Multithreaded Parallel Computer Model with Performance Evaluation *

J. Cui[1], J. L. Bordim[1], K. Nakano[1], T. Hayashi[1], N. Ishii[2]

[1] Department of Electrical and Computer Engineering
Nagoya Institute of Technology
[2] Department of Intelligence and Computer Engineering
Nagoya Institute of Technology
Showa-ku, Nagoya 466-8555, Japan

Abstract. The main contribution of this work is to introduce a multithreaded parallel computer model (MPCM), which has a number of multithreaded processors connected with an interconnection network. We have implemented some fundamental PRAM algorithms, such as prefix sums and list ranking algorithms, and evaluated their performance. These algorithms achieved optimal speedup up to at least 16 processors.

1 Introduction

The Parallel Random Access Machine (PRAM) is a standard parallel computer model with a shared memory [2, 5]. The PRAM has a number of processors (PE for short) working synchronously and communicating through the shared memory. Each processor is a uniform-cost Random Access Machine (RAM) with standard instruction set. This model essentially neglects any hardware constraints which a highly specified architecture would impose. In this respect, the model gives free rein in the presentation of algorithms by not admitting limitations on parallelism which might be imposed by specific hardware. However, the PRAM is an unrealistic parallel computer model, because it has a shared memory uniformly accessed by processors.

Parallel computers based on the principle of multithreaded have been developed, such as HEP [12], Monsoon [9, 10], Horizon [7], and MASA [3]. In this paper, we introduce the *multithreaded parallel computer model* (MPCM for short), which has a number of *multithreaded processors* (MP for short) connected with an interconnection network. This model is based on the multithreaded computer architecture [1, 8, 6, 11, 12, 13] which allows fast context-switching, and communicates through message passing. Each multithreaded processor is highly pipelined, and runs several threads in time-sharing manner. Furthermore, processors switch threads in every clock cycle.

* This work is in part supported by the Grant-in-Aid for Scientific Research (B)(2) 10205209 (1999) from the Ministry of Education, Science, Sports and Culture of Japan.

The main contribution of this work is to develop an MPCM simulator, implement fundamental PRAM algorithms on it, and evaluate their performance. The PRAM algorithms we have implemented include prefix sums algorithm and list ranking algorithm. We coded these algorithms using MPCM machine instructions, and evaluated the number of execution cycles performed. The algorithms we have implemented achieved linear speedup up to at least 16 processors.

2 Multithreaded parallel computer model

An MP has several register sets, each of which has a program counter (PC) and several (say, 32) registers. Each register can store several (say, 32) bits. Although the multithreaded processor has several register sets, it has a single control unit that fetches, decodes, executes an instruction, and writes the result. Thus, an instruction specified by the PC of each register set is processed by the control unit in turn. The instruction may read/write the registers and the local memory of the MPs. However, it cannot directly access the registers in the other register sets. Each execution flow performed by a register set is regarded as a thread. All register sets (threads) share the same program and each register set executes these machine instructions sequentially.

Let $R_1, R_2, \ldots, R_m$ denote m register sets of the MPs. Suppose that each register set executes T instructions of a program. Let I_i^t ($1 \leq i \leq m$, $1 \leq t \leq T$) denote the instruction that R_i executes at time t. Note that a sequence $I_i^1, I_i^2, \ldots, I_i^T$ is not a program code for PE_i; this is a resulting sequence of the instructions executed by PE_i. The behavior of MP is described as follows:

for $t = 1$ **to** T **do**
  for $i = 1$ **to** m **do**
    R_i executes I_i^t

A *phase* is a single iteration of the internal loop, that is, in a single phase the MP executes m instructions $I_1^t, I_2^t, \ldots, I_m^t$.

In most commercial microprocessors, the pipeline mechanism is employed to decrease the average instruction execution time. An instruction process is partitioned into several stages. The pipeline may stall due to the dependency of executed instructions. On the other hand, in the multithreaded processor, instructions executed in a phase have no dependency. This fact allows us to have a large number of pipeline stages.

3 Multithreaded parallel computer model simulator

Our MPCM simulator uses the *DLX* RISC architecture instruction set [4]. We added several instructions for communication and synchronization between processors to the instruction set. These instructions include *network write*, *network read* and *barrier synchronization*. We have implemented MPs having 40 pipeline stages, which is reasonable for highly pipelined arithmetic computation unit [4].

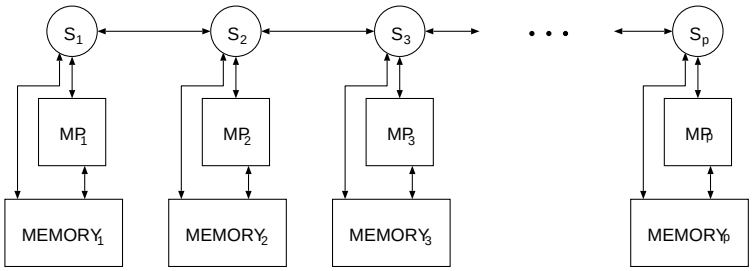

Fig. 1. The multithreaded parallel computer model.

We use one-dimensional linear array interconnection network model as shown in Figure 1. We chose this network topology because it has the weakest communication ability, and also because it is widely used. This weak ability will focus on the power of the MPCM. Each processor is connected to a switch and each neighboring switch is connected by a bidirectional link. A packet can transfer only 32 bit(1 word) to the left or to the right direction. We assume that it takes 4 cycles to transfer a packet to the neighboring switch. This is reasonable because the frequency of the clock of VLSI chips is 3-5 times faster than that of the mother board. From this assumption, a packet transfer from MP_i to MP_j needs at least $4|i-j|$ cycles. Furthermore, we need to consider the case that one processor and its right neighbor will transfer to its left processor simultaneously. Since each communication link transfers one packet at the same time, one of these two packets can not be transferred, in this case, the packet with further destination is transferred first. Here, we assume that each switch has only an unbounded buffer and each switch only process the packet at the first position of buffer at any time.

When we implement PRAM algorithms in our MPCM simulator, n PRAM processors $PE_0, PE_1, PE_2, \cdots, PE_{n-1}$ are equally assigned to p multithreaded processors $MP_0, MP_1, MP_2, \cdots, MP_{p-1}$. Thus, each processor $MP_i (0 \leq i \leq p-1)$ has n/p register sets $R_{i,0}, R_{i,1}, \cdots, R_{i,n/p-1}$, and these register sets perform tasks that the PRAM processors $PE_{(i-1)\cdot n/p}, PE_{(i-1)\cdot n/p+1}, \cdots, PE_{i\cdot n/p-1}$ would execute.

Furthermore, memory assignment is important in multithreaded architecture model. In our research, it is known which memory location each processor should process. For example, for an array $a[0, n-1]$ of input data, processor PE_i processes $a[i]$, so shared memory module on the PRAM can be simply divided into p local memory modules. We assume that the input data is allocated to corresponding local memory in advance and ignore the time for input and output.

4 PRAM algorithms implemented in the MPCM

In this section, we briefly describe PRAM algorithms that we have implemented in our simulator.

4.1 Prefix sums algorithm

For n values $a_0, a_1, \cdots, a_{n-1}$, the prefix sums problem asks to compute the values of $p_i = a_0 + a_1 + \cdots + a_i$ for every i $(0 \leq i \leq n-1)$. For example, given a sequence of integer numbers $A = \{3, 1, 0, 4, 2\}$, the prefix sums are $\{3, 4, 4, 8, 10\}$.

The PRAM prefix sums algorithm that we have implemented is as follows: Each $PE_i (0 \leq i \leq n-1)$ is used to update $a[i]$.

for $j = 0$ to $\lceil \log n \rceil - 1$ do
  for $i = 1$ to $n-1$ do in parallel
    if $i - 2^j \geq 0$ then $a[i] = a[i] + a[i - 2^j]$

The above prefix sums algorithm runs in $O(\log n)$ time using n processors [5]. However, this algorithm is not work optimal because the product of the computing time and the number of processors is $O(n \log n)$. By assigning two or more data to each processor, cost optimization can be achieved. The input data of size n is equally partitioned into s groups such that the ith $(1 \leq i \leq s)$ group is $A_i = \{a((i-1) \cdot \frac{n}{s} + 1), a(i \cdot \frac{n}{s} + 2), \ldots, a(i \cdot \frac{n}{s})\}$. Then, the (local) prefix sums within each group are computed in $O(\frac{n}{s})$ time using a single processor. After that, the prefix sums of the sums of $A_0, A_1, \ldots A_{s-1}$ are computed in $O(\log s)$ time using the prefix sums algorithm described above. Finally, the prefix sums are added to the local prefix sums in obvious way. Clearly, this algorithm runs in $O(n/s + \log s)$ time using s processors.

4.2 List ranking algorithm

Consider a linked list of n nodes whose order is specified by an array p such that $p[i]$ contains a pointer to the next node i in the list, for $1 \leq i \leq n$. We assume that $p[i] = i$ when i is the tail of the list. The list ranking problem is to determine the distance of each node from the tail of the list.

The PRAM algorithm to determine the position of each node on a linked list is as follows.

```
for i = 0 to n − 1 do in parallel
    if p[i] = i then r[i] = 0 else r[i] = 1
for j = 1 to ⌈log n⌉ do
    for i = 1 to n − 1 do in parallel
        begin
            r[i] = r[i] + r[p[i]]
            p[i] = p[p[i]]
        end
```

This algorithm runs in $O(\log n)$ time using n processors [5].

5 Performance evaluation

This section shows the performance evaluation of the above PRAM algorithms using our simulator.

Table 1 shows the number of cycles for n input data, p processors and n/p register sets/processor. We can verify that with $n = 512$, the speedup is almost linear up to 16 processors. This is because the number of threads in each processor is larger than the pipeline stages, and hence, enough instructions can be provided to the MP control unit. On the other hand, when $p \geq 32$, the number of threads range from 1 to 16, and the algorithm runs in 8000 to 9000 cycles regardless of the number of processors. When $p = 512$, the speedup turns down due to a large amount of communication among processors. In the case of $n = 8k(8192)$, linear speedup is nearly achieved up to 256 processors. If we increase input data, and use the same number of processors, the number of register sets n/p increase, and the number of threads put into the pipeline also increase.

Table 1. The number of cycles of the prefix sums algorithm

n \ p	1	2	4	8	16	32	64	128	256	512
512	95250	51712	26175	13471	9424	8996	8698	8510	8318	8910
1k	211986	114176	57727	29631	15711	11196	10760	10470	10350	10244
2k	466962	249856	126207	64639	34111	18975	14040	13564	13330	13338
4k	1042411	542720	273919	140031	73599	40639	24287	18932	18488	18366
8k	2252780	1193936	590847	301567	157951	94847	59455	33759	27968	27636
16k	4841452	2555857	1292235	646143	337407	184063	116095	78271	51423	45340

Table 2 shows the number of cycles of the optimal prefix sums algorithm for 64k input data, p processors and m register sets/processor. To each register set, $64k/mp$ input data is assigned. The smallest number of cycles is achieved with $m = 64$ and p ranging from 1 to 16. That is because not only 64 is larger than the pipeline stages of 40, but also the input data assigned to each register set for $m = 64$ is larger than with $m > 64$. If the number of processors is larger than 16, the smallest number of cycles varies according to the amount of threads, input data assigned to each register set and communication overhead.

Table 2. The number of cycles of the optimal prefix sums algorithm: 64k input

m \ p	1	2	4	8	16	32	64	128	256	512	1024	2048
1	16256573	16259137	8135003	4073865	2044111	1029941	523467	270796	145053	82942	53103	40432
2	16259380	8135462	4074372	2044666	1030548	524094	271392	1455588	83346	53248	40062	37132
4	8135700	4074540	2044852	1030744	524300	271608	145804	83536	53352	39948	36544	41480
8	4078820	2045028	1030912	524500	271768	145956	83656	53404	39872	36212	40632	55569
16	2045352	1031152	524708	272052	146241	83956	53681	40116	36337	40516	54960	86856
32	1031532	525092	272432	146640	84380	54144	40612	36808	40892	55064	94656	159416
64	806591	421440	226655	130015	82655	60511	52319	54047	66655	96927	160286	–
128	840063	452992	259263	163519	117183	96703	100031	111964	136255	196956	–	–
256	904959	517888	325503	230783	209772	188580	177023	204974	254828	–	–	–
512	1036031	649728	458495	364287	320255	318828	325375	369919	–	–	–	–

Table 3 shows the number of cycles of the list ranking algorithm for $n-$node linked list and p processors. Linear speedup is nearly achieved up to 16 proces-

Table 3. The number of cycles of the list ranking algorithm

n \ p	1	2	4	8	16	32	64	128	256	512
512	129024	64205	38509	24726	19324	18635	18527	18860	19480	25834
1k	283648	141515	84865	53660	37403	31412	30812	31076	32679	33890
2k	618496	309805	185579	117971	82408	64463	57732	57181	58340	62123
4k	1361887	665855	402404	258380	181745	143661	124289	117539	117564	120900
8k	2924511	1450445	868441	551296	386144	303014	261414	240420	233131	235939
16k	6250463	3108902	1865968	1201185	845634	668873	578670	533555	512162	505642

sors. However, for $p \geq 32$, the speedup raises very slowly, because communication among processors is so random that it spends most execution time. In order to decrease the communication overhead we need to employ higher performance networks, such as mesh and hypercube.

References

1. M. Amamiya, H. Tomiyasu, S. Kusakabe, Datarol: a parallel machine architecture for fine-grain multithreading, *Proc. 3rd Working Conference on Massively Parallel Programming Models*, 151–162, 1998.
2. A. Gibbons and W. Rytter, *Efficient Parallel Algorithm*, Cambridge University Press, 1998.
3. R. H. Halstead and T. Fujita, MASA: A multithreaded processor architecture for parallel symbolic computing, *Proc. 15th International Symposium on Computer Architecture*, 443–451, 1988.
4. John L. Hennessy, and David A. Patterson, *Computer Architecture–A Quantitative Approach*, Morgan Kaufmann, 1990.
5. J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.
6. Robert A. Iannucci ed., *Multithreaded Computer Architecture: A Summary of the state of the Art*, Kluwer Academic, 1990.
7. J. T. Kuehn and B. J. Smith, The Horizon supercomputing system: architecture and software, Proc. Supercomputing 88, 28–34, 1988.
8. M. Loikkanen and N. Bagherzadeh, A fine-grain multithreading superscalar architecture, *Proc. of Conference on Parallel Architectures and Compilation Techniques*, 1996.
9. G. M. Papadopoulos and D. E. Culler, Monsoon: an explicit token-store architecture, *Proc 17th International Symposium on Computer Architecture*, 82–91, 1990.
10. G. M. Papadopoulos and K. R. Traub Multithreading: A revisionist view of dataflow architecture, *Proc 18th International Symposium on Computer Architecture*, 342–351, 1991.
11. R. G. Prasadh and C.-L Wu, A Benchmark Evaluation of a Multi-Threaded RISC Processor Architecture, *Proc. of International Conference on Parallel Processing*, pp. 84–91, 1991.
12. B. J. Smith, Architecture and applications of the HEP multiprocessor system, *Proc. of SPIE –Real-Time Signal Processing IV*, Vol. 298, Aug, 1981
13. J.-Y. Tsai and P. C. Yew, The Superthreaded Architecture: Thread Pipelining with Run-time Data Dependence Checking and Control Speculation *Proc. of Conference on Parallel Architectures and Compilation Techniques*, 1996.