Practical PRAM Simulation on 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}@star.iinf.polsl.gliwice.pl

Abstract

A randomized simulation of an EREW PRAM on a module parallel computer is studied. The simulation algorithms based on the universal hashing were implemented on T9000 transputers. The results of experiments with the polynomial hash functions are presented. The optimally efficient simulation involving parallel slackness is also investigated.

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

1 Introduction

A parallel random access machine (PRAM) is the most commonly used model for describing parallel computations [10, 4, 1]. Although the model has many advantages, it is unrealistic from the technological point of view, since on large-scale machines a parallel shared memory access can only be accomplished at the cost of a substantial time delay. Several PRAM simulation algorithms have been proposed in the literature (for the survey see [5]) which allow execution of PRAM programs on more realistic parallel machines. This paper concerns a practical randomized simulation of an exclusive read, exclusive write (EREW) PRAM on a module parallel computer (MPC).

The remainder of the paper is organized as follows. In section 2 we describe the simulation models. Section 3 presents the theoretical framework of the randomized PRAM simulation. Section 4 describes the Parsys SN9500 parallel computer which served as a platform for our experiments. In section 5 an EREW PRAM matrix multiplication algorithm used for the purpose of simulation is presented, and in section 6 the details of its PRAM executions are given. In section 7 we discuss the PRAM simulation. Section 8 describes the optimally efficient simulation, and section 9 concludes the paper.

2 Simulation models

An (n, m)-PRAM consists of n processors, P_0, P_1, ..., P_{n-1}, each of which is a random access machine (RAM), and a shared memory of m addresses, also called *variables* (Fig. 1a). The processors work synchronously, i.e. no processor will proceed with the next instruction until all have finished the previous one. In every step, a PRAM processor executes its own RAM instruction. In particular, it 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 operation (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. An EREW variant of the PRAM model allows only one processor to read or write to a particular variable of the shared memory in a given step.

*Research supported by the State Committee for Scientific Research under the grant BK-206/RAu2/96.
†Research supported by the State Committee for Scientific Research under the grant no. 8T11C 031 11.

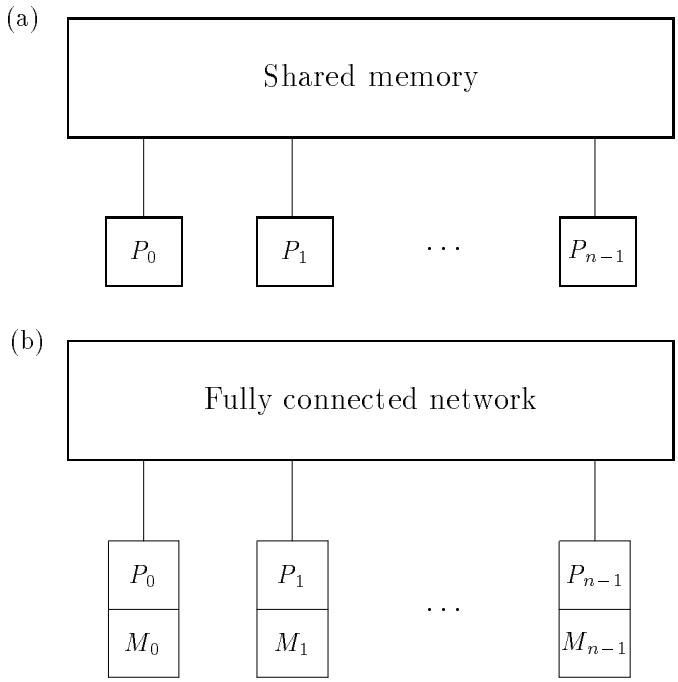

Figure 1: The (a) PRAM and (b) MPC models

An (n, m)-MPC consists of n synchronously working RAM processors, each of which has an associated memory module (Fig. 1b) [8]. We assume that a memory module is a collection of m/n variables. Every processor may access every memory module via a fully connected network[1] linking the processors. It is assumed that an access alone 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.

Note that despite of its disadvantages, like the necessity of accessing the memory locations via the network, the MPC is a more realistic computation model than the PRAM.

3 Randomized PRAM simulation

By a simulation of machine M_1 on machine M_2 we understand an algorithm that allows an instruction from machine M_1 to be executed on machine M_2. Consider an (n, m)-PRAM which is to be simulated on an (n, m)-MPC. Both machines have the same number of processors so we assume that the ith processor of the MPC will run the same program as the ith processor of the PRAM. The basic problem which must be solved by the simulation algorithm concerns the memory management, and it can be formulated as follows. 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 memory 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 n memory modules of the MPC such that the amount of module contention is minimized given any set (pattern) 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 universal class H of hash functions. The function h is used in order to distribute

[1] In such a network every processor is connected directly to each other processor.

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.

Definition. *Let $c \in \mathbb{R}$, $k \in \mathbb{N}$, $m \in \mathbb{N}$, $n \in \mathbb{N}$. 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 $x_1, x_2, \ldots, x_k \in [0 \ldots m-1]$, pairwise distinct, and all $y_1, y_2, \ldots, y_k \in [0 \ldots n-1]$,*

$$|\{h \in H : h(x_l) = y_l \text{ for } 1 \leq l \leq k\}| \leq c|H|/n^k. \ \square$$

The definition implies that when a function h randomly chosen from H is used for distributing any set of addresses $\{x_1, x_2, \ldots, x_k\}$ among the MPC modules, the probability of locating all these addresses in a specified module (or in any subset of the modules) is c/n^k.

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 x, $0 \leq x < m$, for which $h(x) = j$. The details of a single simulation of a PRAM program execution 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 x_i generated by processor P_i of the MPC apply h to x_i and obtain the memory module index $y_i = h(x_i)$. Issue a request for a variable x_i stored at module M_{y_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, each MPC processor executes its local computation and the next step is simulated.

Clearly, the simulation is randomized as the hash function h is chosen at random from class H.

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

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

for some $a_i \in [0 \ldots m-1]$ and $a_i \neq 0$ for some $i \geq 1$, be the set of all polynomials of degree at most $k-1$. Since for every $x_1, x_2, \ldots, x_k$ and $y_1, y_2, \ldots, y_k \in [0 \ldots m-1]$, $x_1, x_2, \ldots, x_k$ pairwise distinct, there is at most one non-trivial polynomial g of degree at most $k-1$ with $g(x_l) = y_l$, $1 \leq l \leq k$, it can be concluded that class G is 1 strongly k universal. For the purpose of simulation, class G is modified into the form

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

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

The quality of a simulation can be evaluated by the following metrics: slowdown, memory blowup and efficiency [5]. If a PRAM program requires time T for its execution, and if it can be executed on an MPC with the same number of processors in $O(Tf(n))$ time, then the *slowdown* is $f(n)$. A simulation algorithm usually requires an increase in the amount of memory utilized, referred to as memory blowup. If a PRAM requires M memory addresses to execute a program, and an MPC requires $O(Mg(n))$ memory addresses to execute the same program, then the *memory blowup* is $g(n)$. The third metric of the quality of a simulation is the efficiency, defined as the ratio of time-processor products. If a PRAM program takes time T on n processors, and a simulation requires time T' on n' processors, then the *efficiency* of the simulation is $E = Tn/T'n'$. Simulations where E is a constant independent of n and n' are often referred to as *optimally efficient* or *constant time-processor product* simulations. Note that the time-processor product defines the cost of parallel computations.

4 Parsys SN9500 parallel computer

The Parsys SN9500, which served as a platform for our experiments, comprises up to 32 fully connected asynchronously working IMS T9000 transputers. A single T9000 integrates a 32-bit integer processor and a 64-bit floating point processor of superscalar pipelined architectures [7]. Its peak performance is expected to be 200 MIPS and 25 MFLOPS. The four T9000 serial communication links are to provide a total of 80 Mbytes/sec bidirectional bandwidth[2]. The T9000 on-chip virtual channel processor which operates concurrently with the central processing unit allows physical links to be shared transparently by a large number of virtual channels. The message packetization and multiplexing operations necessary for virtual channel communication are implemented directly in hardware. The Parsys SN9500 is equipped with the complete routing devices (each on a single chip) which ensure that any size of T9000 network can be constructed. The device connects 32 links to each other via a 32×32 way, non-blocking crossbar switch with sub-microsecond latency. It uses *wormhole routing* which minimizes communication latency, because it can start outputting a packet which is still being input. The crossbar switch allows packets consisting of up to 32 bytes of data to be passed through all its links at the same time.

5 Matrix multiplication algorithm

In order to measure the performance of a simulation, an EREW PRAM square matrix multiplication algorithm was chosen as an example (Fig. 2). In this algorithm, each processor P_i, $0 \leq i \leq n-1$, by multiplying the apropriate rows and columns of matrices A and B computes every nth row of the resultant matrix C starting from row i. In effect, each processor determines $s \times s/n$ elements of matrix C. It is easy to see that the algorithm is indeed of the EREW type, as no memory address conflicts occur during the read and write operations.

```
-- n — number of PRAM processors
-- s × s — size of matrices A, B and C
A, B, C : array [0..s-1][0..s-1] of float;
parfor P_i, 0 ≤ i ≤ n-1 do
  row := i;
  while row < s do
    for j := i to i+s-1 do
      col := j mod s;
      tmp := 0;
      for k := 0 to s-1 do
        tmp := tmp + A[row][k] * B[k][col];
      end for;
      C[row][col] := tmp;
    end for;
    row := row + n;
  end while;
end parfor;
```

Figure 2: The EREW PRAM square matrix multiplication algorithm

[2] According to the INMOS specification of 50 MHz processor.

6 PRAM execution

The PRAM execution of the matrix multiplication algorithm was mimicked on the Parsys SN9500 parallel computer populated with $n = 16$ transputers. 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 matrix multiplication algorithm was coded in occam (Fig. 3) and executed on the matrices A, B and C of size $s \times s$, where $s = 16, 32, 48, \ldots, 96$. The matrices A and B were copied into the local memory of the transputers

```
INT row:
REAL32 tmp, tmpa, tmpb:
[s][s] REAL32 A, B, C:
SEQ
  row := i
  WHILE row < s
    SEQ
      SEQ j = i FOR s
        VAL col IS (j REM s):
        SEQ
          tmp := 0.0(REAL32)
          SEQ k = 0 FOR s
            tmp := tmp + (A[row][k] * B[k][col])
          C[row][col] := tmp
      row := row + n
```

Figure 3: The matrix multiplication algorithm in occam for PRAM execution

before an execution. Each transputer computed its own portion of matrix C and stored it in the local memory. The sizes $s \times s$ of the matrices were chosen so that the processors were equally loaded. Clearly, the processors do not communicate with each other while performing their tasks. Therefore one may assume that the algorithm was executed in the similar way as it would be executed in a real PRAM equipped with processors of T9000 class, and a shared memory of the speed equal to the speed of the transputer local memory. The only difference is that the transputers executed their tasks asynchronously, whereas in a real PRAM this execution would be synchronous.

The times of the PRAM executions of the matrix multiplication algorithm are shown in Table 1. The times were measured by making use of an internal high priority transputer timer incremented every 1 μs. Ave, σ, Max and Min denote the average execution time, the standard deviation, the maximum and minimum execution time, respectively, among the $n = 16$ transputers.

s	Ave	σ	Max	Min
16	414.06	0.46	415	414
32	3,096.88	0.88	3,098	3,096
48	10,410.63	5.62	10,430	10,406
64	24,887.56	6.87	24,904	24,869
80	48,907.19	22.88	48,926	48,860
96	84,981.75	47.39	85,064	84,838

Table 1: Timings of the PRAM execution of the matrix multiplication algorithm

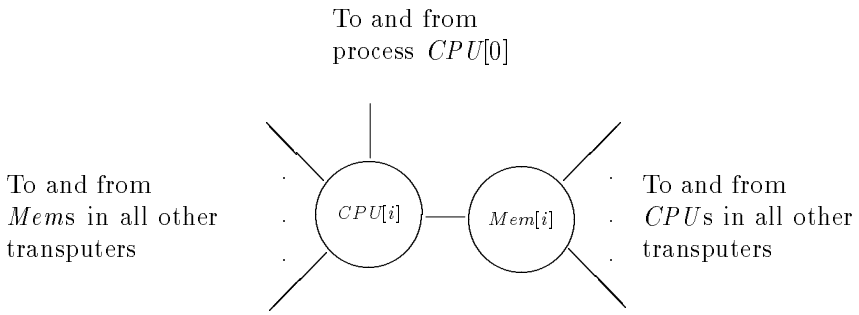

Figure 4: Processes running on the ith transputer of the simulator (each line denotes a pair of occam channels)

7 PRAM simulation

As mentioned before, our goal was to simulate a PRAM on an MPC. Since the real MPC was not physically available, we have designed and implemented its simulator on the Parsys SN9500 parallel computer. The structure of the simulator is similar to that of the MPC (Fig. 1b). Each processor P_i and memory module M_i, $0 \leq i \leq n-1$, is simulated by a single occam process, with both processes corresponding to a pair (P_i, M_i) placed on a single transputer. A high priority process called $Mem[i]$, $0 \leq i \leq n-1$, simulates a module of the shared memory M_i (Fig. 4). 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 MPC processor. Since all transputers used by the simulator 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, $1 \leq i \leq n-1$, 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 the simulator performs I/O operations, initiates the work of all processes and measures the time of computations.

In order to be executed on the MPC, the matrix multiplication algorithm was written in occam (Fig. 5). Note that the following lines of the algorithm in Fig. 2

```
for k := 0 to s − 1 do
  tmp := tmp + A[row][k] ∗ B[k][col] ;
end for ;
C[row][col] := tmp ;
```

represent the operations as shown below

```
      for k := 0 to s − 1 do
(1)     Load A[row][k] into tmpa ;
(2)     Load B[k][col] into tmpb ;
(3)     tmp := tmp + tmpa * tmpb ;
      end for ;
(4)   Store tmp into C[row][col] ;
```

```
INT row:
REAL32 tmp, tmpa, tmpb:
SEQ
  row := i
  VAL OffsetB IS s * s:
  VAL OffsetC IS 2 * (s * s):
  WHILE row < s
    SEQ
      VAL RowOff IS row * s:
      SEQ j = i FOR s
        VAL col IS (j REM s):
        SEQ
          tmp := 0.0(REAL32)
          SEQ k = 0 FOR s
            SEQ
(a)           Load((RowOff + k), tmpa)
(b)           Load(((k * s) + OffsetB) + col, tmpb)
(c)           tmp := (tmpa * tmpb) + tmp
(d)       VAL AddressC IS ((row * s) + OffsetC) + col:
(e)       Store(AddressC, tmp)
      row := row + n
```

Figure 5: The matrix multiplication algorithm in occam for MPC execution

The operations (1) and (2) which access the shared memory were implemented by using the *Load* procedure calls (see lines (a) and (b) in Fig. 5). Once a reading request is completed, the *Load* procedure executes the code synchronizing the work of all CPUs of the simulator. The line (4) was implemented by lines (d) and (e) in Fig. 5. 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 CPUs.

Using the occam code of the matrix multiplication algorithm (Fig. 5) the two series of experiments were conducted for $n = 16$ transputers, in which the polynomial hash functions of the form $h(x) = g(x) \bmod 16$ and $g(x) = \left(\sum_{0 \le i < k} a_i x^i\right) \bmod 32771$ were applied, for $k = 1$ and $k = \log n - 1 = 3$. Before every experiment, the new random coefficients of the polynomials were generated. If the degree of the polynomial hash function h is higher, e.g. equal to 3, one can expect that the simulation is faster, as the hash function distributes the addresses among the memory modules more evenly and the shorter queues of requests which arise are serviced quicker. However, on the other hand, an evaluation of a polynomial of higher degree is more computationally expensive and adversely influences the speed of the simulation. Therefore in practice the degree of the polynomial should be chosen as a result of some compromise.

The timings (in microseconds) obtained during the tests are shown in Tables 2a and 2b, and illustrated in Fig. 6. The graphs in Fig. 6 depict slowdowns defined as $S = T_{\mathrm{MPC}}/T_{\mathrm{PRAM}}$, where T_{MPC} is the time of the MPC execution of the algorithm (Tables 2a and 2b), and T_{PRAM} is the PRAM execution time of the algorithm (Table 1).

As can be seen from graphs (a) and (b) the multiplication of matrices on the MPC lasts roughly 500 times longer than on the PRAM. The reason for this low performance is the small *grain size* of computations, which we define as the ratio of local memory accesses and computations to shared memory accesses. 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)

(a) polynomial h of degree 1

s	Ave	σ	Max	Min	Slowdown
16	196,645	70,823	487,458	167,717	474.9
32	1,563,557	547,189	3,818,708	1,338,435	504.9
48	5,108,992	1,188,367	9,740,046	4,543,431	490.7
64	11,876,992	2,128,596	19,475,317	10,785,310	477.2
80	23,086,566	3,396,817	33,650,322	21,036,007	472.0
96	40,334,448	5,499,371	5,642,2612	35,687,337	474.6

(b) polynomial h of degree 3

s	Ave	σ	Max	Min	Slowdown
16	211,522	5,705	224,250	204,021	510.8
32	1,671,597	19,259	1,695,185	1,627,174	539.8
48	5,605,206	47,427	5,686,079	5,492,663	538.4
64	13,272,802	70,459	13,416,087	13,162,598	533.3
80	25,862,479	98,051	26,104,223	25,671,670	528.8
96	44,687,530	155,177	45,001,225	44,455,347	525.8

Table 2: Timings of the MPC execution of the algorithm

shared memory access and a global synchronization (cf. lines (a)–(e) in Fig. 5). It is worth noting that the matrix multiplication algorithm with a small grain of computations is a demanding test for the PRAM simulation.

The graphs in Fig. 6 also show how a degree of the polynomial hash function influences the performance of the practical 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 (compare also the standard deviations in Tables 2a and 2b). 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 synonyms.

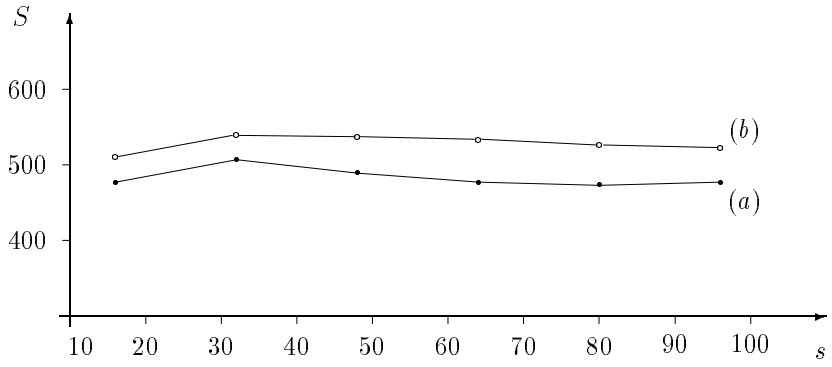

Figure 6: Slowdown versus s ($s \times s$ is the size of matrices A, B and C; (a) and (b) — h is a polynomial of degree 1 and 3, respectively)

8 Optimally efficient simulation

If the efficiency of the simulation is a constant independent of the numbers of the PRAM and MPC processors, n and n', i.e.

$$E = \frac{Tn}{T'n'} = O(1)$$

then we say that the simulation is optimally efficient. We have investigated the efficiency of the simulation for the matrix multiplication algorithm. First, the algoritm was executed in the PRAM fashion using $n = 16$ transputers. Then, it was run on the MPC which was simulated by making use of $n' = 2$, 4 and 8 transputers. The multiplied matrices were of size $s \times s$, where $s = 16$, 32, 48, ..., 96, which means that during the PRAM execution each transputer carried out a single task, or *computation thread*, of evaluating $s \times s/16$ elements of matrix C. These 16 threads were then spreaded equally among n' transputers, i.e. for $n' = 2$, 4 and 8, a single transputer executed (via time sharing) 8, 4 and 2 threads, respectively. In fact, one additional thread was run on each transputer which simulated a module of the shared memory.

The number of computation threads is called the *parallel slackness*, and it is introduced to maintain high utilization of the simulating processors through overlapping their local computations with global communications. Namely, if one thread requests a shared memory access, then instead of the MPC processor remaining idle during the time the request is serviced, it could context-switch to another thread.

The test results are shown in Table 3 and illustrated in Fig. 7. It can be seen that the efficiency is generally low and slightly exceeds 0.01. This means that the cost of a simulated execution of the matrix multiplication algorithm is about 100 times higher than the cost of its execution on the PRAM. Furthermore, for every s the efficiency winds down as the number of transputers increases. One reason for this is a drop of parallel slackness which occurs when the number of transputers goes up. As a consequence, the overlapping of local computations and global communications within the processors declines what adversely influences the efficiency. Another reason for this falling efficiency is the necessity of synchronization of transputers after every computation step. As explained in section 7, it is performed by the root transputer through collecting messages from all the transputers and then sending them a signal to resume work. Surely the cost of such a synchronization is higher for the larger number of transputers. Yet another, more subtle reason is locality of memory references. Recall that the address space of the shared memory is divided into the memory blocks. Each of these blocks is serviced by a single transputer running an appropriate $Mem[i]$ process (cf. section 7). Now if a shared memory reference arising in a given transputer leads to the memory block handled by the same transputer then such a reference is serviced faster, as in this case there is no need for a communication via the network. Clearly, the probability of occurring of local references grows when the number of simulating processors is smaller.

9 Conclusions

The problem of the randomized simulation of an EREW PRAM on an MPC was studied. The practical experiments indicate that the PRAM simulation is still not fast enough to be useful in practice, at least with respect to the transputer hardware. We found out that the cost of a (simulated) shared memory access (recall, implemented in the fully connected transputer network via wormhole routing) is relatively high in comparison both with the cost of a local memory access and a local computation. One of the reasons for this is the fact that a size of messages exchanged during a shared memory access is small — an access request can be 5 or 9 bytes long, and a reply 1 or 5 bytes. According to the measurements presented in [6] only roughly a half of a peak link bandwidth of the T9000 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 slightly faster 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 (sometimes very wildly), especially for the polynomial of degree 1. It is explicable, for the mapping of addresses among the memory modules of the MPC is

s	n	n'	T	T'	Tn	$T'n'$	E
16	16		414		6,624		
		2		210,484		420,968	0.01574
		4		122,175		488,700	0.01355
		8		81,859		654,872	0.01011
32	16		3,097		49,552		
		2		1,656,904		3,313,808	0.01495
		4		959,008		3,836,032	0.01292
		8		643,484		5,147,872	0.00963
48	16		10,411		166,576		
		2		5,557,815		11,115,630	0.01499
		4		3,223,276		12,893,104	0.01292
		8		2,154,302		17,234,416	0.00967
64	16		24,888		398,208		
		2		13,096,187		26,192,374	0.01520
		4		7,578,480		30,313,920	0.01314
		8		5,068,336		40,546,688	0.00982
80	16		48,907		782,512		
		2		25,528,184		51,056,368	0.01533
		4		14,805,265		59,221,060	0.01321
		8		9,899,037		79,192,296	0.00988
96	16		84,982		1,359,712		
		2		44,171,758		88,343,516	0.01539
		4		25,592,990		102,371,960	0.01328
		8		17,116,490		136,931,920	0.00993

Table 3: Results of the efficiency tests

not so uniform as in the case of polynomials of higher degrees. The application of a special-purpose hardware for the hash function evaluation could not only speed up the efficiency of simulations but also make the simulation times more stable and predictable through employing higher degree polynomials.

As to the simulations with a smaller number of processors ($n' < n$), we found out that the efficiency of these simulations is generally low and does not exceed 0.01. The efficiency also declines when a number of transputers increases. It is caused by a dwindling parallel slackness and a rising cost of synchronization incurred for the larger number of transputers. Another reason, as explained in details in section 7, is the lower probability of occurring of local memory references which are serviced faster.

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.

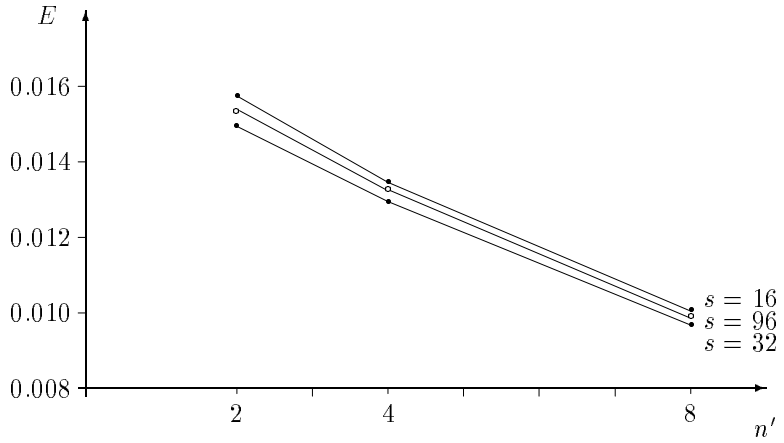

Figure 7: Efficiency of the simulation versus n'

[3] Czech, Z.J., Mikanik, W., Randomized PRAM simulation using T9000 Transputers, Proc. of the International Conf. on High-Performance Computing and Networking (HPCN), Brussels, Belgium, April 1996; Lectures Notes in Computer Science 1067, Springer-Verlag, 736-743.

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

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

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

[7] The T9000 Transputer Hardware Reference Manual, INMOS Ltd 1993, 1st ed.

[8] 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.

[9] Reingold, E.M., Nievergelt, J., and Deo, N., Combinatorial Algorithms. Theory and Practice, Prentice-Hall, Englewood Cliffs, N.J., (1977).

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