 such that:

$$T_{Strassen\_mix}(n) < 4.7\, n^{\log 7},\ \forall n > n_0 \tag{5}$$

By another way, we give below a better bound. It is immediate that $\forall n,\ \exists q \in [[\frac{n_0}{2}] + 1,\ n_0]$ and $\exists p > 0$ integer, so that $(q-1) \cdot 2^p < n \le q \cdot 2^p$; note that p is the minimal number of recursion steps in order to obtain a block dimension less or equal to n_0. For a proper recursion, initial matrices are padded with zeroes, up to size $q \cdot 2^p$. Using the recursion formula $T(n) = 7T(\frac{n}{2}) + 18\frac{n^2}{4}$, with the initial condition $T(q) = 2q^3 - q^2$, it follows that:

$$T_{Strassen\_mix}(n) = \frac{2q^3 + 5q^2}{q^{\log 7}}(q \cdot 2^p)^{\log 7} - 6(q \cdot 2^p)^2 < \frac{2q^3 + 5q^2}{(q-1)^{\log 7}}\, n^{\log 7} = c_q\, n^{\log 7}$$

The problem to be solved is:

$$\min_{n_0} \max_{q \in [[\frac{n_0}{2}]+1, n_0]} c_q$$

Listing the values of c_q, it is easy to make the best choice: $c_{21} = 4.62$, $c_{40} = 4.644$. So, the optimal n_0 is 40, and the corresponding bound, which improves that of (5):

$$T_{Strassen\_mix}(n) < 4.65\, n^{\log 7},\ \forall n > 40$$

As an observation, Strassen computed his bound implicitely using the interval [16,31], which, as can be seen, is not the best.

An analogous computation for Winograd leads to $n_0 = 36$, $T_{Wino\_mix}(n) < 4.56\, n^{\log 7}$; we did not find any reference to a similar result.

These results were obtained based only upon arithmetic complexity reasons. Other factors, such as implementation complexity or numerical stability are discussed in the sequel. Recall once again that the analysis above is based on the hypothesis that the ratio of floating-point multiplication time vs. floating-point addition time is equal to 1; if the ratio is greater than 1, the value of n_0 decreases.

	Standard	Winograd				Strassen		
n		n_0				n_0		
		8	16	32	64	16	32	64
32	0.22	0.29	0.24	*0.22*	0.22	0.25	0.22	*0.22*
48	0.75	1.11	0.83	*0.74*	0.75	0.90	0.76	*0.75*
64	1.76	2.19	1.78	*1.68*	1.76	1.90	*1.72*	1.76
80	3.43	5.37	3.80	3.29	*3.20*	4.22	3.47	*3.27*
96	5.91	8.05	6.08	5.45	*5.44*	6.67	5.72	*5.54*
128	13.97	15.84	13.01	*12.28*	12.65	14.03	*12.75*	12.82
192	47.01	57.52	43.76	39.35	*39.28*	48.23	41.60	*40.34*
256	111.27	112.97	93.15	*88.03*	90.67	100.99	*92.04*	92.56
384	374.99	407.37	311.03	280.17	*279.72*	343.91	297.49	*288.66*
512	888.23	799.24	660.47	*624.66*	643.14	718.10	*655.43*	659.08

TABLE 1
Timings (in seconds) for sequential algorithms on a transputer T800

2.4. Experiments. In order to find the best cutoff dimension n_0, we performed a series of experiments on implementations of standard, Winograd and Strassen algorithms. All programs were written in C, with double precision variables, compiled with the C3L compiler and executed on a single transputer of a supernode; for more information on this type of computer, as well as on available software tools, see [1] and section 6 of the present paper. The results are reported in table 1.

From this table, we can remark that the optimal cutoff dimension for Winograd's algorithm is $32 \leq n_0 < 64$; as an example, for the experiments above, the value $n_0 = 48$ will collect the best times, for the chosen dimensions n. The difference between the theoretical result ($n_0 < 16$, as experiments are made only for unpadded matrices) and the experimental one is due to the great overhead due to implementation: dynamic memory allocation, function calls, more integer arithmetic. It is recommended, for numerical stability, to choose, as far as possible, a greater value; Bailey [2] uses 127, IBM's ESSL library even more.

As expected, Strassen's method is only a little worse than Winograd's and has a greater n_0.

Both fast methods become much better than the standard one beginning at matrix dimension of the order of 100 (speed improvement: about 10%).

2.5. A brief discussion on numerical stability. One must never forget numerical stability; fast algorithms often have worse numerical properties than the standard ones. For the $MM(n)$ problem we refer to the original work of Brent [4], rediscovered by Higham [12]. We remind only the absolute error for Strassen's method:

$$\|E\| \leq \left[\left(\frac{n}{n_0} \right)^{\log 12} (n_0^2 + 5n_0) - 5n \right] u \|A\| \cdot \|B\| + O(u^2) \tag{6}$$

where $\|A\| = \max_{i,j} |a_{i,j}|$ and u the unit roundoff of the machine; for Winograd's method the error bound is similar, but with an exponent of $\log 18 \approx 4.17$ replacing $\log 12 \approx 3.585$, while for the standard algorithm:

$$\|E\| \leq nu\|A\| \cdot \|B\| + O(u^2)$$

The standard algorithm is obviously more stable than the fast ones. It can be seen that the bound in (6) is improved by a bigger n_0. A way to get error within an acceptable limit is to permit only few levels of recursion (as in the IBM ESSL library [13] - four levels), i.e. keeping constant and small $\frac{n}{n_0}$, which has the biggest exponent; this implies also a limitation of speed improvement, but a compromise is necessary.

2.6. The problem of memory space. The Parallel Computation Thesis states (see [14]) that parallel time is polynomially equivalent to sequential space. From a practical point of view, the data dependency graph in a sequential algorithm may be seen as a precedence graph, despite different interpretations. Minimizing memory space is important because on one hand it leads to an efficient sequential algorithm, which may be used by each processor of a parallel computer, and on another hand it gives an insight into the efficiency of the parallelization. The main characteristic of fast sequential algorithms compared to a standard algorithm, is to reduce time complexity, while increasing space complexity. The parallelization of such a fast algorithm is expected to lead to a slower algorithm, but requiring fewer processors. Thus, it is important to study precisely the space complexity needed for fast matrix multiplication.

Unfortunately, the problem of determining the minimal memory space needed for the execution of an algorithm is NP-complete. It can be found, under the name of REGISTER SUFFICIENCY, in [10].

The data dependency graph for Strassen's or Winograd's algorithms having a small number of vertices (25 and 22, respectively, as can be seen in the next section), the problem of minimizing memory space was solved by an exhaustive, but carefully designed, program. The extra memory (over that necessary for the result) is $\frac{2}{3}n^2$ cells for Strassen's algorithm and n^2 cells for Winograd's algorithm; the first result is also reported in [2] from Kreczmar [15].

3. Parallel complexity study. In the sequel, we propose parallel versions of both (Winograd and Strassen) fast algorithms for $MM(n)$; since they are defined recursively, we first consider the parallelization on one level consisting of reducing the matrix dimension from n to $\frac{n}{2}$, i.e. using equations (1), (3) only once.

The parallel implementation of fast $MM(n)$ methods is based on the notion of precedence task graph introduced in [5]. An elementary task is defined as an indivisible unit of work, specified only in terms of its external behaviour (inputs, outputs, execution time, etc.).

The parallel complexity study principle consists of splitting the program into elementary tasks, whose order of execution is directed by precedence constraints. According to the literature, the precedence constraint relation is denoted $\prec$; if, for tasks $T_i, T_j,\ T_i \prec T_j$, then the execution of task T_j can begin only after the completion of task T_i. The precedence task graph (which is a DAG) is built directly from these constraints; if $T_i \prec T_j$, then a path exists from T_i to T_j.

We presume a coarse grain computation, i.e. a task contains block (level-3 BLAS) operations (not matrix element level operations).

3.1. Precedence task graph for Winograd's method. The elementary tasks are the following, as directly resulting from equations (3) (there are 7 multiplication tasks and 15 addition tasks):

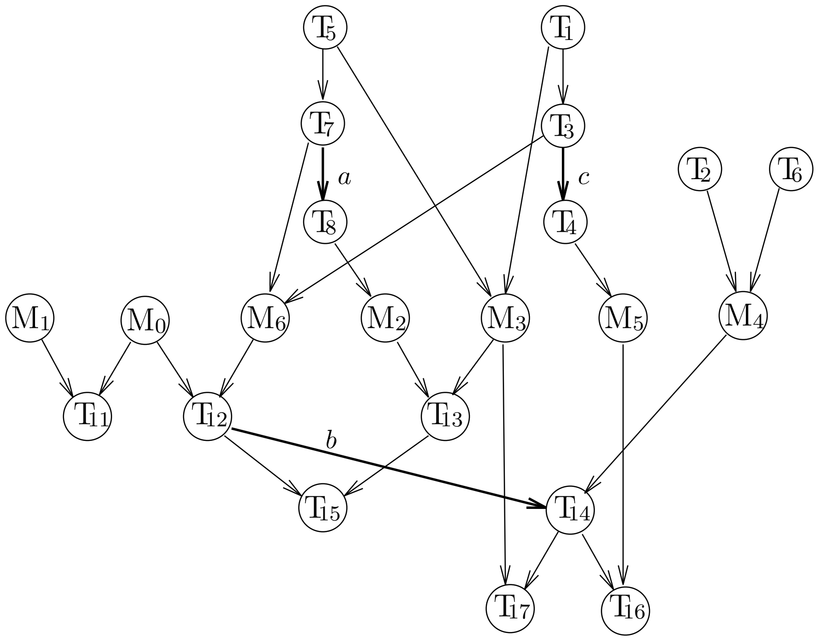


FIG. 1. *Precedence task graph for Winograd's algorithm*

Task $T_1 = A_{1,1} - A_{2,1}$		
Task $T_2 = A_{2,1} + A_{2,2}$	Task $M_0 = A_{1,1} \cdot B_{1,1}$	Task $T_{11} = M_0 + M_1$
Task $T_3 = T_1 - A_{2,2}$	Task $M_1 = A_{1,2} \cdot B_{2,1}$	Task $T_{12} = M_0 - M_6$
Task $T_4 = T_3 + A_{1,2}$	Task $M_2 = A_{2,2} \cdot T_8$	Task $T_{13} = M_3 - M_2$
Task $T_5 = B_{2,2} - B_{1,2}$	Task $M_3 = T_1 \cdot T_5$	Task $T_{14} = T_{12} + M_4$
Task $T_6 = B_{1,2} - B_{1,1}$	Task $M_4 = T_2 \cdot T_6$	Task $T_{15} = T_{12} + T_{13}$
Task $T_7 = T_5 + B_{1,1}$	Task $M_5 = T_4 \cdot B_{2,2}$	Task $T_{16} = T_{14} + M_5$
Task $T_8 = T_7 - B_{2,1}$	Task $M_6 = T_3 \cdot T_7$	Task $T_{17} = T_{14} + M_3$

and lead to the precedence task graph of figure 1; hereafter the notation for the task name and for its output is the same.

It is well known that, in a precedence task graph, width gives information about the necessary (or maximum) number of processors and depth about the parallel execution time. We try to minimize the depth (the number of tasks in the longest path in the graph), in order to obtain a shorter execution time. The depth of this graph is 6; in order to reduce it to 5, we have to 'cut' the edges denoted by a and b in figure 1. We also cut edge c for the following reason: there are 7 multiplication tasks of time $O((\frac{n}{2})^{\log 7})$, whose execution time is much greater than that for the other tasks; we intend to obtain a scheduling which implies parallel execution for all these 7 tasks; for a minimal global execution time, each longest path has to be divided by the multiplication stage in the same way.

The cost of this reduction is the introduction of some redundant work; there are now 18 (instead of 15) addition tasks of time $O((\frac{n}{2})^2)$. Tasks that have changed (compared to the previous version of the precedence graph) and the new ones are listed above:

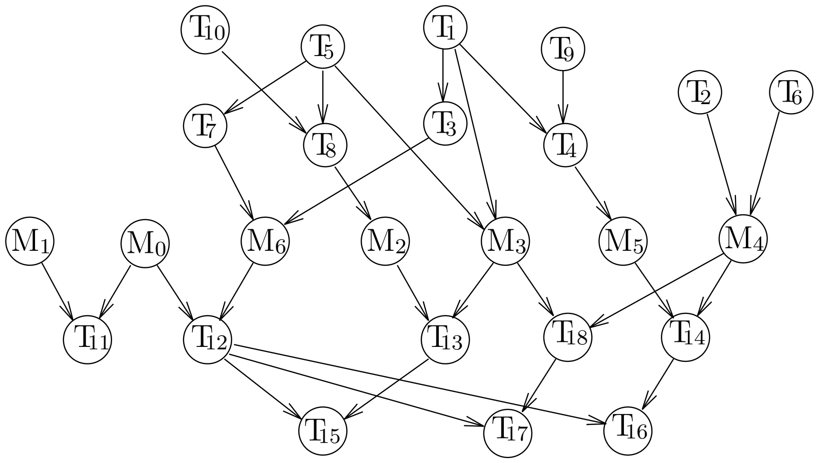


FIG. 2. *Improved precedence task graph for Winograd's algorithm*

Task $T_9 = A_{1,2} - A_{2,2}$
Task $T_{10} = B_{1,1} - B_{2,1}$
Task $T_{18} = M_3 + M_4$

Task $T_4 = T_1 + T_9$
Task $T_8 = T_5 + T_{10}$
Task $T_{14} = M_4 + M_5$
Task $T_{16} = T_{12} + T_{14}$
Task $T_{17} = T_{12} + T_{18}$

The new precedence task graph is shown in figure 2. The analysis from this subsection was first done in [18].

3.2. Precedence task graph for Strassen's method. There are several ways to obtain tasks that fulfill Strassen's algorithm; we have chosen the most promising one, i.e. the one with minimal depth: 4. The elementary tasks are the following and lead to the precedence graph of figure 3:

Task $T_1 = A_{1,1} + A_{2,2}$
Task $T_2 = B_{1,1} + B_{2,2}$
Task $T_3 = A_{1,2} - A_{2,2}$
Task $T_4 = B_{2,1} + B_{2,2}$
Task $T_5 = B_{2,1} - B_{1,1}$
Task $T_6 = A_{1,1} + A_{1,2}$
Task $T_7 = A_{2,1} + A_{2,2}$
Task $T_8 = B_{1,2} - B_{2,2}$
Task $T_9 = A_{2,1} - A_{1,1}$
Task $T_{10} = B_{1,1} + B_{1,2}$

Task $M_0 = T_1 \cdot T_2$
Task $M_1 = T_3 \cdot T_4$
Task $M_2 = A_{2,2} \cdot T_5$
Task $M_3 = T_6 \cdot B_{2,2}$
Task $M_4 = T_7 \cdot B_{1,1}$
Task $M_5 = A_{1,1} \cdot T_8$
Task $M_6 = T_9 \cdot T_{10}$

Task $T_{11} = M_0 + M_1$
Task $T_{12} = M_2 - M_3$
Task $T_{13} = M_5 - M_4$
Task $T_{14} = M_0 + M_6$
Task $T_{15} = T_{11} + T_{12}$
Task $T_{16} = M_2 + M_4$
Task $T_{17} = M_3 + M_5$
Task $T_{18} = T_{13} + T_{14}$

At a first view of the precedence graphs, it seems that Strassen's algorithm is more appropriate for parallelization; both graphs contain 7 multiplication tasks and 18 addition tasks, but Strassen's depth is smaller; if a large number of processors is available, parallel execution time for Strassen's algorithm will be shorter.

4. Task allocation. The general context is distributed-memory MIMD parallel computers. The model is multiport (several ports of a processor can work in parallel), links between processors are bidirectional and communication is presumed full duplex,

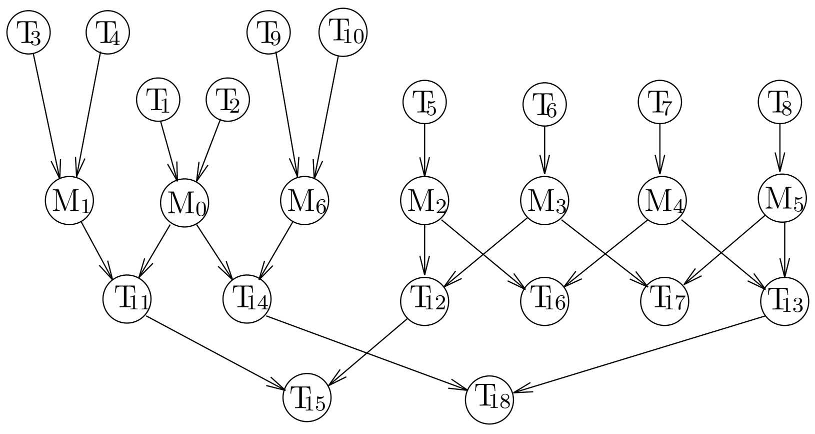


FIG. 3. *Precedence task graph for Strassen's algorithm*

though a half duplex model is also considered; p denotes the number of processors, and $P_0, P_1, \ldots, P_{p-1}$ the processors. The elementary step consists in computation followed by neighbour communication, but it is possible that steps exist that consist only of computation or only of communication.

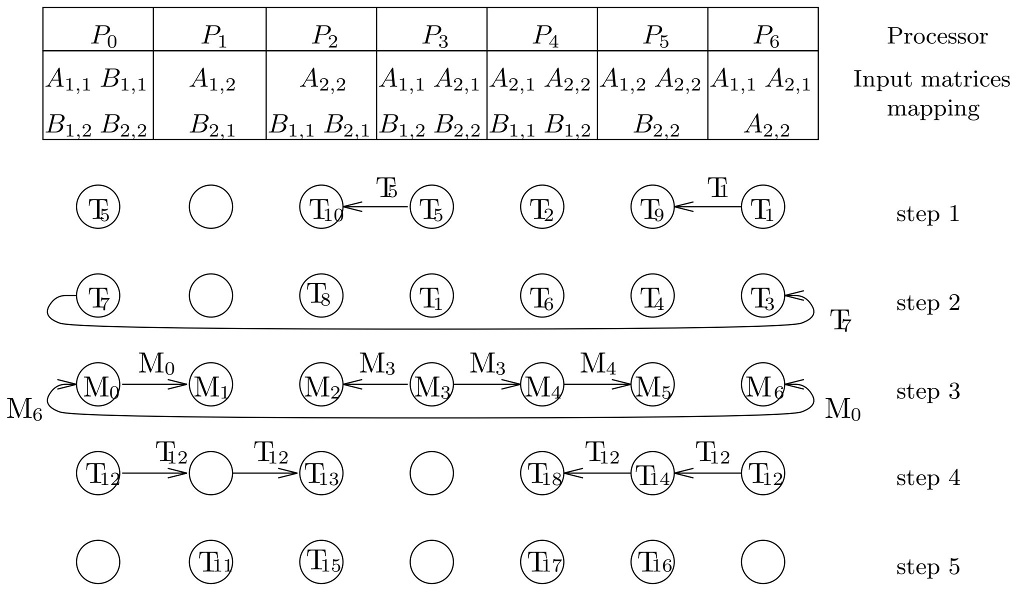
4.1. Task allocation on a ring. As we mentioned in the previous section, tasks consisting of multiplications have a much greater execution time than the others. Thus, let us consider a ring of 7 processors and try an allocation that permits the parallel execution of the 7 multiplication tasks; recall that we are concerned, for the moment, only with the first level of recursion of the fast $MM(n)$ algorithms. For both Winograd's and Strassen's algorithms such allocations are detailed in figures 4 and 5. Criteria to obtain these parallelizations were:

- parallel execution of multiplication tasks;
- minimal communication between processors;
- the memory needed by each processor, as well as an uniform repartition of matrix blocks, were ignored for the moment;

Some details about communication are necessary (references are to figure 4):

- in step 3, processors P_0 and P_6 exchange matrices computed in the same step (communication is presumed simultaneous, but, of course, it can be done sequentially);
- in step 3, processor P_3 sends the same matrix to its two neighbours (same remark as above);
- in step 4, processor P_1 receives a matrix from its left and sends the same matrix to its right (this communication is presumed pipelined);

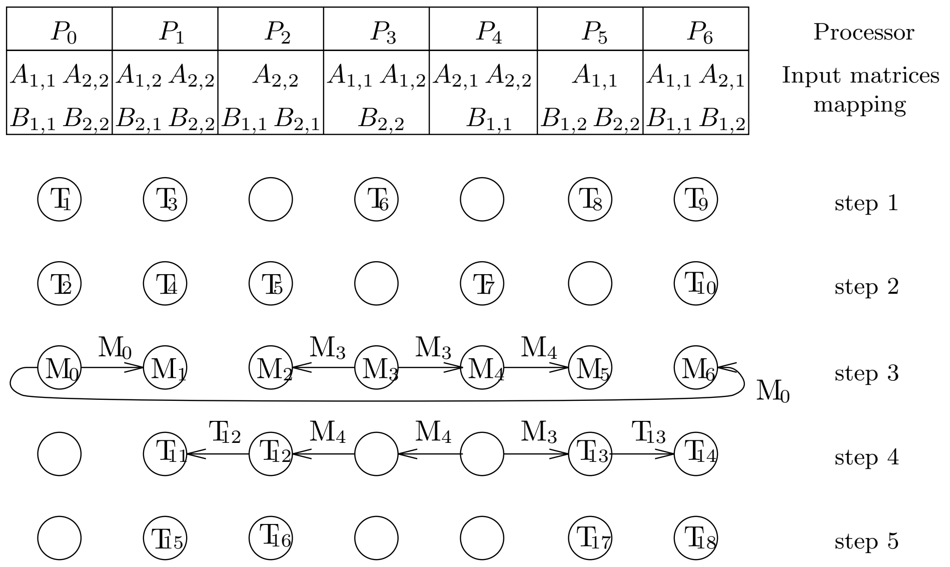
From the initial splitting into elementary tasks, described in the previous section, some redundant work was introduced for Winograd's algorithm; tasks T_1, T_5 and T_{12} are duplicated and placed on processors that otherwise were idle (*Duplicate and Accelerate* principle). For Strassen's algorithm, such means are not necessary. The chosen strategy guaranties the execution in the time constrained by the longest path of the precedence graph, and thus is optimal for Winograd; for Strassen it is also optimal, for the mentioned number of processors, which is smaller than the width of the precedence graph, and then, tasks that could be scheduled in parallel, have in fact sequential execution.



P_0	P_1	P_2	P_3	P_4	P_5	P_6	Processor
$A_{1,1}$ $B_{1,1}$ $B_{1,2}$ $B_{2,2}$	$A_{1,2}$ $B_{2,1}$	$A_{2,2}$ $B_{1,1}$ $B_{2,1}$	$A_{1,1}$ $A_{2,1}$ $B_{1,2}$ $B_{2,2}$	$A_{2,1}$ $A_{2,2}$ $B_{1,1}$ $B_{1,2}$	$A_{1,2}$ $A_{2,2}$ $B_{2,2}$	$A_{1,1}$ $A_{2,1}$ $A_{2,2}$	Input matrices mapping

step 1: T_5, –, T_{10}, T_5, T_2, T_9, T_1; arrows T_5, T_1
step 2: T_7, –, T_8, T_1, T_6, T_4, T_3; arrow T_7
step 3: M_0, M_1, M_2, M_3, M_4, M_5, M_6; arrows M_0, M_3, M_3, M_4, M_6, M_0
step 4: T_{12}, –, T_{13}, –, T_{18}, T_{14}, T_{12}; arrows T_{12}, T_{12}, T_{12}, T_{12}
step 5: –, T_{11}, T_{15}, –, T_{17}, T_{16}, –

FIG. 4. *Task allocation and execution scheme for parallel Winograd*



P_0	P_1	P_2	P_3	P_4	P_5	P_6	Processor
$A_{1,1}$ $A_{2,2}$ $B_{1,1}$ $B_{2,2}$	$A_{1,2}$ $A_{2,2}$ $B_{2,1}$ $B_{2,2}$	$A_{2,2}$ $B_{1,1}$ $B_{2,1}$	$A_{1,1}$ $A_{1,2}$ $B_{2,2}$	$A_{2,1}$ $A_{2,2}$ $B_{1,1}$	$A_{1,1}$ $B_{1,2}$ $B_{2,2}$	$A_{1,1}$ $A_{2,1}$ $B_{1,1}$ $B_{1,2}$	Input matrices mapping

step 1: T_1, T_3, –, T_6, –, T_8, T_9
step 2: T_2, T_4, T_5, –, T_7, –, T_{10}
step 3: M_0, M_1, M_2, M_3, M_4, M_5, M_6; arrows M_0, M_3, M_3, M_4, M_0
step 4: –, T_{11}, T_{12}, –, –, T_{13}, T_{14}; arrows T_{12}, M_4, M_4, M_3, T_{13}
step 5: –, T_{15}, T_{16}, –, –, T_{17}, T_{18}

FIG. 5. *Task allocation and execution scheme for parallel Strassen*

Parallel complexity study. Denote by m_q the number of multiplications and by a_q the number of additions necessary to solve sequentially the $MM(q)$ problem (by an algorithm not yet specified: standard, Winograd, Strassen or mixed), where $q = \frac{n}{2}$ is the dimension of the blocks. Also denote by $t_c(n)$ the number of matrix elements which are transmitted (for a simplier analysis, we do not count transmitted bytes; we do not consider the extra time needed by pipelining, by a full duplex transmission on a link or by a simultaneous transmission made by a processor on both links with its neighbours) and by $M(n)$ the maximal (over all processors) memory space occupied by the inputs (number of matrix elements).

For solving $MM(n)$ on a ring of processors, the complexity of parallel Winograd's algorithm is:

$$t_m(n) = m_q \qquad t_a(n) = n^2 + a_q \qquad t_c(n) = n^2 \qquad M(n) = n^2 \tag{7}$$

while for parallel Strassen's algorithm:

$$t_m(n) = m_q \qquad t_a(n) = n^2 + a_q \qquad t_c(n) = \frac{n^2}{2} \qquad M(n) = n^2 \tag{8}$$

These equations show that parallel Strassen is expected to be a little faster than parallel Winograd, due only to communication.

Bounding memory in parallel Strassen algorithm. Instead of searching for insignificant improvements of the communication time, we try to minimize the memory space. Figure 6 represents an allocation for Strassen's precedence task graph built under a supplementary constraint:

- initially, each processor has at most one block (of dimension $\frac{n}{2}$) of each input matrix A, B;

As it will be seen, the aim of this constraint is not only to keep memory space within acceptable limits, but will be a favorable factor in speeding up implementations on torus and hyper-torus.

We note that an initial step consisting only of communications is added. The number of steps including communication is 4. For this parallel version of Strassen's algorithm, called *parallel Strassen with minimal memory*, the complexity formulae are:

$$t_m(n) = m_q \qquad t_a(n) = n^2 + a_q \qquad t_c(n) = n^2 \qquad M(n) = \frac{n^2}{2} \tag{9}$$

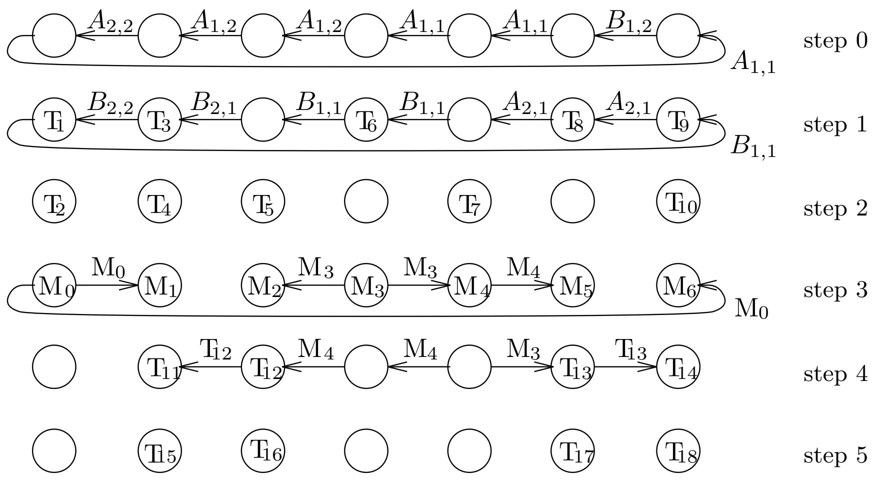
For Winograd's algorithm with memory constraint, we can also find an allocation, but it implies many initial communications and becomes unpractical.

4.2. Generalization to a hyper-torus. From the previous study on a ring of 7 processors, it is easy to recursively build a mapping on a hyper-torus. Recall that a hyper-torus denotes a k-dimensional torus with $k > 1$; we briefly denote a k-dimensional hyper-torus a k-torus. Recall also an informal recursive definition of the hyper-torus: a k-torus is a ring of $(k-1)$-tori, and a 1-torus is a ring.

Since the discussed fast algorithms are recursive, the main idea of the allocation on a hyper-torus is to embed recursion levels in successive rings of the hyper-torus. We first analyze Winograd's algorithm.

Consider a k-torus with $p = 7^k$ processors (each ring of the recursive definition is made up of 7 tori). The algorithm described in the previous subsection is applied for each 'ring', with the input matrices distributed over the processors that compose that ring. Each processor has only blocks of dimension $q = \frac{n}{2^k}$. The algorithm described in figure 4 is adapted as follows:

P_0	P_1	P_2	P_3	P_4	P_5	P_6	Processor
$A_{1,1}$ $B_{1,1}$	$A_{2,2}$ $B_{2,2}$	$A_{2,2}$ $B_{2,1}$	$A_{1,2}$ $B_{2,2}$	$A_{2,2}$ $B_{1,1}$	$A_{1,1}$ $B_{2,2}$	$A_{2,1}$ $B_{1,2}$	Input matrices mapping



$A_{2,2}$ $A_{1,2}$ $A_{1,2}$ $A_{1,1}$ $A_{1,1}$ $B_{1,2}$ $A_{1,1}$ step 0
T1 $B_{2,2}$ T3 $B_{2,1}$ $B_{1,1}$ T6 $B_{1,1}$ $A_{2,1}$ T8 $A_{2,1}$ T9 $B_{1,1}$ step 1
T2 T4 T5 T7 T10 step 2
M0 M_0 M1 M2 M_3 M3 M_3 M4 M_4 M5 M6 M_0 step 3
T11 T12 T12 M_4 M_4 M_3 T13 T13 T14 step 4
T15 T16 T17 T18 step 5

FIG. 6. *Task allocation and execution scheme for parallel Strassen with minimal memory*

- the input matrix distribution is the same, but considering that each processor of the initial ring is now a $(k-1)$-torus with 7^{k-1} processors, each $\frac{n}{2} \times \frac{n}{2}$ block is in fact distributed over this torus;

- steps 1 and 2 are made exactly in the same manner, each processor doing the operations with the blocks it owns;

- step 3 (multiplication) becomes recursive (recall that recursion levels are actually unrolled); the $MM(\frac{n}{2})$ problem that has to be solved by each $(k-1)$-torus is split into 7 $MM(\frac{n}{4})$ problems; each $(k-1)$-torus executes steps 1 and 2, etc.

- only at the last level of recursion, i.e. after k executions of steps 1 and 2, when each processor has a $MM(q)$ problem to solve locally, a sequential algorithm is applied;

- steps 4 and 5, the gathering of the results, are made in a reverse order, from inner to outer 'rings'; there are k executions of these steps;

- finally, the result is distributed among 4^k processors;

Parallel complexity study. Now, our goal is to derive complexity formulae for this parallel algorithm, similar with the ones for the ring architecture. As the algorithm is designed so that each processor has to solve a $MM(q)$ problem, it follows that $t_m(n) = m_q$ and $t_a(n) = a_q + O(n^2)$. To find the coefficient hidden by the big O, let us discuss the block distribution of the input matrices over the hyper-torus.

On a ring (block dimension $\frac{n}{2}$) the most loaded processor has 3 blocks of B and 1 of A (there are processors having 2 blocks of each matrix, but an asymmetric distribution will be worse for k-torus, with $k > 1$). On a 2-torus (block dimension $\frac{n}{4}$), with the embedded recursion algorithm, there will exist a processor with 3 blocks of each initial (from ring architecture) 3 blocks of B, i.e. 9 blocks of B, and 1 block of

A. Using inductive reasoning:

$$M(n) = \left[\left(\frac{3}{4}\right)^k + \left(\frac{1}{4}\right)^k\right] \cdot n^2 \tag{10}$$

That is, as k increases (and p), the required memory for a processor decreases very slowly, and not proportionally to the number of processors, like most algorithms on MIMD architectures. It turns out that memory requirements imply a growth in the number of additions. Only additions that appear in steps 1 and 2 are concerned.

For simplicity, exemplify on a 2-torus (with 7×7 processors), for the most loaded processor P_{loaded}. The inner ring to which P_{loaded} belongs must solve a $MM(\frac{n}{2})$ problem; P_{loaded} has 3 $\frac{n}{4} \times \frac{n}{4}$ blocks of a $\frac{n}{2} \times \frac{n}{2}$ matrix; but these blocks must be computed at the first level of recursion, by P_{loaded} as its part in the outer ring. So that, at this level, it will be 3 times more additions, for steps 1 and 2. The number of additions will be: $t_a(n) = [(2 \cdot 3 + 2) + 4] \cdot \left(\frac{n}{4}\right)^2 + a_{\frac{n}{4}}$. To be more explicit, there are (block additions):

- $2 \cdot 3$ - first level (outer ring), steps 1 and 2;
- 2 - second level (inner ring), steps 1 and 2;
- 4 - steps 4 and 5, second and first level; in fact, processor P_0, the most loaded on a ring, has nothing to do in step 5, as can be seen in figure 4; but P_0 stays idle while other processors perform step 5.

Note that for steps 4 and 5, the input matrix distribution is not important, as inputs for tasks in these steps are the results of the multiplication step; thus, only one block is handled for each processor.

A generalization is now simple (each term of the sum represents the maximal number of block additions performed in a recursion level, for steps 1, 2, 4, 5):

$$t_a(n) = a_q + [2 \cdot (3^{k-1} + 1) + \ldots + 2 \cdot (3^1 + 1) + 4] \cdot \frac{n^2}{4^k} = a_q + (3^k + 2k - 1)\frac{n^2}{4^k} \tag{11}$$

As each addition is either followed or preceeded by a communication, the same reasoning gives the following formula:

$$t_c(n) = (3^k + 2k - 1)\frac{n^2}{4^k} \tag{12}$$

Similar equations can be derived for parallel Strassen's algorithm (figure 5), but with a P_{loaded} that has 2 blocks of A and 2 of B; thus the coefficients of n^2 get new expressions:

$$\begin{gathered} t_m(n) = m_q \qquad t_c(n) = 2k \cdot \tfrac{n^2}{4^k} \qquad M(n) = \tfrac{1}{2^{k-1}} \cdot n^2 \\ t_a(n) = a_q + [2 \cdot (2^{k-1} + 1) + \ldots + 2 \cdot (2^1 + 1) + 4] \cdot \tfrac{n^2}{4^k} = a_q + (2^{k+1} + 2k - 2)\tfrac{n^2}{4^k} \end{gathered} \tag{13}$$

We now turn to the parallel Strassen algorithm with minimal memory; similar equations as above can be found, with the remark that counting additions is very simple: each level of the embedded recursion contributes with 4 block additions, in steps 1, 2, 4, 5 of the algorithm (the same situation is met for the number of transmitted matrix elements, although not exactly in the same steps).

$$t_m(n) = m_q \qquad t_a(n) = a_q + 4k \cdot \tfrac{n^2}{4^k} \qquad t_c(n) = 4k \cdot \tfrac{n^2}{4^k} \qquad M(n) = 2 \cdot \tfrac{n^2}{4^k} \tag{14}$$

Remarks. Some remarks on equations (10)-(14) are necessary. Assume that the sequential algorithm used for the $MM(q)$ problem is of complexity $t_m = c_m q^\omega$, $t_a = c_a q^\omega$, where $\omega = 3$ for the standard algorithm, and $\omega = \log 7$ for a fast algorithm. Then:

- for all three implementations of this subsection, considering only the most significant terms, the execution time is:

$$t(n) = (c_m + c_a) \cdot q^\omega + O(n^2) = O\left(\left(\frac{n}{2^k}\right)^\omega\right)$$

- the efficiency of parallelization - computed for any fast method - has a similar expression as the following, written for parallel Strassen with minimal memory (as an example):

$$E_p(n) = \frac{(c_m + c_a)n^\omega + O(n^2)}{7^k[(c_m + c_a)q^\omega + 4k(1+\tau)\frac{n^2}{4^k}]}$$

where τ is the number of arithmetic operations that can be executed while a matrix element is transmitted between two processors; it is obvious that $E_p(n) \to 1$ as $n \to \infty$, and this assertion is true for all discussed parallel algorithms.

- if the coefficient of n^ω decreases proportionally to the number of processors p, the coefficient of n^2 is decreasing much more slowly; a short analysis shows that parallel Strassen is the most efficient for small k, while parallel Strassen with minimal memory is the best for bigger k; the value of k for which the two methods have the same complexity depends on τ.

There is one more advantage of the parallel Strassen with minimal memory algorithm. For all three parallel algorithms described in figures 4 - 6, for a ring topology, each one can be described as follows (we assimilate communication before sequential matrix multiplication to steps 1 and 2 and the one after multiplication to steps 4 and 5):

```
do steps 1 and 2 communicating with immediate neighbours on ring
do step 3 (without communication)
do steps 4 and 5 communicating with immediate neighbours on ring
```

In the embedded recursion, at level j of recursion, a processor communicates with immediate neighbours on ring j of the recursive description of a hyper-torus. Only the parallel Strassen with minimal memory supports a description like the following one:

```
for $j = 1$ to $k$
    do steps 1 and 2 communicating with immediate neighbours on ring $j$
do step 3 (without communication)
for $j = k$ downto 1
    do steps 4 and 5 communicating with immediate neighbours on ring $j$
```

where steps description is exactly the same as for the ring topology, with parametrized communication. For Winograd or Strassen without memory constraint, steps 1 and 2 must be changed, because processors have to perform a variable number of block additions or communications. This implies that each processor has its own program, different from other ones, and makes programming fastidious.

Conclusions. An analysis was made in this subsection of three parallel fast matrix multiplication algorithms implemented on a hyper-torus topology. For all of them an asymptotic efficiency equal to 1 was found. The best algorithm that we considered is parallel Strassen with minimal memory, for the following reasons:

- a good arithmetic and communication complexity, as can be seen in equations (14);

- a much easier implementation; it was the only retained for experimentations on torus;

- it can be easy coupled with a standard algorithm on sub-tori in order to use a number of processors which is not a power of 7, but only a multiple.

5. Other algorithms for parallel matrix multiplication. In this section we recall short descriptions of standard algorithms on MIMD architectures, and we present a new mixed parallel algorithm, which results from the combination of a fast method with a standard one.

On a ring topology. We describe two methods for matrix multiplication.

1. The first is the most regular; input matrices are partitioned in square blocks of size $\frac{n}{p} \times \frac{n}{p}$. Each processor P_j has in its local memory a block column of input matrices, that is blocks $A_{i,j},\ B_{i,j},\ i = 0 \dots (p-1)$; finally, each processor will have the correspondent block column of the result C; result blocks are initialized with zeroes. For processor P_j, the algorithm is:

```
for $k = 0$ to $p-1$
    for $i = 0$ to $p-1$
        multiply $A_{i,local}$ with $B_{(j+k) \bmod p,j}$ and add the result to $C_{i,j}$
    in parallel do $\{$*block column shift of A*$\}$
        send local block column of A to the right
        receive a block column of A from left and store it as local
```

The complexity formulae are (for more information see [11]):

$$t_m(n) = p^2 m_{\frac{n}{p}} \qquad t_a(n) = p^2 a_{\frac{n}{p}} + n^2 \qquad t_c(n) = n^2 \qquad M(n) = 2\frac{n^2}{p} \tag{15}$$

2. The second is obtained by imposing conditions of minimal communication, implying that $p = 8$; each processor has an $\frac{n}{2} \times \frac{n}{2}$ block of inputs A and B; it computes the multiplication of the blocks; only half of processors make a block addition. For P_i the algorithm is:

```
multiply local blocks of $A$ and $B$
if $i$ is even
   send the result to right
else
   receive a block result from left
   add local result block with the received one
```

and leads to:

$$t_m(n) = m_{\frac{n}{2}} \qquad t_a(n) = a_{\frac{n}{2}} + \frac{n^2}{4} \qquad t_c(n) = \frac{n^2}{4} \qquad M(n) = \frac{n^2}{2} \tag{16}$$

On a torus topology. Three algorithms are presented below:

1. The standard algorithm is the following (processors are denoted $P_{i,j}$, upon their position in the torus; each one is presumed to have in its local memory input matrices blocks $A_{i,j}$, $B_{i,j}$, and to compute result block $C_{i,j}$, which is initialized with zeroes; there are $p \times p$ processors):

```
for $k = 0$ to $p - 1$
    if $(i - j + k) \bmod p = 0$ then {*proc. starting diffusion on its row*}
        send local block of A to right
    else receive a block of A from left and store it
        if $(i - j + k) \bmod p \neq p - 1$ {*other proc. than that ending diff.*}
            send the block to right
    multiply the diffused block of A with the local block of B
        and add to the local block of result
    in parallel do {*vertical block shift of B* }
        send local block of B above
        receive a block of B from below and store it as local
```

The complexity formulae are:

$$t_m(n) = pm_{\frac{n}{p}} \qquad t_a(n) = pa_{\frac{n}{p}} + \frac{n^2}{p} \qquad t_c(n) = 2\frac{n^2}{p} \qquad M(n) = 2\frac{n^2}{p^2} \tag{17}$$

For a detailed description see [9]. Another form of the parallel standard algorithm, reducing communication by means of preskewing, is due to Cannon (for more information see [3]); we implemented it but did not notice significant differences over the algorithm above; so that we do not report any result.

2. A generalization of the algorithm with minimal communication can be made on a torus. The definition is an embedded recursion, similar with that used for fast algorithm, but with $p = 8^k$. General formulae for complexity can be found in [8]; those for the torus are:

$$t_m(n) = m_{\frac{n}{4}} \qquad t_a(n) = a_{\frac{n}{4}} + \frac{n^2}{8} \qquad t_c(n) = \frac{n^2}{8} \qquad M(n) = \frac{n^2}{8} \tag{18}$$

These relations show that communication is really minimal: there are $b = 2^k$ blocks, $p = b^3$ processors (implying maximal parallelism) and only $k = \log b$ blocks are communicated by the most loaded processor.

3. As usual parallel computers have a number of processors that is a power of 2, it is natural to think to a variant of a fast algorithm, adaptable to various dimensions of the torus; in fact one dimension is 7; denote the other with d. The idea is (again) to embed successive levels of the algorithm in successive rings of a torus, but now with a first level using parallel Strassen with minimal memory and the second the parallel standard on a ring of d processors; we think that this brief and informal description is sufficient and present complexity formulae:

$$t_m(n) = d^2 m_{\frac{n}{2d}} \qquad t_a(n) = d^2 a_{\frac{n}{2d}} + (d+4)\frac{n^2}{4d} \qquad t_c(n) = (d+4)\frac{n^2}{4d} \qquad M(n) = \frac{n^2}{4d} \tag{19}$$

6. Experimental results. In order to present experimental results we recall here all algorithms presented in this paper.

1. On ring topology:
    - StaR - standard algorithm;

   - StaComR - standard algorithm with minimal communication;
   - WinoR - parallel Winograd;
   - StrComR - parallel Strassen (with minimal communication);
   - StrMemR - parallel Strassen with minimal memory;
2. On torus topology:
   - StaT - standard (with diffusion on rows);
   - StaComT - standard with minimal communication;
   - StrMemT - parallel Strassen with minimal communication;
   - MixT - mixed Strassen and standard;

For each algorithm, on each processor, a sequential method is finally applied; a letter, S or W, is added to the names above, as the sequential method is the standard or the mixed Winograd. For algorithms that can be implemented on various number of processors, that number is added to the name. So that, StaTS64 is the parallel standard algorithm on a torus 8×8, with sequential standard method on a processor. If context is relevant, this final information will be omitted.

Experiments were carried on a Telmat MegaNode computer; this is a parallel computer of the family of supernodes, based on the T800 transputer, with configurable network topology. Processors (i.e. transputers) are grouped in nodes; each node has 16 transputers connected via a programmable electronic switch; two nodes are coupled in a tandem; tandems can be coupled via a superior level of switches. This architecture permits any network configuration of degree at most 4 (each transputer has 4 bidirectional connection links). Each transputer of the MegaNode has a local memory of 4Mb. For more information see [1].

For the experiments, the used network configurations were rings of 7 or 8 processors and tori of 49 (7×7), 64 (8×8) and 63 (9×7) processors.

Denote by $T_X(n)$ the execution time for a certain algorithm X that solves the $MM(n)$ problem; it can be written explicitly:

$$T_X(n) = (t_m(n) + t_a(n) + \tau t_c(n))\alpha$$

where α is the time necessary for a floating point operation and τ is the number of floating operations executed during the transmission between two neighbour processors of a floating point number.

Define, as usually, the efficiency on p processors of a parallel algorithm X as (S is the sequential algorithm):

$$\epsilon_X(n) = \frac{T_S(n)}{pT_X(n)}$$

In order to compare two algorithms for the same topology and for the same number of processors, we define the speed ratio of algorithm X to algorithm Y as:

$$Sr(n) = \frac{T_Y(n)}{T_X(n)}$$

If X is faster than Y, it is obvious that $Sr(n) > 1$. If the number of processors is different, but on the same topology, speed ratio is redefined as work ratio:

$$Wr(n) = \frac{p_Y T_Y(n)}{p_X T_X(n)}$$

where p_Y, p_X are the number of processor on which algorithms Y and X are implemented.

n	StaR8	StaR7	StaComR	WinoR	StrComR	StrMemR
64	0.29	0.31	0.25	0.34	0.30	0.35
128	2.01	2.14	1.81	2.13	2.01	2.10
256	14.93	16.61	12.79	14.04	13.60	13.91
512	109.76	132.49	90.08	95.09	93.33	94.53
768	354.25	397.19	284.78	296.03	292.08	294.77
1024	795.08	954.27	632.89	652.94	645.94	650.72

TABLE 2
Timings for ring topology, fast local sequential method

n	StaR8	StaR7	StaComR	WinoR	StrComR	StrMemR
64	0.29	0.31	0.25	0.34	0.30	0.35
128	2.01	2.14	1.86	2.18	2.06	2.16
256	14.93	16.15	14.23	15.49	15.04	15.36
512	115.07	130.11	111.33	116.34	114.58	115.77
768	383.53	428.51	372.89	384.14	380.20	382.89
1024	903.38	1024.00	880.48	900.55	893.53	898.31

TABLE 3
Timings for ring topology, standard local sequential method

We present in the sequel experimental results for the described methods and comment them. We will insist on the results on torus topology, those for ring being similar.

6.1. Experiments on a ring. Results are presented in table 2 for fast sequential method (mixed Winograd) and in table 3 for standard sequential method; execution times are given in seconds. From both tables it can be seen that:

- fast algorithms timings differ only in a few occasions; in fact their theoretical complexities are the same for the leading term and differ only at the coefficient of the n^2 term; we choose StrMemR for comparison with other algorithms.

- standard algorithm with minimal communication StaComR is only a little faster than StaR for standard sequential method, but much faster for fast sequential method; the explanation is that StaComR works, at processor level, with blocks of dimension $\frac{n}{2}$, while StaR with blocks of $\frac{n}{8}$; there are two more levels of sequential recursion for StaComR. The asymptotic speed ratio is (we refer to relations derived in precedent sections; for sake of brevity, we denote by only m_q the term hiding the greatest power of n, that is $m_q + a_q$):

$$Sr(n) = \frac{8^2 m_{\frac{n}{8}}}{m_{\frac{n}{2}}} = \frac{64}{49} \approx 1.31$$

- we compare now StrMemR with StaR7, i.e. a fast algorithm vs. the most usual for ring architecture. The asymptotic speed ratio is :

$$Sr(n) = \frac{7^2 m_{\frac{n}{7}} + O(n^2)}{m_{\frac{n}{2}} + O(n^2)}$$

For standard sequential method $Sr(n) = \frac{8}{7} \approx 1.14$, while for fast sequential method $Sr(n) = 7^{3-\log 7} \approx 1.45$. Figure 8 presents the speed ratio for StrMemRW vs. StaRS7,

n	StaT64	StaT49	StaComT	MixT63	StrMemT
128	0.44	0.49	0.27	0.35	0.45
256	2.59	2.85	1.85	1.87	2.50
512	15.10	20.30	12.95	13.82	15.46
1024	103.91	141.17	90.74	99.63	100.65
1536	327.31	422.04	286.26	341.90	308.52
2048	726.31	988.92	635.48	745.24	675.02

Table 4
Timings for torus topology, fast local sequential method

n	StaT64	StaT49	StaComT	MixT63	StrMemT
128	0.44	0.49	0.27	0.35	0.45
256	2.59	2.85	1.93	1.87	2.58
512	15.77	19.96	14.65	13.82	17.15
1024	117.45	151.13	113.97	101.28	123.88
1536	388.57	501.89	381.08	342.99	403.33
2048	912.16	1180.10	899.05	794.08	938.58

Table 5
Timings for torus topology, standard local sequential method

i.e. a comparison between the fastest algorithm and the 'classic' one; for $n = 1024$ it can be observed that fast method is about 57% faster, and only 2% for $n = 128$; as, in this case, speed ratio infinitely grows with matrix dimension (following a curve shape of the form $n^{3-\log 7}$), for bigger n better improvements are expected.

- if StrMemR is compared with the fastest standard algorithm, StaComR, an work ratio of $\frac{8}{7} \approx 1.14$ is expected for any sequential method (block dimensions are equal to $\frac{n}{2}$ for both algorithms); that means that time for StrMemR on 7 processors has to be the same that time for StaComR on 8 processors; of course, due to greater communication complexity, StrMemR will always have greater execution time.

For all comparisons above, data resulted from experiments show an asypmptotic behaviour that confirms the theoretical values. It can be appreciated that on a ring topology, fast algorithms significantly improve speed, beginning from matrix dimensions of 200.

From the point of view of efficiency, parallel Strassen with minimal memory permits a good parallel implementation, as figure 11a show. As sequential timings were too expensive for bigger n, we present results only for $n \leq 512$ (as n grows efficiency will grow too). Naturally, we present efficiency only for StrMemRW; if the standard sequential method is used, globally, the algorithm may be considered mixed (a parallel phase based on the fast method, followed by a standard sequential one); thus a comparison with a unitary sequential algorithm will be irrelevant in terms of efficiency. Sequential timings are taken from table 1, for Winograd method, $n_0 = 32$.

6.2. Experiments on a torus. Results are presented in table 4 for fast sequential method (mixed Winograd) and in table 5 for standard sequential method; execution times are given in seconds.

Analogous remarks as those for ring architecture will be now presented, for pairs of algorithms; asymptotic limits are derived; some of them can be considered somewhat unexpected.

- for standard algorithms, StaComT vs. StaT64, the same considerations as

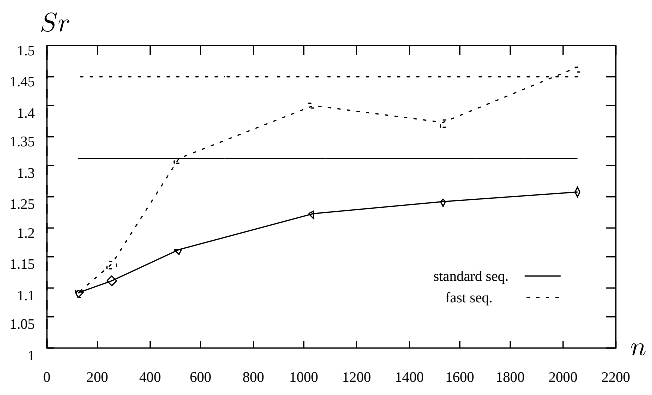


FIG. 7. *StrMemT vs. StaT49*

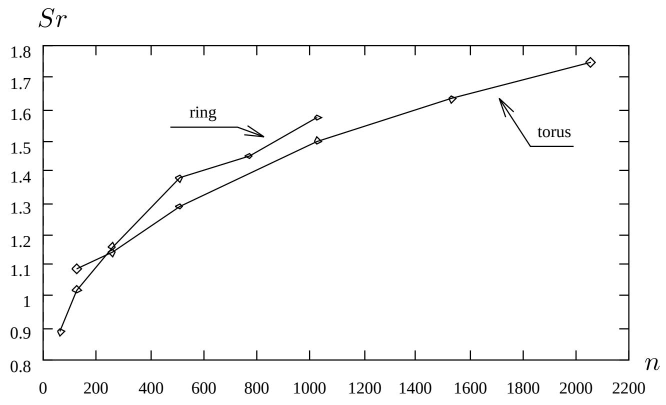


FIG. 8. *StrMemW vs. StaTS*

above; the speed ratio has now an asymptotic limit of 1.14 for fast sequential method; there is only one extra sequential level of recursion: blocks of $\frac{n}{4}$ for StaComT, of $\frac{n}{8}$ for StaT64.

- comparison between fast algorithm StrMemT and standard StaT49 leads to the speed ratio:

$$Sr(n) = \frac{7m_{\frac{n}{7}} + O(n^2)}{m_{\frac{n}{4}} + O(n^2)}$$

For standard sequential method $Sr(n) = \frac{64}{49} \approx 1.31$; for fast sequential method $Sr(n) \approx 1.45$. Actual values in figure 7. Asymptotic values are represented with horizontal lines, with the same type as the lines for the experimental values.

The fastest algorithm (StrMemTW) can be compared with the usual one (StaTS49); from figure 8 it can be seen that speed improvement begins to be significant for matrix dimension of about 200 (as for ring topology) and goes to 75% for $n = 2048$.

Figure 9 presents timings for StrMemTW, StaTW49 and StaTS49, that is algorithms for 49 processors. Two kind of improvements can be noted, over the usual algorithm StaTS49: the use of a fast sequential method, in StaTW49, and of both fast parallel and sequential methods in StrMemTW.

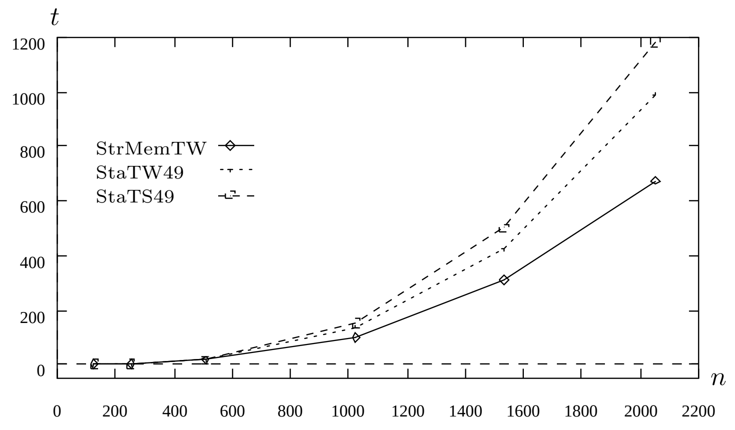


FIG. 9. *Timings (in seconds) for algorithms for 49 processors.*

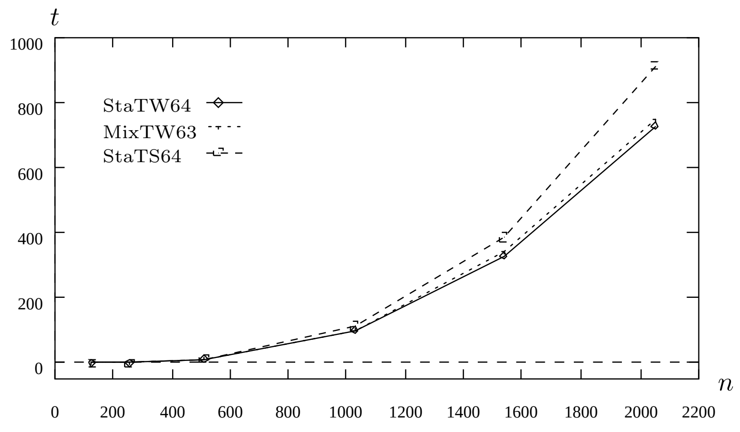


FIG. 10. *Timings (in seconds) for algorithms for 64 processors.*

- the most interesting is the comparison between the mixed algorithm MixT63 (on a torus of 7×9 processors; torus dimensions are chosen so that the number of processors for the mixed algorithm has a value as close as possible to a power of 2) and the standard one StaT64; speed ratio is:

$$Sr(n) = \frac{8m_{\frac{n}{8}} + O(n^2)}{9^2 m_{\frac{n}{18}} + O(n^2)}$$

leading to the following particular values: for standard sequential method $Sr(n) = \frac{9}{8} = 1.125$, while for a fast sequential method a more complicate formula is obtained $Sr(n) = \frac{9^2 \cdot 7^2}{8 \cdot 9^{\log 7}} \approx 0.96$; thus, the best method depends on the sequential algorithm. In figure 10, timings for algorithms on a 64 processors architecture are presented. While the use of a fast sequential method significantly improves the behaviour of the standard algorithm StaT, the mixed algorithm (the one more appropriate to this number of processors) does not bring new improvements.

- if StrMemT is compared with the fastest standard algorithm, StaComT, a work ratio of $\frac{64}{49} \approx 1.31$ is expected for any sequential method (block dimensions are equal to $\frac{n}{4}$ for both algorithms).

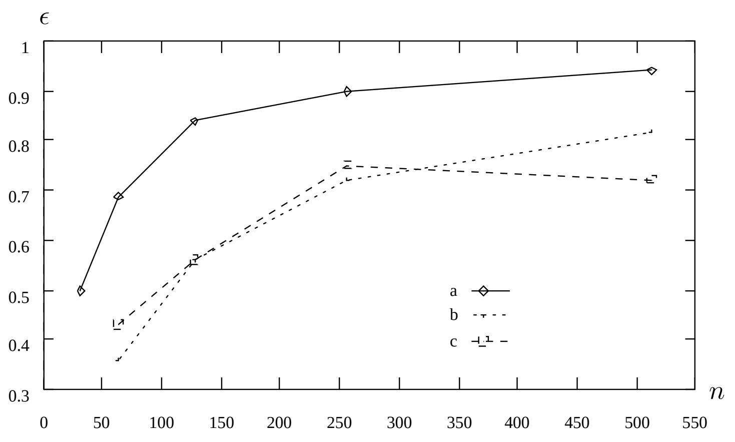


FIG. 11. *Efficiency for: (a) StrMemR (b) StrMemT (c) MixT63*

Efficiency for StrMemT is presented in figure 11b, while for MixTW63 in figure 11c.

6.3. About overheads. We also performed a series of experiments simulating communication, in order to approximate overhead due to communication; denote by T_{nc} execution time without communication. We use a simplified model for the expression of T_p - the execution time for a parallel algorithm:

$$T_p = T_a + T_c + T_l$$

where T_a is the time for arithmetic operations (we can consider that $T_a \approx \frac{T_s}{p}$ where T_s is the serial time), $T_c = T_p - T_{nc}$ is the time spent for communication and $T_l = T_{nc} - T_s$ the time lost in parallel execution because of poor load balancing, specific software implementation, etc. Thus, two kind of overheads can be computed (see [9] for details):

1. overhead due to communication: $O_c = 1 - \frac{T_{nc}}{T_p}$
   In fact, since communication protocol for transputers implies synchronization, an unmeasurable part (that we believe to be small) of this overhead is due to idle time.
2. load balancing, software, algorithmic overhead: $O_l = \frac{T_{nc} - \frac{T_s}{p}}{T_p}$

All variables above are dependent on n, for the $MM(n)$ problem; for notation simplicity, we have omitted n in these formulae.

T_{nc} was measured for all described parallel algorithms; four other tables, similar with tables 2-5, can be presented, but, in order to alleviate presentation, we limit ourselves to some graphics. O_c and O_l were computed for StrMemRW and StrMemTW (thus, only for the fast sequential method), and can be seen in figure 12 (communication overhead) and in figure 13 (load balancing and software overhead).

Some comments are necessary. For small matrix size, communication overhead has much greater values because the ratio computation vs. communication is proportional with n; thus O_c has to have the same aspect as $\frac{1}{n}$ (and it have, as figure 12 shows); however, there is another factor: a pipeline is made with messages with fixed size; so that, for small n pipelining is quasi-inexistent.

The great difference between ring and torus topology is due, principally, to a hardware factor. In a supernode computer, communication time depends on the relative position of the two implied processors; in a tandem communication is about twice

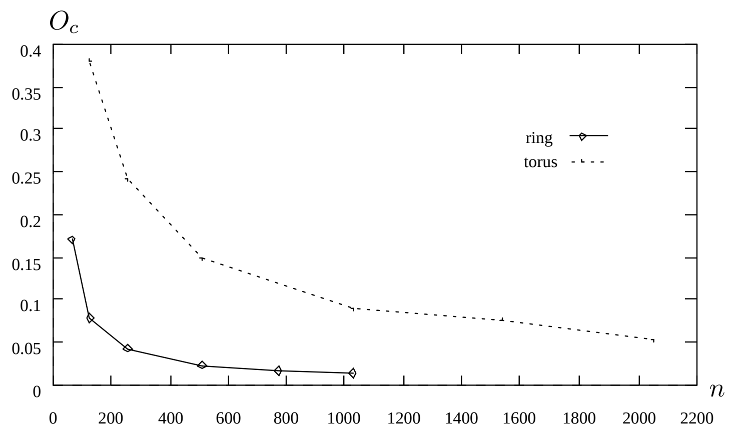


FIG. 12. *Communication overhead for StrMem*

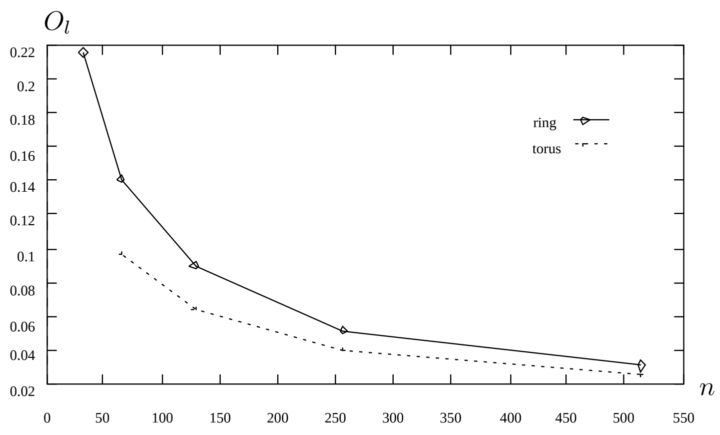


FIG. 13. *Load balancing and software overhead for StrMem*

faster than inter tandem, because of the extra level of switches that links tandems; for details see [1].

As for O_l torus topology seems to be better, we can suppose that the communication overhead contains much of the load balance overhead for torus.

If the standard sequential method is used, a small decrease in communication overhead is expected, as parallel time increases.

7. Conclusions. The presented parallel implementations of fast matrix multiplication algorithms on MIMD architectures are proven to be faster than standard parallel algorithms, on ring or on torus topologies. Speed improvement becomes for matrix dimensions of 200 (about 10% faster) and, for sufficiently big n, measured timings are close to the theoretical predicted values, that is 30% when the local sequential method is the same; when sequential methods are different (fast sequential for fast parallel, standard sequential for standard parallel) maximal measured speed growth was 75% for $n = 2048$; this latter result is consistent with the one reported by Bailey [2], for the same matrix dimension (in another context, a Cray-2 supercomputer), i.e. 101%, but with 35% of this improvement due to other causes.

However, a disadvantage of these parallel fast algorithms is the fixed number of processors, while standard algorithms can be easily customized. The parallel mixed algorithm (MixT), more flexible from this point of view, is good only for standard sequential method. All proposed algorithms can be efficient on dedicated hardware.

On a configurable topology, such as the supernode, these algorithms require a smaller number of processors for the same speed (compared with standard ones), an advantage in a multiuser environment.

As a general conclusion, fast matrix multiplication algorithms cannot be ignored, on MIMD computers as well as on SIMD computers. They can bring, by themselves, a considerable speed-up of applications, that is more important than the implied implementation difficulties.

REFERENCES

[1] Y. Arrouye et al., *Manuel du Meganode, version 2.1*, Tech. Rep. RT79, LMC-IMAG, 1992.

[2] D. Bailey, *Extra high-speed matrix multiplication on the Cray-2*, SIAM J. Sci. Sta. Comput., 9 (1988), pp. 603–607.

[3] P. Bjørstad, F. Manne, T. Sørevik, and M. Vajteršic, *Efficient Matrix Multiplication on SIMD Computers*, SIAM J. Matrix Anal. Appl., 13(1) (1992), pp. 386–401.

[4] R. Brent, *Algorithms for matrix multiplication*, Tech. Rep. CS157, Stanford University, 1970.

[5] E. Coffman and P. Denning, *Operating Systems Theory*, Prentice-Hall, 1973.

[6] J. Cohen and M. Roth, *On the implementation of Strassen's fast multiplication algorithm*, Acta Informatica, (1976), pp. 341–355.

[7] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, in 19 th. Annual ACM Symp. Theory Comp., 1987, pp. 1–6.

[8] B. Dumitrescu, J. Roch, and D. Trystram, *Fast Matrix Multiplication Algorithms on MIMD Architectures*, Tech. Rep. RT80, LMC-IMAG, 1992.

[9] G. Fox et al., *Solving problems on concurrent processors*, Prentice-Hall, 1988.

[10] M. Garey and D. Johnson, *Computers and Intractability - A Guide to the Theory of NP-Completeness*, W.H.Freeman and Company, 1979.

[11] G. Golub and C. Van Loan, *Matrix Computations. Second edition*, The John Hopkins University Press, 1989.

[12] N. Higham, *Exploiting Fast Matrix Multiplication Within the Level 3 BLAS*, ACM Trans. Math. Soft., 16 (1990), pp. 352–368.

[13] IBM, *Engineering and Scientific Soubroutine Library, Guide and Reference, Release 3*, 4th ed., 1988.

[14] R. Karp and V. Ramachandran, *Parallel Algorithms for Shared-Memory Machines*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., Elsevier Science Publishers, 1990.

[15] A. Kreczmar, *On memory requirements of Strassen algorithm*, in Algorithms and Complexity: New Directions and Recent Results, J. Traub, ed., Academic-Press, 1976.

[16] D. Lee and M. Aboelaze, *Linear Speedup of Winograd's Matrix multiplication algorithm using an array of processors*, in 6 th. IEEE Distributed Memory Computing Conference, 1991, pp. 427–430.

[17] V. Pan, *How to Multiply Matrix Faster*, Springer-Verlag, L.N.C.S. vol. 179, 1984.

[18] J. Roch and D. Trystram, *Parallel Winograd Matrix Multiplication*, in Parallel Computing: From Theory to Sound Practice, W. Joosen and E. Milgrom, eds., IOS Press, 1992, pp. 578–581.

[19] V. Strassen, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.

[20] S. Winograd, *Some remarks on fast multiplication of polynomials*, in Complexity of Sequential and Parallel Numerical Algorithms, J. Traub, ed., Academic-Press, 1973, pp. 181–196.