Emulating a PRAM on a Parallel Computer

Holger Blaar[1], Jörg Keller[2], Christoph Keßler[3], and Bert Wesarg[1]

[1] Universität Halle, Institut für Informatik, 06099 Halle, Germany
[2] FernUniversität in Hagen, Fak. Math. und Informatik, 58084 Hagen, Germany
[3] Linköpings Universitet, Dept. of Computer and Inf. Science, 58183 Linköping, Sweden

Abstract. The PRAM is an important model to study parallel algorithmics, yet this should be supported by the possibility for implementation and experimentation. With the advent of multicore systems, shared memory programming also has regained importance for applications in practice. For these reasons, a powerful experimental platform should be available. While the language Fork with its development kit allows implementation, the sequential simulator restricts experiments. We develop a simulator for Fork programs on a parallel machine. We report on obstacles and present speedup results of a prototype.

Key words: Parallel Random Access Machine, Parallel Discrete Event Simulation, PRAM Simulation

1 Introduction

The parallel random access machine (PRAM) has been an important concept to study the inherent parallelism of problems, and devise parallel algorithms, without having to adapt the solution too much towards a particular underlying machine architecture. The conjecture that PRAMs are highly unpractical could be refuted by building a PRAM machine in hardware, the SB-PRAM, and by providing the programming language Fork with a compiler, a runtime system, the PAD library of basic parallel algorithms and data structures, and an SB-PRAM simulator program to support program development and debugging. See [1] for a detailed introduction into all of these issues.

Yet, while our experiences with programming on the SB-PRAM were quite positive, the SB-PRAM machine could not be maintained longer and has been disassembled in 2006. The simulator program being a sequential code, simulation of larger program instances would take days. As parallel algorithms and shared memory programming currently gain renewed interest because of the advent of multicore CPUs, and teaching such a class should contain practical assignments beyond toy problems [2], we decided to provide a new, powerful platform for Fork programming: a program to emulate the SB-PRAM instruction set architecture on a contemporary parallel machine.

As the code for the SB-PRAM processor instruction set was already available, we concentrated on simulation of and access to the shared memory. While there is a wealth of implementations and literature of software distributed shared memory (S-DSM) [3], they try to exploit locality by replication, i.e. caching. Yet, parallelism and shared memory access in PRAM algorithms normally is very fine grained and highly non-local, so that these approaches show poor performance on PRAM algorithms, which necessitates a customized implementation. Simulating a PRAM on realistic parallel computers has been extensively studied [4–6]. Actually all presented techniques use a distributed memory environment as the target platform. Yet, the communication density they require cannot be fulfilled by today's message-passing machines. Hence, we opted for a parallel computer with shared memory.

We decided to follow a strategy similar to parallel discrete event simulation (PDES), see e.g. [7]. As the PRAM processors are assumed to run synchronously on the assembler instruction level, each memory access can be tagged with a time-stamp, i.e. the instruction count since system start, so that the models match well. Optimistic simulation techniques are successfully used in network and combat simulation [7]. Chandrasekaran et al. [8] report

an attempt to use optimistic methods for the Wisconsin Wind Tunnel (WWT) [9], using ideas similar to ours. WWT and its successor WWT II [10] use conservative discrete-event, direct-execution techniques to simulate parallel architectures of all kinds on existing parallel host machines. The result of their attempt was negative in terms of speedup, but the WWT has a focus slightly different from our work. It involves not only the processor, but also caches, memories, and communication networks. Thus, our approach may well succeed where theirs failed. In particular, a master thesis [11] produced a simple read–optimistic parallel implementation of the simulator, with some success for a special class of PRAM programs.

We will develop different PDES implementations of the SB-PRAM simulator and report on the obstacles along the way to speedups greater than 1. We will report on a prototype implementation and its performance. Finally we will give an outlook how small changes in the instruction set architecture might allow much larger speedups.

The remainder of this article is organized as follows. In Section 2 we provide all technical information necessary to attack the problem at hand, and present a first parallel PRAM simulator. In Section 3 we benchmark this simulator and several other variants, explaining this way the obstacles we met. We also report the performance results of a prototype implementation. Section 4 concludes and gives an outlook on further developments.

2 PRAM and PRAM Simulation

2.1 Parallel Random Access Machine

The *parallel random access machine* (PRAM) is a model for parallel computation. It is widely accepted and used as the most common model to analyze parallel algorithms without consideration of particular details of the actual machine on which the algorithm will run on. The PRAM model is a generalization from the *random access machine* (RAM) model of sequential computation. In a PRAM, an arbitrary number of RAM processors work on a shared memory, all with the same clock source but each with its own PC and a unique identifier (ID). The memory is accessible from all processors in unit time. We even allow multiple processors to either read or write a single memory address at the same clock event, i.e. we focus on the concurrent read concurrent write (CRCW) PRAM. However, we make no specification what happens if a read and a write happen simultaneously to the same address.

Writing to the same memory address with more than one processor needs a conflict resolution protocol to specify which value is written to the memory address. The most common protocols in increasing order of strength are:

Weak All processors must contribute the same special value (for example 0).
Common All processors must contribute the same value.
Arbitrary An arbitrary processor succeeds and writes its value to the memory. All other values will be ignored.
Priority The processor with the highest priority succeeds and writes its value to the memory. All other values will be ignored.
Combining All values from the participating processors will be combined by a commutative and associative function, like addition, bit wise `and`, or the maximum.

The PRAM model is a general purpose model for parallel computation. It gives the ability to design and analyze parallel algorithms It is still not feasible to build a computer which directly implements the PRAM model. Only for very small n, there exist memory modules that can fulfill requests from n processors.

A PRAM can be simulated on a shared memory machine by applying Brent's theorem, but this simulation would be quite inefficient.

The SB–PRAM is a distributed shared memory machine with uniform access time to the global memory for all processors and all processors have the same clock source. That is realized with a bi–directional butterfly interconnect between processor and memory modules. Each processor module contains a copy of the program memory and a local memory, which is usable only as I/O buffer. To hide the network latency each (physical) processor schedules 32 virtual processors in a round-robin manner. The context switch is implicit after one instruction and is completely implemented in hardware. As this hardware multithreading cannot fully hide network latency, memory loads are also delayed by one instruction. The result from a global memory load is not available in the next instruction but in the next but one. The processor is a Berkeley RISC architecture where all data paths are 32 bits wide and the memory is only word addressable. The global memory addresses are hashed over the memory modules with a universal hash function to reduce contention within the memory modules. Furthermore the global memory is logically split into shared and private memory regions.

The PRAM language Fork allows to program the SB-PRAM in a high-level language [1, Ch. 5]. Fork is an extension of the C programming language. Fork programs are executed in *single program, multiple data* (SPMD) style: All processors execute the same program, but may take different control paths through it (MIMD); the number of processors executing the program is fixed at the program start and can be accessed inside the program by a symbol. At run time, processors are organized in groups. A Fork program is statically partitioned into two different kinds of regions: In synchronous regions, the compiler guarantees that all processors of a group execute the same instruction at the same time. In asynchronous regions, no such guarantee is given. Fork maintains a group hierarchy, beginning with the group of all started processors. Synchronous execution in synchronous regions can be relaxed to processor subsets by splitting groups dynamically into subgroups. Subgroups are created implicitly at private branches[4] or explicitly with the `fork` construct. At any time during execution, this group hierarchy forms a tree, and Fork guarantees the synchronicity of processors in each leaf group that executes a synchronous region.

2.2 Parallel Discrete Event Simulation

In the previous section the term *simulation* was used to simulate one machine with another machine. In this section it is used to simulate a closed system over time on a computer. These systems react and change on events at discrete time values. These types of simulations are called *discrete event simulations* (DES), and parallel DES (PDES) if executed on a parallel computer.

The closed system to be simulated is represented by a set of state variables, a global clock that indicates how far the simulation has progressed, and a list of unprocessed events. Each event has a time stamp which indicates when this event must be processed in global time. The simulation kernel repeatedly takes the event with the lowest time stamp from the list of unprocessed events, sets the global clock to this event's time stamp, and processes this event. The event can change a state variable and can generate zero or more events with a time stamp greater than the current clock. The constraint that the simulation always takes the event with the lowest time stamp provides a sufficient condition to guarantee the simulation's correctness. Consider two events E_1 and E_2, where E_1 has a lower time stamp than E_2. If E_2 is processed before E_1 and reads variables that will be changed by E_1, or E_2

[4] A private branch is a branch where the condition depends on private data from the processor, like the processor ID.

alters a state variable that E_1 will read, than this may cause errors in the simulation model. These kinds of errors are called *causality errors*.

Yet, the above condition is not necessary, as it is possible that in a discrete event simulation events occur that do not influence each other, so that a reversed order of processing does not cause a causality error. If the simulation kernel can detect such *independent* events, the kernel can process these events in parallel without violating the correctness of the simulation. Such simulation kernels are called *conservative parallel discrete event simulations*, see e.g. [7] for a survey. Another class of parallel simulations, called *optimistic*, do not avoid causality errors, but detect and correct such errors, e.g. by rolling back to a system state before the occurrence of the error. They gain performance advantages by being able to extract more parallelism, if roll back occurs seldom. The most commonly known optimistic PDES is the *Time Warp* protocol.

The Time Warp protocol is based on the concept of *Virtual Time*[5] [12]. In the time warp protocol, the system state is represented by a number of *logical processes* (LP), each consisting of a local clock, a state variable, a list of saved states, an input queue and an output queue. The local clock indicates how far this LP has advanced in simulation time. Each event is transported via a message from the LP generating the event to the LP whose state variable will be changed by the event. In addition to the time stamp when the event must be processed, the message contains a time stamp with the local time of the LP sending the message. Each LP behaves like a sequential DES kernel: it repeatedly takes the unprocessed event with smallest time stamp from its input queue, sets the local clock to this time, and processes the event.

Whenever an event arrives at an LP with a time stamp smaller than the LP's local clock, a causality error has happened. Such an event is called a *straggler*. To recover from the causality error all actions from processed events with a time stamp greater than the straggler message are undone, and the local clock is set back to the time stamp of the straggler. To be able to restore the state variable of the LP to the value it had before the straggler's time stamp, the state is saved periodically. The events undone may have generated other events, which have already been sent to other LPs. To undo these events, Jefferson introduced the concept of the *anti–message*. Whenever an LP sends a message the kernel generates a second message (the anti–message) which is stored in an output queue of the sending LP in send time order. These two messages are identical except that the anti-message is tagged with a negative sign. In a rollback situation the process sends all anti–messages from the output queue which have a send time stamp greater than the straggler message.

If an LP receives an anti–message, there are three possible situations:

1. If the positive message corresponding to the anti–message has not yet been processed, both messages will be deleted. This suffices as the state of the destination LP has not been altered yet by the positive message to be undone.
2. If the positive message has not yet arrived in the destination process, then the anti–message is stored in the input queue. When the positive message will arrive, both messages will be deleted.
3. If the positive message has already been processed, then the destination LP must roll back to a local time prior to the processing time of the positive message, and both messages are deleted.

Thus, messages and anti–messages are always created and destroyed in pairs.

The global control uses the *global virtual time* (GVT) as a lower bound on how far the simulation has progressed. The GVT is the minimum of all local clocks and all time stamps of messages in transit. As no LP can roll back to a time prior to the GVT, saved states

[5] Virtual Time is defined on a real time space but can trivially be used in a discrete time space.

and messages in input and output queues with time stamps lower than the GVT can be removed[6]. This is called *fossil collection* (FC). The GVT also helps in the termination of the simulation, yet we will not go into details here.

2.3 Sequential SB–PRAM simulation

The programming environment of Fork includes a sequential simulator program of the SB–PRAM called PRAMsim, to enable program development and test without access to the SB–PRAM itself. The PRAMsim focuses only on simulating the SB–PRAM on instruction set level and thus does not simulate the network, memory modules, and hashing. The simulator takes as input an executable for the SB–PRAM. The executable contains the program text, initialized global and local memory, relocation data, and the symbol table. The simulator calls the *pram loader* from the SB–PRAM tool chain, which relocates the binary and produces the initial program, global, and local memory images. After initializing the physical and virtual processors, the simulation loop is started. This simulation loop is illustrated in the pseudo code depicted in Algorithm 2.1.

As we do not simulate the network, and do not access the special purpose registers of physical processors, we do not consider the partitioning of virtual processors on physical processors, i.e. we assume identity of physical and virtual processors.

Algorithm 2.1 PRAMsim simulation loop

```
1: while not end of simulation do
2:     for vp ∈ VP do
3:         execute one instruction of virtual processor vp
4:     end for
5: end while
```

The ordered set VP contains all virtual processors in increasing order of their processor IDs. Thus the while loop processes one instruction of all virtual processors in a round-robin manner, and obeys the priority for concurrent accesses and operations such as multi-prefix. Consider the following example: Let $Q_s \subseteq VP$ be an ordered subset with $n = |Q_s|$ processors q_i, each performing in step t the multi–prefix operation `mpadd` with operand v_j on memory address s. Let v_s denote the value in the memory at address s prior to time step t. Processor q_i gets the return value

$$r_i = v_s + \sum_{0 \le j < i} v_j, 0 \le i < n$$

and the value stored in s afterwards will be

$$v_s^* = v_s + \sum_{0 \le j < n} v_j$$

Note that for multi-prefix and load operations on the global memory, the return value from the memory is not directly written to the target register of the processor, but delayed for one instruction as in the SB–PRAM machine itself. This is done by keeping it in a temporary variable together with the target register index.

This simulation scheme is obviously very simple and thus very efficient. Therefore this algorithm should be recognized as the fastest known sequential solution for simulating the SB–PRAM.

Each physical processor has a local memory attached, which only can be accessed from the associated virtual processors. Further this memory is only visible to the operating system

[6] However, at least one saved state must remain to enable roll back.

and cannot be used for user code. In contrast to the global memory, loads from the local memory are not delayed. Hence, a global memory load directly followed by a local memory load must be handled specially by the simulator.

System input and output (I/O) is provided through the system call interface which is accessible by the assembler instruction `sysc`. This system call interface is simulated natively on the host system of the simulator. These simulated routines (like open, close, read, write, ...) access the global memory directly without the delayed load and without increasing the instruction counter of the processor.

Beside the simulation loop the PRAMsim provides commands to inspect and modify all registers and memories of the simulated SB–PRAM. Also debugging is supported via breakpoints and interrupts by the user.

2.4 Parallel SB–PRAM Simulation

In order to accelerate the simulation of an SB–PRAM program, we model the simulation as a parallel DES. As the virtual time, we use the number of executed instructions as it is discrete. Each SB–PRAM processor is modeled as an LP, the state being its register set. The instruction counter is used as the local time clock. For the rollback mechanism the whole register set is copied and stored in a circular linked list. The list provides easy and efficient rollback and fossil collection. The global memory might be modeled as a single LP, but we use multiple LPs for performance reasons, as explained below.

Events are accesses to the global memory, i.e. each processor executes instructions until it encounters a global memory access. In case of a store instruction, the processor sends the store event to the global memory, and continues. In case of a load instruction, the processor sends the load request event to the global memory, and waits for the reply. This seems to contradict the time warp philosophy, but it is clear that the processor cannot advance without the reply, and furthermore, it is also clear that the processor will not receive any other events, that it may process.

The global memory could be modeled on several levels of granularity. To sketch both extremes, either each global memory cell could be one LP, or the global memory as a whole could be one LP. The former solution has the advantage that a roll back in a cell-LP will only affect processor LPs that have accessed this cell. Its disadvantage is the huge number (billions) of LPs to be simulated. In the latter extreme advantage and disadvantage are interchanged: there is only one object, but a roll back caused by one memory cell would have to affect all memory accesses. Furthermore, it is to be expected that this LP would serialize the simulation. As a compromise, we model each global memory page as an LP.

Events processed by a page consist of the different memory access operations possible: load, store, multi-prefix, sync. While each memory page LP follows the time warp protocol, the state saving must be handled differently, in order to avoid storing complete pages. We adopt the *reverse computation* approach [13]. When events perform functions that have an inverse, then instead of storing the state one may store events, and undoes them, i.e. perform the inverse functions, in reverse order of storing. Strictly speaking, store instructions do not have an inverse, but we store the old value of the cell (known as before image in database systems) with the event in this case.

When modeling the memory in this way, it becomes clear that system I/O, which is executed natively on the host machine, would either require to stop the simulation, or would lead to complex integration with high overhead. Therefore, we do not consider it at this time.

Note also that delayed load now can be modeled accurately. Remember that the result of a memory request in time step i is available in the register of the processor at time step

$i+2$. A processor sends a message in time step i to the memory, consequently the message has a send time of i and a receive time of $i+1$. The memory module processes this message at time step $i+1$ and will return a reply message with send time $i+1$ and receive time $i+2$, if requested. So the reply message is available in the processor at time step $i+2$ and can be loaded into the destination register before the instruction at time step $i+2$ will be executed.

3 Prototypes and Experiments

3.1 Overhead Determination

We have implemented a sequential time warp variant of PRAMsim and compared it with the original PRAMsim, to find out the amount of overhead introduced by the time warp protocol. The measurements were taken on a computer with 4 Opteron processors running at 2.4 GHz, and 4 GByte of main memory. Only one processor was used to execute the simulation, the other processors carried the remaining system load. As a benchmark, we use the startup code of the Fork runtime system, by running a Fork program where the `main` routine directly returns. For this benchmark, about 8600 instructions are executed by each PRAM processor. As the processor LPs are simply scheduled round-robin in the sequential time warp PRAMsim, and all execute identical code, causality errors do not occur. Thus the comparison of the runtimes shows the overheads quite clearly. Figure 1 shows the runtimes for different numbers of PRAM processors simulated, and reveals a slowdown of 13.68 for $p = 2^{16}$.

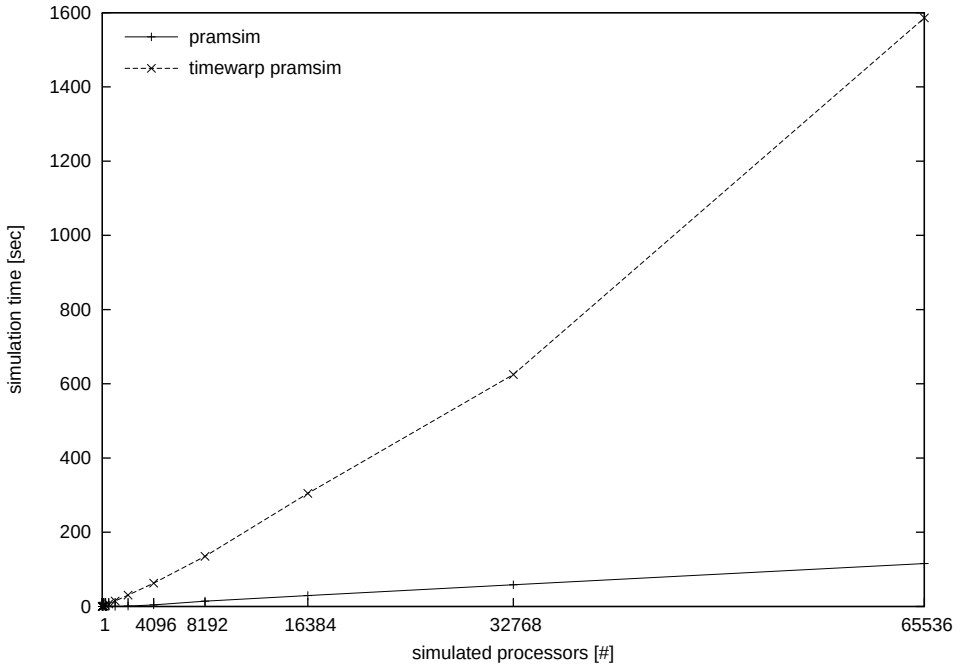

Fig. 1. Comparing Time Warp PRAMsim and original PRAMsim

This result can be explained by the fact that the simulation of one PRAM instruction on average needs 32 Opteron instructions only. Thus queue management, fossil collection and other parts of the time warp protocol greatly increase this number, considering that more than 20% of the instructions access the global memory. Yet it means that at least 14 Opteron processors would be necessary to break even on the runtime, even under the very optimistic

assumption that a parallel execution does not introduce additional overheads such as roll backs.

This initial failure lead to two consequences. First, given the fine granularity of the communication between the LPs, a message-passing implementation on a cluster computer seems out of reach, even when a low-latency network such as myrinet would be used. Thus, in the sequel we concentrate on a parallelization on a shared memory parallel computer. Second, we have to reduce the overhead of the pure time warp protocol by incorporating other simulation techniques.

3.2 Shared Memory Parallelization

We start with a straightforward parallelization of the sequential PRAMsim. The for-loop of Algorithm 2.1 is distributed over a number of *threads*, the PRAM global memory is allocated in the shared memory of the host computer. In order to avoid races during concurrent execution of multi-prefix commands, which first read a cell and then write it again, the execution of all memory accesses in one step is postponed until all PRAM processors have finished that step. This is detected by a barrier command, then one of the threads performs all memory accesses sequentially, and finally another barrier guarantees that none of the threads starts simulation of the next step before the memory access phase is finished. The result is displayed as Algorithm 3.1, where VP_{id} is the part of the PRAM processors to be simulated by thread id.

Algorithm 3.1 Threaded PRAMsim simulation loop

```
1:  while not end of simulation do
2:      for vp ∈ VP_id do
3:          execute one instruction of virtual processor vp without memory access
4:      end for
5:      barrier
6:      if id = 0 then
7:          for vp ∈ VP do
8:              perform the memory request of virtual processor vp, if any
9:          end for
10:     end if
11:     barrier
12: end while
```

This scheme is obviously very simple and does not introduce any new data structures. Yet it introduces overhead for the synchronization inside the barrier. Figure 2 depicts the runtimes for 1, 2, and 4 threads, where 1 thread means the sequential PRAMsim. The synchronization cost is expected to be about the same independently of the number of PRAM processors simulated, and the number of threads used. This is clearly reflected by the 2– and 4-threaded versions having almost identical runtimes, and the curves having almost similar slopes, i.e. constant offset, for larger numbers of PRAM processors simulated.

To reduce this overhead, we combine this scheme with ideas from the time warp PRAMsim: like an LP, each processor is simulated for several instructions until it reaches a global memory read. The simulation of this processor is only continued if a reply from the global memory LP arrives. Yet in order to avoid roll backs (and thus the overhead for the associated data structures), the memory LP only processes those read requests with time stamps less than or equal to the GVT, where the GVT in this simulation scenario is the minimum time stamp of the requests not yet processed by the memory. We call this simulation scheme *C–PRAMsim*, it is depicted in Algorithm 3.2. The only overhead which is left from the Time Warp PRAMsim is a sorted global input queue at the memory, and the generation of requests.

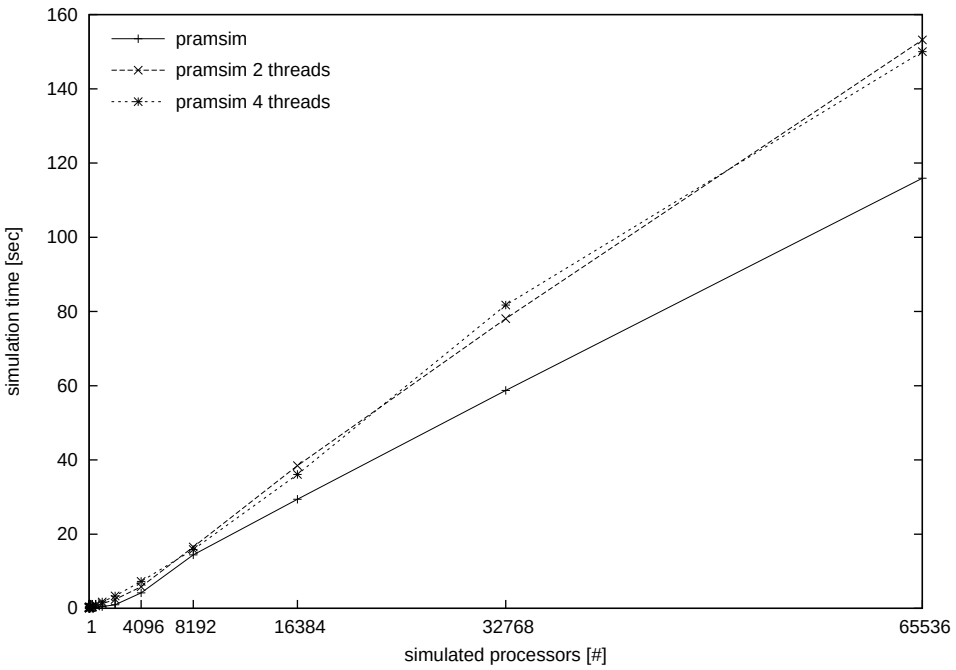

Fig. 2. Simple parallel PRAMsim

Algorithm 3.2 C–PRAMsim simulation loop

```
1:  while not end of simulation do
2:      for vp ∈ VP_id do
3:          execute instructions of virtual processor vp before first load request
4:          issue load request
5:      end for
6:      barrier
7:      if id = 0 then
8:          determine GVT
9:          process all requests less or equal to the GVT
10:     end if
11:     barrier
12: end while
```

We assess C–PRAMsim by comparing it with the Time Warp PRAMsim and the sequential PRAMsim. The Time Warp PRAMsim is parallelized in a simple manner by distributing all LPs onto 1, 2 and 4 threads, and simulating them in a round-robin manner. Figure 3 depicts the runtime results on the startup code. While the C–PRAMsim on 2 threads performs much better than the Time Warp PRAMsim, it is still slower than the parallel version from Algorithm 3.1 and the version with 4 threads is slower than the one with 2 threads.

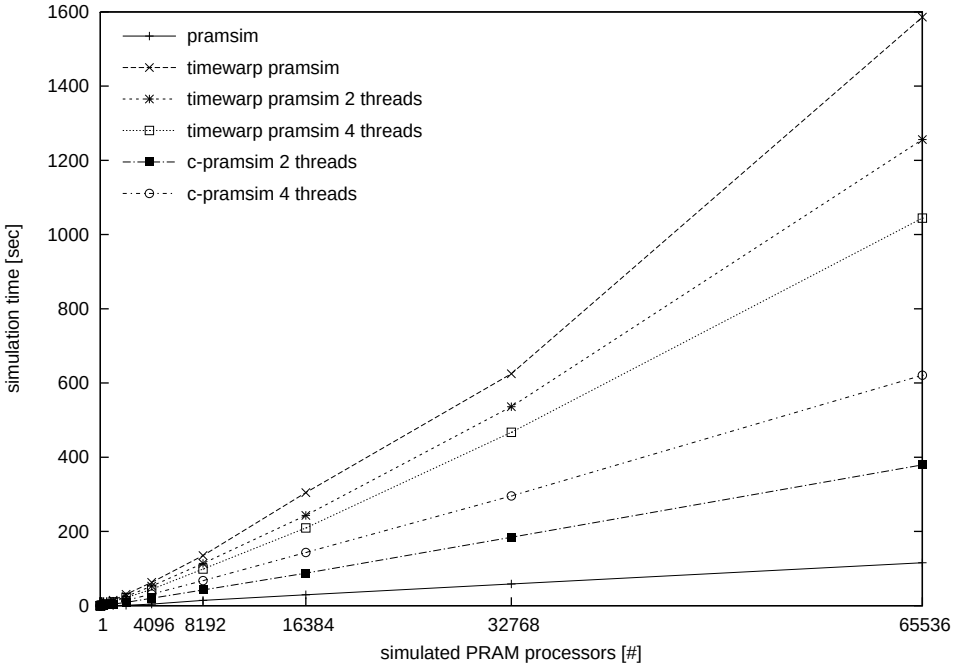

Fig. 3. Assessment of C–PRAMsim

In order to further test the idea of C–PRAMsim, we choose a different benchmark, to be seen in Algorithm 3.3. The s-loop is programmed in assembler, and thus does not access the global memory at all. The i-loop accesses global memory for each iteration, as the Fork compiler places variables in the global memory. Also the Fork compiler places a barrier after the s-loop. Thus, by varying the value of s, the density of memory reads can be controlled: at least $2s$ instructions[7] without memory reads occur between accesses to i and the barrier. In order to see the effect of this benchmark, the simulation time is reduced by the time to simulate the startup code, denoted as normalized in the figures.

Algorithm 3.3 Synthetic benchmark routine to reduce memory accesses

```
1: for i ∈ [1, n] do
2:     for j ∈ [1, s] do
3:         {do nothing}
4:     end for
5:     {implicit barrier from Fork compiler}
6: end for
```

Figure 4 depicts the runtime results for varying values of s, for $n = 128$ and $p = 8192$. We see that for $s \geq 48$, the C–PRAMsim on 2 threads gets faster than the sequential PRAMsim. However, $2 \cdot 48 = 96$ instructions without memory access is unlikely in an

[7] One arithmetic instruction and one conditional jump per iteration.

application program. Also the 4 thread variant is still slower than the 2 thread variant, mainly because the memory requests are processed sequentially.

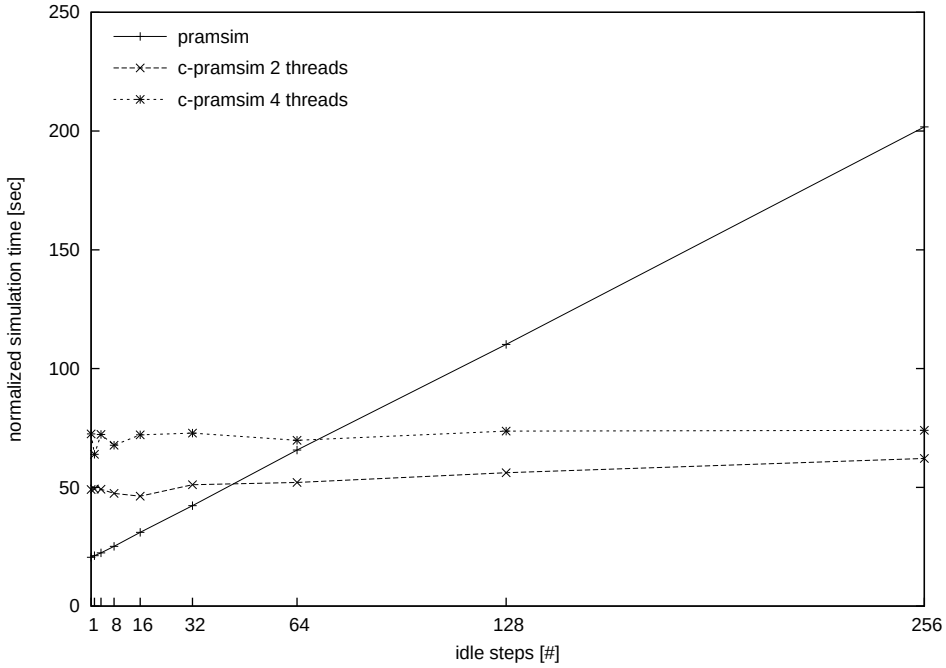

Fig. 4. C–PRAMsim with synthetic benchmark

Thus, we finally parallelize the memory access phase. We partition the memory requests according to their target address ranges, in order to be able to employ multiple threads in parallel. Also we have to find a way to compute the GVT on all threads without introducing new overhead.

Let T be the number of threads, B a two dimensional array of buckets with size $T \times T$, and M an array of size T. In the processor phase each thread $t \in [1, T]$ uses only the buckets in row t of B. A request is put into bucket $B(t, i)$ if the request will be performed by thread i in the memory phase. The thread t also maintains the minimum read request time stamp in $M(t)$. In the memory phase thread t accesses only the column t of B and sorts all new requests from this column into its memory input queue. All threads read the array M to calculate the new simulation horizon. This scheme guarantees that the simulation horizon is correctly maintained and no additional synchronization is needed.

The result of this PRAMsim implementation (called C^2–PRAMsim) is presented in Figure 5. We see that the C^2–PRAMsim performs better than the C–PRAMsim, and that the 4 thread variant is faster than the 2 thread variant.

Lastly, we replace the assembler coded s–loop of Algorithm 3.3 by Fork code. This will bring back dense memory accesses, because of the private loop variable j. The runtime results are depicted in Figure 6, where the C^2–PRAMsim performs better than the C–PRAMsim, and is faster for 4 threads.

4 Conclusion and Future Work

The idea to use optimistic simulation techniques for parallelizing the PRAM simulator was very promising. The Time Warp mechanism can explore almost any parallelism that the simulated model exposes, and the PRAM simulator possesses a high degree of parallelism. Yet, it turned out that the PRAM simulator is very sensitive to added overhead because

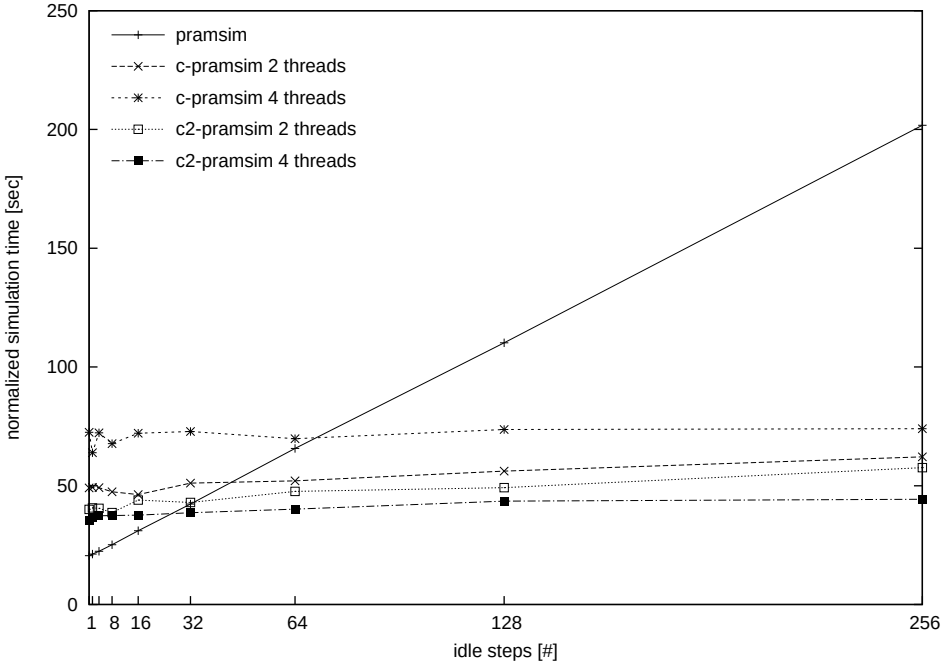

Fig. 5. C^2–PRAMsim with synthetic benchmark

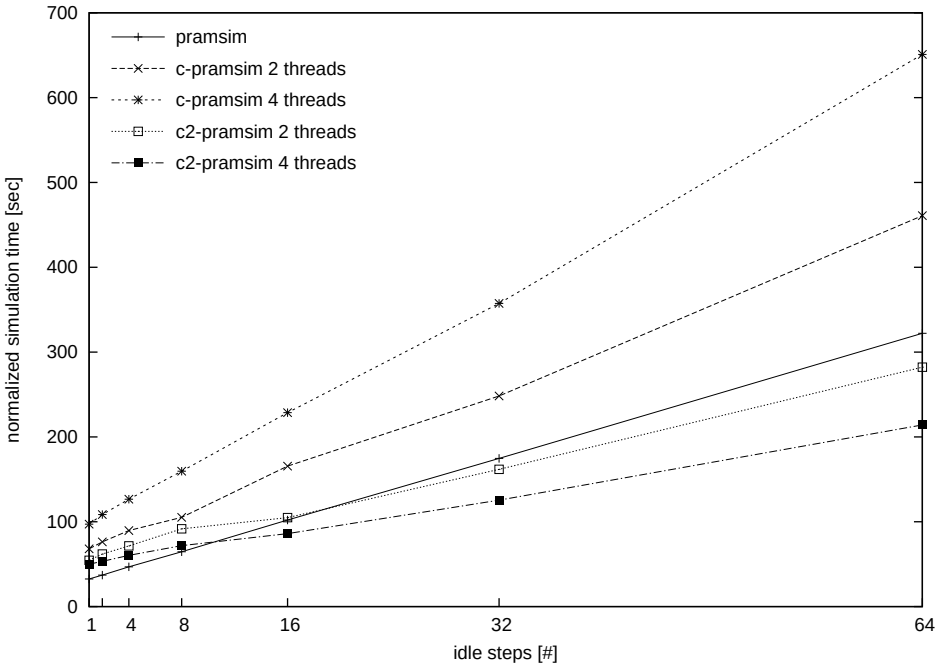

Fig. 6. C^2–PRAMsim with synthetic benchmark and dense accesses

of the very fine granularity of the simulation. This also prevents a simple parallelization with shared memory programming techniques. In the end, a combination of ideas from Time Warp PDES and shared memory programming proved successful to gain a moderate speedup.

The focus of this work was on changing the simulator while leaving the instruction architecture unaltered. If the focus is shifted to the complete tool chain, more precisely the Fork language and compiler, it seems possible to use language characteristics to work out new ideas. Distinguishing accesses to shared and private regions reduces the density of accesses to be synchronized. The group hierarchy of a Fork program is another potential source for improvement. If the simulator can access the group information it only needs to synchronize accesses within a group. Also, if the mapping of PRAM processors onto cluster processors reflects the group hierarchy, this can be profitably used to simplify synchronization. Pushed to the extreme, all processors of a group may be simulated sequentially on one cluster processor, eliminating synchronization alltogether. As speedup now depends on the group structure of the simulated program, this is only a partial solution. Finally, if synchronizations of groups are not handled in software but by the simulator, accesses to shared memory can be further reduced.

There also is another idea to use group information. Within a group, synchronization may happen with write requests, as the PRAM processors within that group reach the same write. Subsequent shared reads can then be performed immediately without further synchronization. As writes typically are less frequent than reads, this would further reduce the density of synchronizations.

The changes required in the tool chain to expose the group structure are moderate, so that the insights gained here will lead to much higher speedups at a fair price.

References

1. Keller, J., Kessler, C., Traeff, J.L.: Practical PRAM Programming. John Wiley & Sons, Inc., New York, NY, USA (2000)
2. Kessler, C.W.: A practical access to the theory of parallel algorithms. In: Proc. of ACM SIGCSE '04 Symposium on Computer Science Education. (2004)
3. Huseynov, J.: (Distributed shared memory home pages) `http://www.ics.uci.edu/~javid/dsm.html`.
4. Harris, T.J.: A survey of PRAM simulation techniques. ACM Comput. Surv. **26**(2) (1994) 187–206
5. Karp, R.M., Luby, M., auf der Heide, F.M.: Efficient PRAM simulation on a distributed memory machine. In: STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing, New York, NY, USA, ACM Press (1992) 318–326
6. Pietracaprina, A., Pucci, G.: The complexity of deterministic PRAM simulation on distributed memory machines. Theory of Computing Systems **30**(3) (1997) 231–247
7. Fujimoto, R.M.: Parallel discrete event simulation. Commun. ACM **33**(10) (1990) 30–53
8. Chandrasekaran, S., Hill, M.D.: Optimistic simulation of parallel architectures using program executables. In: Workshop on Parallel and Distributed Simulation. (1996) 143–150
9. Reinhardt, S.K., Hill, M.D., Larus, J.R., Lebeck, A.R., Lewis, J.C., Wood, D.A.: The wisconsin wind tunnel: virtual prototyping of parallel computers. In: SIGMETRICS '93: Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems, New York, NY, USA, ACM Press (1993) 48–60
10. Mukherjee, S.: Wisconsin wind tunnel II: A fast and portable parallel architecture simulator. In: Workshop on Performance Analysis and Its Impact on Design (PAID). (1997)
11. Clauß, C.: Paralleler PRAM–Simulator. Master thesis, Computer Science Department, FernUniversität Hagen, Germany (2007)
12. Jefferson, D.R.: Virtual time. ACM Trans. Program. Lang. Syst. **7**(3) (1985) 404–425
13. Carothers, C.D., Perumalla, K.S., Fujimoto, R.: Efficient optimistic parallel simulations using reverse computation. In: Workshop on Parallel and Distributed Simulation. (1999) 126–135