Randomized PRAM Simulation Using T9000 Transputers

Zbigniew J. Czech * Wojciech Mikanik †
Institute of Computer Science, Silesia University of Technology
44–100 Gliwice, Poland
Fax: +48-32-37-27-33
E-mails: {zjc,wmikanik}@zeus.polsl.gliwice.pl

Abstract

The parallel random access machine (PRAM) is the most commonly used general-purpose machine model for describing parallel computations. Unfortunately the PRAM model is not physically realizable, since on large machines a parallel shared memory access can only be accomplished at the cost of a significant time delay. A number of PRAM simulation algorithms have been presented in the literature. The algorithms allow execution of PRAM programs on more realistic parallel machines. In this paper we study the randomized simulation of an EREW (exclusive read, exclusive write) PRAM on a module parallel computer (MPC). The simulation is based on utilizing universal hashing. The results of our experiments performed on the MPC built upon Inmos T9000 transputers throw some light on the question whether using the PRAM model in parallel computations is practically viable given the present state of technology.

Key words. Parallel computing, PRAM model, randomized PRAM simulation algorithms, module parallel computer, universal hashing, T9000 transputer

1 Introduction

The parallel random access machine (PRAM) is the most commonly used general-purpose machine model for describing parallel computations. The PRAM consists of a set of processors, where each processor is a random access machine (RAM). All processors share the memory and communicate through it. The PRAM is relatively easy to program, because one does not need to allocate storage within a distributed memory or specify interprocessor communication. Unfortunately the PRAM model is not physically realizable, since on large machines a parallel shared memory access can only be accomplished at the cost of a significant time delay.

A number of PRAM simulation algorithms have been presented in the literature (for the survey see [4]). The algorithms allow execution of PRAM programs on more realistic parallel machines. Among several types of such machines is a fully connected parallel computer called a module parallel computer (MPC). The MPC consists of a set of RAM processors. Each processor of the MPC has an associated memory module and is connected via communication links to all other processors. A memory module operates sequentially responding to only one data access request at a time.

In this paper we study the randomized simulation of an EREW (exclusive read, exclusive write) PRAM on an MPC. The simulation is based on utilizing universal hashing. The results of our experiments performed on the MPC built upon Inmos T9000 transputers throw some light on the question whether using the PRAM model in parallel computations is practically viable given the present state of technology.

The remainder of the paper is organized as follows. In section 2 we describe the PRAM model. Section 3 defines the MPC. Section 4 presents some theoretical results regarding the randomized

*Research supported by the Polish Committee for Scientific Research under the grant BK-228/RAu2/95.
†Research supported by the European Committee under the Tempus programme grant IMG-94-PL-1098 and by the Polish Committee for Scientific Research under the grant BK-228/RAu2/95.

PRAM simulation. In Section 5 we describe the architecture of the Parsys SN9500 parallel computer which served as the platform for our experiments. In Section 6 the PRAM simulators which have been designed and implemented are discussed. Section 7 presents a matrix multiplication algorithm used for the purpose of simulation. In Section 8 the experiments which were conducted are described. Section 9 concludes the paper. The Appendix contains the results of the experiments.

2 PRAM model

An (n, m)-PRAM consists of n RAM processors, $P_0, P_1, \ldots, P_{n-1}$, and a shared memory of m locations, also called *variables* (see Fig. 1). The processors work synchronously, i.e. no processor will

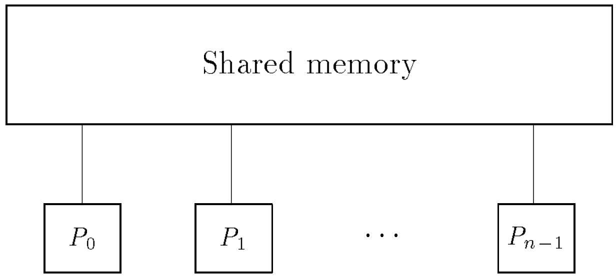

Figure 1: The PRAM model of computation

proceed with instruction $i + 1$ until all have finished instruction i. In every step of the PRAM, each processor executes a private RAM instruction. In particular, each processor may read a variable from the shared memory into its local memory, write a variable from its local memory to the shared memory, or perform some internal computation (e.g., addition, multiplication, boolean operation etc.) on the variables contained in its local memory. It is assumed that the execution of each instruction takes unit time. Depending on whether various processors may access the same memory location on a given step or not, the following variants of the PRAM model are distinguished:

- the exclusive read, exclusive write (EREW) PRAM, in which at most one processor may read or write to a particular variable,
- the concurrent read, exclusive write (CREW) PRAM, in which multiple processors may read from a particular variable, but at most one processor may write to a particular variable,
- the concurrent read, concurrent write (CRCW) PRAM, in which multiple processors may read or write to any variable.

There is also a further classification of the CRCW PRAM model based on a writing conflict resolution strategy which specifies what is written when more than one processor writes to a particular variable on a given step. For more details regarding this classification see [8, 3, 1].

3 Module parallel computer

A module parallel computer (MPC) consists of n RAM processors, each of which has an associated memory module [7]. A memory module is a collection of variables. Every processor may access every memory module via a fully connected network linking the processors (see Fig. 2). It is assumed that an access takes constant time. The memory modules, however, are sequential devices, i.e. all access requests that arrive at a memory module in a given step are processed one at a time. This can result in memory contention, in which an access request is delayed because of a concurrent request to the same module.

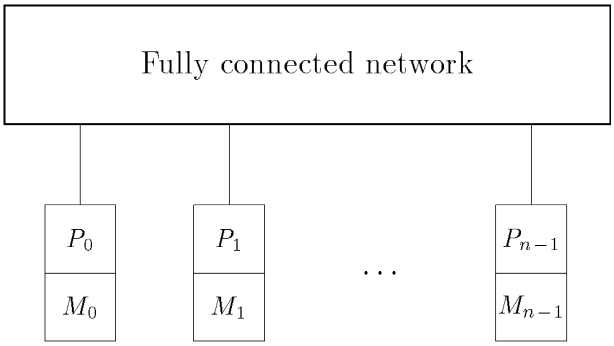

Figure 2: The module parallel computer

4 Randomized PRAM simulation

By a simulation of machine M_1 on machine M_2 we understand an algorithm that allows an instruction from M_1 to be executed on M_2. Our goal is to simulate an EREW PRAM on a more realistic parallel machine, namely, an MPC. The basic problem which must be solved by the simulation algorithm concerns the memory management, and it can be formulated as follows. Consider an (n, m)-PRAM which is to be simulated on an MPC with n memory modules, so that each memory module will hold m/n memory locations. Suppose that on a given MPC step each processor issues a memory request. Then, in the best case each request will go to a different module, and all requests may be serviced in $O(1)$ time (recall that the communication time between the MPC processors is constant). In the worst case however, all n requests may be directed to the same memory module, and will be serviced in $\Omega(n)$ time. The problem of memory management is how to map the logical addresses of the PRAM into the physical addresses of the MPC distributed over its n memory modules such that the amount of module contention is minimized given any set of n requests which are to be serviced.

One of the approaches to solve this problem is based on utilizing universal hashing, as introduced by Carter and Wegman [2]. During a simulation the MPC processors apply a hash function h chosen randomly from a class of universal hash functions H. The function h is used in order to distribute the logical addresses of the PRAM among the memory modules of the MPC. It is expected that on every simulation step the function h will spread the requests evenly among the memory modules of the MPC regardless of the memory access patterns of the PRAM. The class of universal hash functions is defined as follows.

Definition 4.1 *Let $c \in R$, $k \in N$, $m \in N$, $n \in N$. A class of hash functions $H = \{h : h : [0 \ldots m-1] \rightarrow [0 \ldots n-1]\}$ is c strongly k universal if for all $a_1, \ldots, a_k \in [0 \ldots m-1]$, pairwise distinct, and all $b_1, \ldots, b_k \in [0 \ldots n-1]$,*

$$|\{h \in H : h(a_i) = b_i \text{ for } 1 \leq i \leq k\}| \leq c|H|/n^k.$$

While simulating the PRAM we assume that the ith processor of the MPC runs the same program as the ith processor of the PRAM. The shared memory of the PRAM is divided among n memory modules of the MPC in such a way that memory module M_j, $0 \leq j < n$, contains all PRAM addresses a, $0 \leq a < m$, for which $h(a) = j$. The details of the simulation can be described as follows.

Initialization. Choose $h \in H$ at random and store h in every processor of the MPC.

Step by step simulation. For the logical address a_i generated by processor P_i of the MPC apply h to a_i and obtain the memory module index $b_i = h(a_i)$. Issue a request for a variable a_i stored at module M_{b_i}. A memory module M_j, $0 \leq j < n$, collects all requests for variables in M_j and serves them sequentially. When all requests are served the next PRAM step is simulated.

Now given the above scheme, the question arises how efficient is the simulation, or how long are the queues of requests in front of each memory module. Since the PRAM processors operate synchronously all memory requests issued in a particular step must be serviced before the simulation of the next step can begin. Therefore our objective is to minimize the length of the longest queue of requests in front of the memory modules (recall that all these requests are serviced sequentially) as it bounds the efficiency. To study it in more detail we need to define some parameters describing the length of the queues. Let $S = \{a_1, a_2, \ldots, a_p\}$, $S \subseteq [0 \ldots m-1]$, be a set of addresses of arbitrary cardinality p, and let $h \in H$. Define

$$R_{\max}(h, S) = \max_{0 \le j < n} |\{a \in S : h(a) = j\}|, \quad \text{and}$$

$$R^p_{\max} = \max_{S, |S|=p} \sum_{h \in H} R_{\max}(h, S)/|H|.$$

$R_{\max}(h, S)$ is the length of the longest queue in front of any memory module when function $h \in H$ is used and set S of addresses is issued by the processors. $\sum_{h \in H} R_{\max}(h, S)/|H|$ is the expected value of $R_{\max}(h, S)$, and $R^p_{\max}$ is the worst case of that value taken with respect to all possible sets S. Mehlhorn and Vishkin proved the following theorem [7].

Theorem 4.1 *Let H be a c strongly k universal class of functions from $[0 \ldots m-1]$ to $[0 \ldots n-1]$. Then*

$$R^p_{\max} \le k + cpn(p/n)^k/k!$$

for all $p \in N$.

Proof. Let S be defined as before and let $\mathcal{P}_i(S)$ be the probability that $R_{\max}(h, S) \ge i$, i.e. $\mathcal{P}_i(S) = |\{h \in H : R_{\max}(h, S) \ge i\}|/|H|$. Then $\mathcal{P}_p \le \ldots \mathcal{P}_k \le \ldots \mathcal{P}_2 \le \mathcal{P}_1 \le 1$ and

$$\begin{aligned} R^p_{\max} &= \max_{S, |S|=p} \sum_{i=1}^{p} \mathcal{P}_i(S) \le (k-1) \cdot 1 + (p-k+1) \cdot \max_{S, |S|=p} \mathcal{P}_k(S) \\ &\le k + p \cdot \max_{S, |S|=p} \mathcal{P}_k(S). \end{aligned}$$

Let $\mathcal{P}_{k,j}(S)$ be the probability that at least k addresses of S are mapped onto memory module j. Then we have $\mathcal{P}_k(S) \le \mathcal{P}_{k,0}(S) + \mathcal{P}_{k,1}(S) + \ldots + \mathcal{P}_{k,n-1}(S)$. Since H is c strongly k universal, for a fixed set $\{a_1, a_2, \ldots, a_k\}$ it holds

$$|\{h \in H : h(a_l) = j \text{ for } 1 \le l \le k\}| \le c \cdot |H|/n^k.$$

Hence $\mathcal{P}_{k,j}(S) \le c\binom{p}{k}/n^k$ and $\max_{S, |S|=p} \mathcal{P}_k(S) \le cn\binom{p}{k}/n^k \le cn(p/n)^k/k!$. □

Mehlhorn and Vishkin introduced the following class G of universal hash functions. Let m be a prime, let k be an integer and let

$$G = \left\{ g : g(x) = \left(\sum_{0 \le i < k} a_i x^i \right) \bmod m \right\} \tag{1}$$

for some $a_i \in [0 \ldots m-1]$ and $a_i \ne 0$ for some $i \ge 1$, be the set of all polynomials of degree at most $k-1$. Since for every $x_1, \ldots, x_k$ and $y_1, \ldots, y_k \in [0 \ldots m-1]$, $x_1, \ldots, x_k$ pairwise distinct, there is at most one non-trivial polynomial g of degree at most $k-1$ with $g(x_i) = y_i$, $1 \le i \le k$, it can be concluded that G is 1 strongly k universal. For the purpose of the simulation, class G has to be modified into the form

$$H_1 = \{h : h(x) = g(x) \bmod n\}. \tag{2}$$

Given an address x of the PRAM, $g(x)$ can be interpreted as a global address in the MPC, which corresponds to location $\lfloor g(x)/n \rfloor$ of module $h(x) = g(x) \bmod n$. Unfortunately, for polynomials of degree greater than 1, the mapping of PRAM addresses x into their internal locations in memory modules is not one to one. In other words, several addresses can be mapped into the same location in a given module. We call these addresses the *synonyms*. To handle this problem, a memory module maintains for each location $\lfloor g(x)/n \rfloor$ a table of pairs (x, data in PRAM location x) for all x mapped to that location. Thus PRAM address x is accessed by searching the table of synonyms associated with location $\lfloor g(x)/n \rfloor$.

Carter and Wegman proved the following theorem which makes possible to assess the universality of class H_1.

Theorem 4.2 *If* $H = \{h : h : [0 \ldots m-1] \to [0 \ldots m-1]\}$ *is* c *strongly* k *universal and* $r : [0 \ldots m-1] \to [0 \ldots n-1]$ *is such that* $|r^{-1}(j)| \leq \lceil m/n \rceil$ *for all* j, $0 \leq j < n$, *then class*

$$\hat{H} = \{r \circ h : h \in H\}$$

is $\hat{c}$ *strongly universal where* $\hat{c} = (n \lceil m/n \rceil / m)^k \, c \leq (1 + n/m)^k c$.

Proof. See [2]. □

From Theorems 4.1 and 4.2 it follows that the expected length of the longest queue $R^p_{\max}$ when $h \in H_1$ is used is

$$R^p_{\max} \leq k + \hat{c}pn(p/n)^k/k! \leq k + (1 + n/m)^k pn(p/n)^k/k!. \tag{3}$$

5 Parsys SN9500 architecture

The two main components of the Parsys SN9500 parallel computer are the Inmos T9000 transputer and the ST C104 (or C104 for short) packet routing device. The T9000 has much greater capabilities than any of its predecessors from the transputer family. Its peak performance is expected to be 200 MIPS and 25 MFLOPS (according to the Inmos specification of 50 MHz T9000), with links running at up to 100 Mbits/sec in each direction. The on-chip virtual channel processor which operates in parallel with the central processing unit allows physical links to be shared transparently by a large number of virtual channels. The packetization and multiplexing operations are implemented directly in hardware. The C104 allows to construct networks of very large number of fully-interconnected T9000s without use of any routing software. It has 32 bidirectional data links and two control links. It also includes a full 32×32 non-blocking crossbar switch, enabling messages to be routed from any of its links to any other link. The C104 uses "worm-hole routing" which minimizes communication latency, because the chip can start outputting a packet which is still being input. The use of a crossbar switch allows packets to be passed through all links at the same time. The C104 can route packets of any length [6].

The SN9500 contains five C104s and up to 32 fully-interconnected T9000s (Fig. 3). Each data link of each T9000 is connected to one of the C104 routing devices. Except for two of the T9000s, data link 0 of each T9000 is connected to C104[0], link 1 to C104[1], etc. This means that every T9000 is connected to every other T9000 via only one C104. The data links of the two T9000s and of the interface card are connected to the fifth C104 which in turn is connected via four pairs of its data links to each of the other routing devices [5].

6 PRAM simulators

The two kinds of simulators called SIM1 and SIM2 have been designed and implemented in the occam language on the Parsys SN9500 parallel computer. The structure of SIM1 is similar to that of the MPC (see Fig. 2). Each processor P_i and memory module M_i, $i \in [0 \ldots n-1]$, is simulated by a single occam process, with both processes corresponding to a pair (P_i, M_i) placed on a single transputer.

The second simulator, SIM2, simulates the multithreaded module parallel computer (MMPC) as shown in Fig. 4. The MMPC consists of n processor-memory module pairs (P_i, M_i), where each processor executes a number of computation threads $T_i^0, T_i^1, \ldots, T_i^{u-1}$, $i \in [0 \ldots n-1]$. The number of threads u is called the parallel slackness of the MMPC. The parallel slackness is introduced in order to maintain high utilization of the simulating processors through overlapping their local computations with global communication. More specifically, if one thread of the MMPC requests a shared memory access, then instead of the simulating processor remaining idle during the time the request is serviced, it may context-switch to another thread.

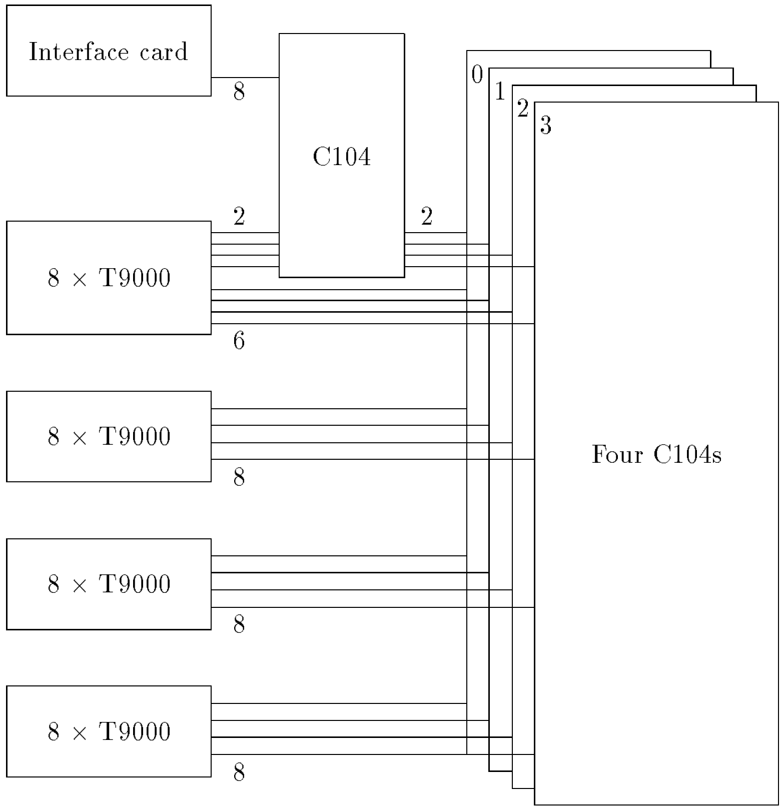

Figure 3: The Parsys SN9500 Parallel Computer

The structure of the simulator SIM2 reflects the structure of the MMPC. Namely, the ith transputer of the simulator runs u processes simulating the computation threads $T_i^0, T_i^1, \ldots, T_i^{u-1}$, and a process simulating a memory module M_i.

Simulator SIM1

As mentioned above, two occam processes run on a single transputer in SIM1. A high priority process called $Mem[i]$, $i \in [0 \ldots n-1]$, simulates a module of the shared memory M_i (Fig. 5a). It accepts memory access requests, performs the appropriate operations and sends back to the requesting process either a content of the specified memory location (for reading) or an acknowledge message (for writing). The second, low priority process called $CPU[i]$, runs the program of the ith PRAM processor. Since all transputers used in SIM1 are fully connected, each $CPU[i]$ communicates with each $Mem[i]$ directly via a pair of occam channels. In a single step a $CPU[i]$ may asynchronously perform an arbitrary number of local operations followed up by at most one access to any of the memory modules. All the $CPU[i]$s synchronize their work after each computation step as follows. First, the $CPU[i]$s, $i \in [1 \ldots n-1]$, send an appropriate message to $CPU[0]$ and wait for a reply. The $CPU[0]$ collects the messages from the $CPU[i]$s and then broadcasts the signal to these processes enabling them to resume their computations. A front-end process running on an additional transputer of SIM1 performs I/O operations, initiates the work of all processes and measures the time of computations.

Simulator SIM2

Altogether u occam processes $CPU[i][0], CPU[i][1], \ldots, CPU[i][u-1]$, $i \in [0 \ldots n-1]$, executing the MMPC threads run on a single transputer of SIM2 (Fig. 5b). Furthermore, there is an $MMU[i]$ process which acts as a concentrator of memory requests from the CPUs. A similar role plays a synchronization process $Sync[i]$ collecting all the synchronization messages from CPUs and then

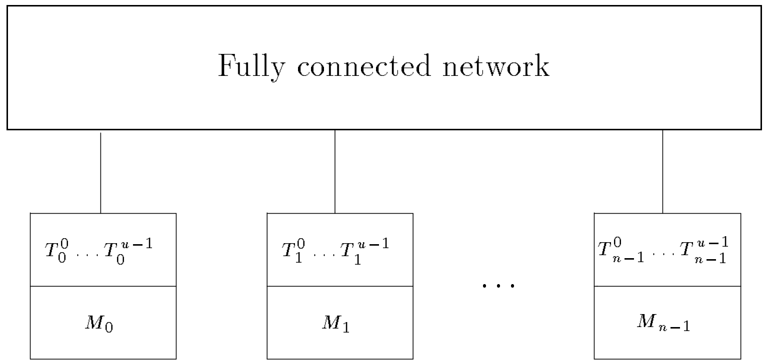

Figure 4: The multithreaded module parallel computer

transmitting to them a signal received from $Sync[0]$. Most of the communication in SIM2 is carried out through the (fast) internal transputer channels. Only the communication among the $Sync[i]$, $Mem[i]$ and $MMU[i]$ processes are effected through the transputer links. With the exception of CPUs, all other processes have high occam priority. A special scheme is applied to ensure that the local computations of a CPU is not interrupted in favour of another CPU process.

7 Matrix multiplication algorithm

In order to measure the performance of the simulators described in the previous section, an EREW PRAM matrix multiplication algorithm has been implemented (see Fig. 6). Each processor P_i, $i \in [0 \ldots n-1]$, of the algorithm computes every nth row of the resultant matrix C starting from row i. We assume that $a \leq c$.

Example. Let the constants defining the sizes of matrices A, B and C be $a = 4$, $b = 3$ and $c = 5$, and let $n = 3$. In the first stage, every processor P_i, $i \in [0 \ldots 2]$, computes value $C[i][i]$, accessing elements $A[i][k]$ and $B[k][i]$, for $k \in [0 \ldots 2]$ (see Fig. 7a). Clearly, there are no memory access conflicts among processors during the read and write operations. Then, each processor P_i computes values $C[i][(i+1) \bmod 5]$, $C[i][(i+2) \bmod 5]$, and so on. Similarly, no memory access conflicts arise during these computations, as all processors P_i and P_j, for $i \neq j$, read elements from different rows and columns of A and B (rows $A[i][\cdot]$ and $A[j][\cdot]$, and columns $B[\cdot][i]$ and $B[\cdot][j]$), and write to different elements $C[i][(i+l) \bmod 5]$ and $C[j][(j+l) \bmod 5]$, $l \in [1 \ldots 4]$. Fig. 7b shows the memory access pattern for $l = 3$ (filled circles mark elements that have been already computed, and unfilled ones the elements to be evaluated). In the next stage, once all the values in row $C[i][\cdot]$ have been evaluated, processor P_i begins the computation of values $C[i+n][\cdot]$ if row $i+n$ exists. In our example, in this stage there is enough work only for processor P_0. Therefore processors P_1 and P_2 finish their work after computing the rows $C[1][\cdot]$ and $C[2][\cdot]$, respectively, and the algorithm completes when P_0 computes all the values in row $C[3][\cdot]$. □

The EREW PRAM matrix multiplication algorithm was implemented in occam (see Fig. 8). Note that the lines in Fig. 6 marked with (*) and (**) represent a sequence of operations as shown below.

(1) Load $A[row][k]$ into *Register1*
(2) Load $B[k][col]$ into *Register2*
(3) $tmp := tmp + Register1 \times Register2$
(4) Store tmp into $C[row][col]$

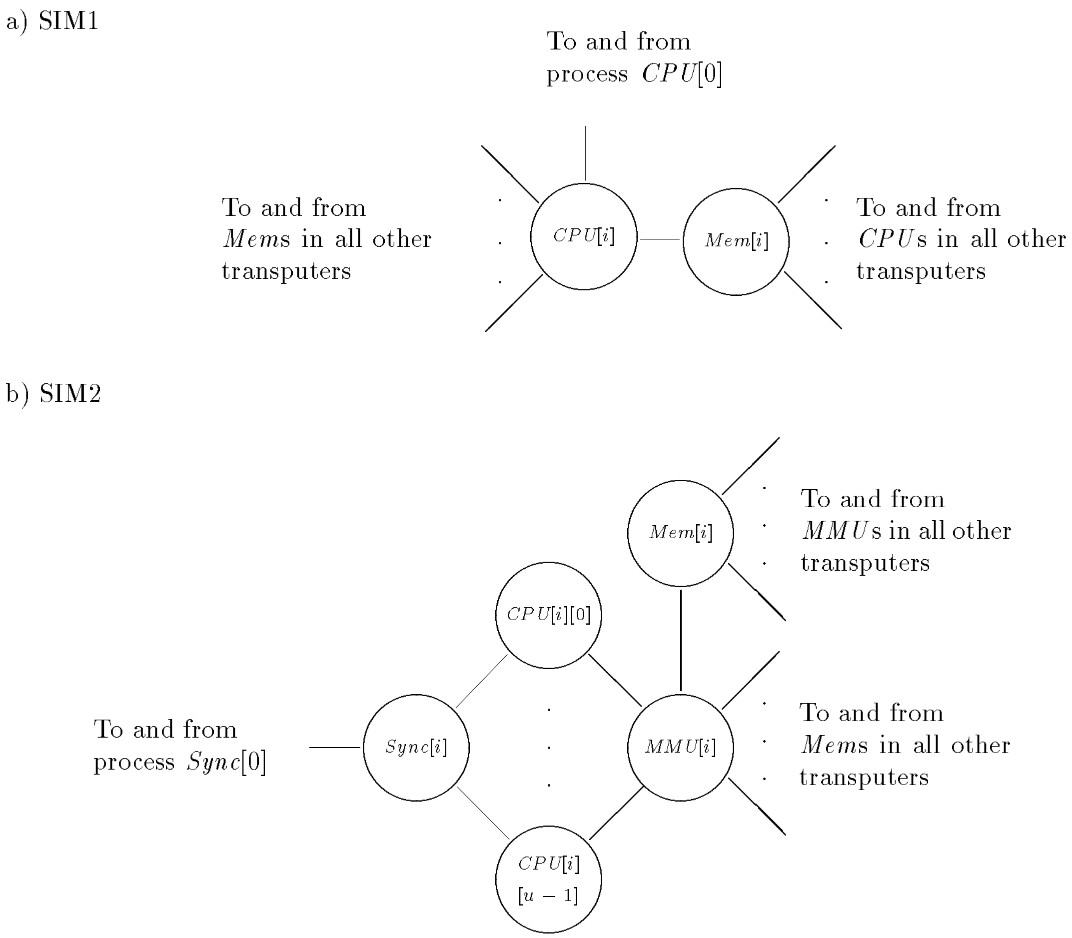

Figure 5: The structure of processes running on the ith transputer of simulators SIM1 and SIM2 (each line denotes a pair of occam channels)

```
        -- n — number of PRAM processors
        -- a, b, c — constants defining the sizes of matrices A, B and C
        A : array [0..a − 1][0..b − 1] of float;
        B : array [0..b − 1][0..c − 1] of float;
        C : array [0..a − 1][0..c − 1] of float;
        parfor i ∈ [0 . . . n − 1 do
          row := i
          while row < a do
            for j := i to (i + c − 1) do
              col := j mod c
              tmp := 0
              for k := 0 to (b − 1) do
(*)               tmp := tmp + A[row][k] × B[k][col]
(**)          C[row][col] := tmp
            end for
            row := row + n
          end while
        end parfor
```

Figure 6: The EREW PRAM matrix multiplication algorithm

The operations (1) and (2) which access the shared memory were implemented by using the *Load* procedure calls and local variables *tmpa* and *tmpb* in place of registers (see lines (a) and (b) in Fig. 8). Once the reading request is completed, the *Load* procedure executes the code synchronizing the work of all *CPU*s of a simulator. The line (4) specified above was implemented by lines (d) and (e) in Fig. 8. The *Store* procedure writes the value of its second parameter into an address of the shared memory defined by its first parameter, and then synchronizes its work with other *CPU*s.

8 Experiments

The goal of the experiments was to investigate the performance of the simulators SIM1 and SIM2. For the purpose of simulation the EREW PRAM matrix multiplication algorithm was used (see Sec. 7). The algorithm was executed on the square matrices A, B and C of size $s \times s$, where $s = 16, 32, 48, \ldots,$ 96. The simulators themselves and the matrix multiplication algorithm were implemented in occam. The experiments were carried out on the Parsys SN9500 parallel computer populated with $n = 16$ T9000 transputers (an additional transputer ran a front-end process). The T9000 Gamma silicon was applied, with a clock speed of 20 MHz and the data links configured to run at 100 Mbits/sec. The computation times were measured by making use of an internal high priority processor timer incremented every 1 μs. Each execution time measurement was averaged over 20 experiments.

The timings of the sequential version of the matrix multiplication algorithm ran on a single T9000 transputer are shown in Table 1 (s defines the size of matrices; *Ave*, σ, *Max* and *Min* denote the average execution time, the standard deviation, the maximum and minimum execution time, respectively, among the 20 experiments).

Experiments on SIM1

The two versions of the matrix multiplication algorithm were implemented. In the first one, all the matrices A, B and C were located in the shared memory. In the second version, only matrix C was stored in the shared memory, whereas the matrices A and B were copied into the local memory of each transputer. As the result, the ratio of local memory accesses and computations to shared memory accesses was increased. We shall call this ratio a *grain size*.

a)

b)

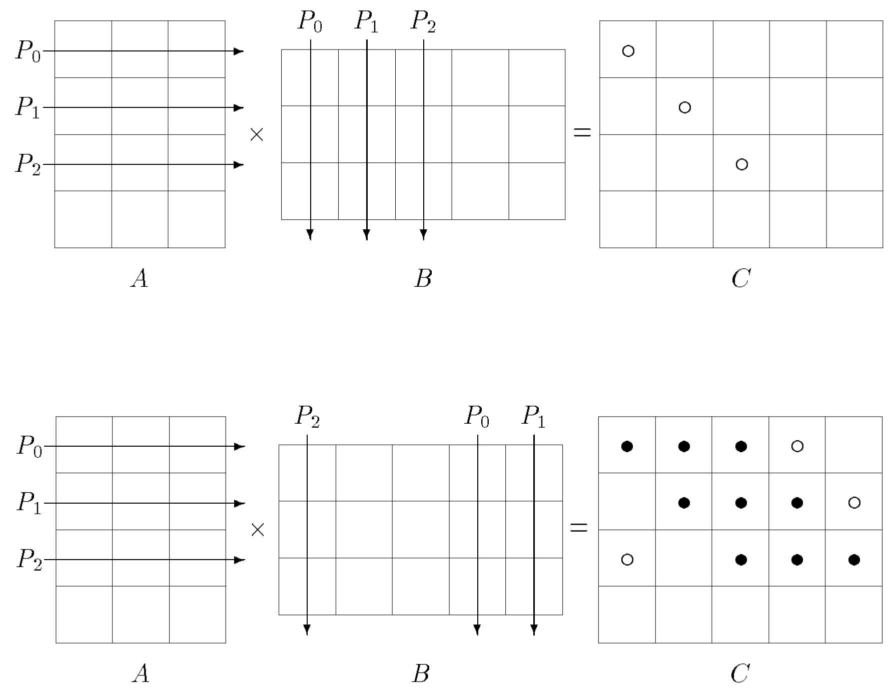

Figure 7: Memory access patterns for the EREW PRAM matrix multiplication algorithm

For these two versions of the algorithm, the two series of experiments were conducted, in which the polynomial hash functions h of degree 1 and degree $\log n - 1 = 3$ were applied, respectively (cf. eqs. (1) and (2)). Before each experiment, the new random coefficients of the polynomials were generated.

The equation (1) indicates that the expected length of the longest queue $R^p_{\max}$ is smaller if the degree of the polynomial hash function h is higher, e.g. equal to 3. In such a case one can expect that the simulation is more efficient, as the shorter queues of requests are serviced quicker. However, on the other hand, an evaluation of a polynomial of higher degree is more computationally expensive and adversely influences the efficiency. Therefore in practice the degree of the polynomial hash function should be chosen as a result of some compromise.

The results of the tests for the first version of the algorithm are shown in Tables 2 and 3, and illustrated in Fig. 9 (graphs (a) and (b)) (the graphs in Fig. 9 depict speedups defined as $S = T_1/T_s$ where T_1 is the execution time of the sequential version of the matrix multiplication algorithm on a single transputer, and T_s is a time of the PRAM simulation of the algorithm). As can be seen from graphs (a) and (b) the multiplication of matrices simulated on 16 processors lasts roughly 30 times longer than on a single processor. The reason of this low performance is a small grain size of the computations. Namely, only two (relatively cheap) local floating-point operations on the matrix elements and a few fixed-point address operations are executed for the three pairs of an (expensive) shared memory access and a global synchronization (cf. lines (a)–(e) in Fig. 8). (It is worth noting that the matrix multiplication algorithm with a small grain is a demanding test for the PRAM simulation.)

The graphs (a) and (b) also show how a degree of the polynomial hash function influences the performance of the simulation. The first degree polynomial gives a little shorter average execution times, although the times themselves are less regular and predictable. For example, for the matrices of size 32×32 the longest execution time in the first series of our experiments was more than twice as long as the longest time in the second series. Those shorter average execution times obtained for the first degree hash function mean that the evaluation time of the function dominates the time of servicing longer queues which likely arise while this function is used plus the time for dealing with

```
        INT row:
        REAL32 tmp, tmpa, tmpb:
        SEQ
          row := i
          VAL OffsetB IS a * b:
          VAL OffsetC IS (a * b) + (b * c):
          WHILE row < a
            SEQ
              VAL RowOff IS row * b:
              SEQ j = i FOR c
                VAL col IS (j REM c):
                SEQ
                  tmp := 0.0(REAL32)
                  SEQ k = 0 FOR b
                    SEQ
(a)                   Load(RowOff + k), tmpa)
(b)                   Load(((k * c) + OffsetB) + col, tmpb)
(c)                   tmp := (tmpa * tmpb) + tmp
(d)               VAL AddressC IS ((row * c) + OffsetC) + col:
(e)               Store(AddressC, tmp)
              row := row + n
```

Figure 8: The EREW PRAM matrix multiplication algorithm in occam

synonyms.

The graphs (c) and (d) in Fig. 9 illustrate the results of the experiments with the second version of the algorithm, in which only the resultant matrix C was stored in the shared memory (Tables 4 and 5 contain the corresponding measurements). In that case the shared memory was accessed only once after s floating-point multiplications and s floating-point additions. Due to the greater grain size, the speedups achieved are much better than previously.

For the matrices of size 32×32 and the hash function of degree 1, we measured the longer average execution time than for the function of higher degree. It was caused by an enormous execution time of a single experiment — almost three times bigger than the average. In that experiment the hash function of the randomly generated coefficients mapped almost all writing requests in every step of the simulation into the same module of the shared memory.

Experiments on SIM2

During the experiments on the simulator SIM2 only the polynomial hash function of degree 1 was used for the matrices of size 96×96. The results obtained are showed in Table 6. Contrary to our expectations the introducing of parallel slackness by running a number of threads on each transputer resulted in only slight improvement of the efficiency of simulations. For example, for the matrices of size 96×96 the speedup equals 5.045 on SIM1 increased to 5.887 on SIM2 (cf. Tables 4 and 6).

9 Conclusions

In the paper the problem of the randomized simulation of an EREW PRAM on an MPC was studied. The two kinds of simulators based on utilizing universal hashing were designed and implemented in the occam language. In the first simulator a number of simulating processors was equal to the number of programs of the PRAM, so that each processor ran a single program. In the second simulator, the

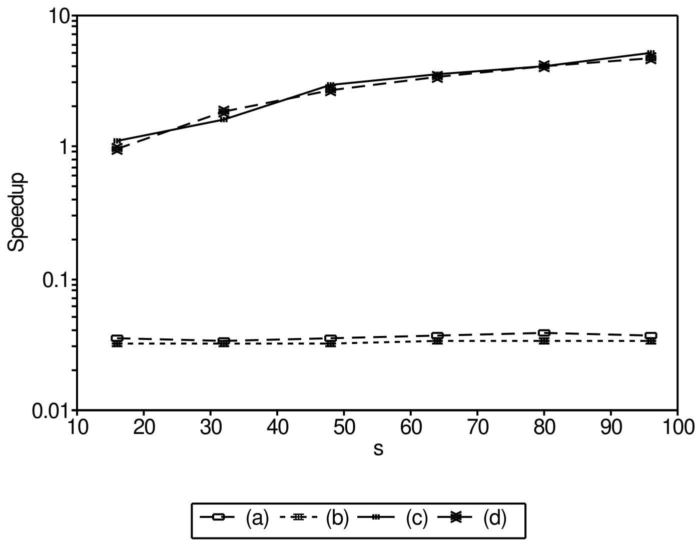

Figure 9: Speedup versus s for simulator SIM1 ($s \times s$ is the size of matrices A, B and C; (a) and (b) — matrices A, B, C in the shared memory, h is a polynomial of degree 1 and 3, respectively; (c) and (d) — only matrix C in the shared memory, h is a polynomial of degree 1 and 3, respectively)

parallel slackness was introduced by executing a number of computation threads on each simulating processor. The practical experiments on the simulators using the Parsys SN9500 parallel computer and the matrix multiplication algorithm as a running example were conducted. The results of the experiments on the first simulator indicate that the PRAM simulation is still not efficient enough to be useful in practice. We found out that the cost of the shared memory accesses (recall, implemented in the fully connected transputer network via the worm-hole routing) is relatively high in comparison with the cost of the local computations. One of the reasons for this is the fact that a size of messages exchanged during an access is small (an access request can be 5 or 9 bytes long, and a reply 1 or 5 bytes), and according to the measurements presented in [5] only roughly a half of the peak T9000 link bandwidth is attained with messages of this size. The experiments exhibit that the simulations in which the polynomial hash function of degree 1 is used are more efficient than for the function of higher degree. This means that the evaluation time of the hash function dominates the time of servicing longer queues which likely arise while a lower degree polynomial is applied. Since the simulation is randomized by nature, the simulation times vary among the experiments, especially for the polynomial of degree 1. It is explicable, for the mapping of addresses among the memory modules of the MPC is not so uniform as in the case of polynomials of higher degrees. Contrary to our expectations the introducing of parallel slackness in the second simulator improved the efficiency of simulations only in a small degree.

Acknowledgments

We wish to thank the Department of Computer Science, the University of Kent at Canterbury, Great Britain, for providing access to the parallel computing facilities.

References

[1] Akl, S.G., The Design and Analysis of Parallel Algorithms, Prentice-Hall, Englewood Cliffs, N.J., (1989).

[2] Carter, J.L., and Wegman, M.N., Universal classes of hash functions, Journ. Comput. Syst. Sci. 18, 2 (1979), 143–154.

[3] Goldschlager, L.M., A unified approach to models of synchronous parallel machines, Journ. ACM 29, 4 (1982), 1073–1086.

[4] Harris, T.J., A survey of PRAM simulation techniques, ACM Computing Surveys 26, 2 (June 1994), 187–206.

[5] Hipperson, A.M., The global communications performance of fully interconnected T9000 networks, (November 1994), manuscript.

[6] The T9000 Transputer Hardware Reference Manual, Inmos Ltd 1993.

[7] Mehlhorn, K., and Vishkin, U., Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories, Acta Informatica 21, (1984), 339–374.

[8] Schwartz, J.T., Ultracomputers, ACM Trans. Program. Syst. 2, 4 (1980), 484–521.

Appendix

s	Ave	σ	Max	Min
16	6,785	2	6,788	6,780
32	52,703	9	52,714	52,678
48	180,009	41	180,073	179,895
64	435,321	47	435,424	435,220
80	857,508	71	857,622	857,345
96	1,489,708	120	1,489,900	1,489,518

Table 1: Timings of the sequential matrix multiplication algorithm

s	Ave	σ	Max	Min	Speedup
16	196,645	70,823	487,458	167,717	0.034
32	1,563,557	547,189	3,818,708	1,338,435	0.034
48	5,108,992	1,188,367	9,740,046	4,543,431	0.035
64	11,876,992	2,128,596	19,475,317	10,785,310	0.037
80	23,086,566	3,396,817	33,650,322	21,036,007	0.037
96	40,334,448	5,499,371	5,642,2612	35,687,337	0.037

Table 2: Timings of the algorithm simulated on SIM1 (matrices A, B and C in the shared memory; h — polynomial of degree 1)

s	Ave	σ	Max	Min	Speedup
16	211,522	5,705	224,250	204,021	0.032
32	1,671,597	19,259	1,695,185	1,627,174	0.032
48	5,605,206	47,427	5,686,079	5,492,663	0.032
64	13,272,802	70,459	13,416,087	13,162,598	0.033
80	25,862,479	98,051	26,104,223	25,671,670	0.033
96	44,687,530	155,177	45,001,225	44,455,347	0.033

Table 3: Timings of the algorithm simulated on SIM1 (matrices A, B and C in the shared memory; h — polynomial of degree 3)

s	Ave	σ	Max	Min	Speedup
16	6,156	470	7,813	5,627	1.102
32	32,595	17,912	93,704	23,882	1.617
48	61,607	4,754	74,405	57,589	2.922
64	123,120	24,775	219,749	110,279	3.536
80	211,167	71,385	500,599	174,963	4.061
96	295,294	32,575	409,832	270,579	5.045

Table 4: Timings of the algorithm simulated on SIM1 (only matrix C in the shared memory; h — polynomial of degree 1)

s	Ave	σ	Max	Min	Speedup
16	7,128	262	7,756	6,587	0.952
32	29,195	464	29,973	28,386	1.805
48	69,124	707	70,313	67,942	2.604
64	129,258	956	131,227	127,856	3.368
80	212,144	1,187	215,239	209,927	4.042
96	321,115	1,599	322,934	318,090	4.639

Table 5: Timings of the algorithm simulated on SIM1 (only matrix C in the shared memory; h — polynomial of degree 3)

Matrices in the shared memory	Ave	σ	Max	Min	Speedup
A, B, C	33,744,908	3,014,267	41,615,364	29,722,720	0.044
C	253,044	9,731	280,439	244,289	5.887

Table 6: Timings of the algorithm simulated on SIM2 ($s = 96$; h — polynomial of degree 1)