W. Karl, J. Keller (Eds.): PARS 2017
Proc. 27th PARS-Workshop

Efficient Simulation of PRAM Algorithms on Shared Memory Machines[1]

Nicolas Berr[2]

Abstract: The parallel random-access machine (PRAM) is an abstract shared memory register machine used in computer science to model the algorithmic performance of parallel algorithms. Although being used as theoretical model for many years, only few attempts have been made to prove technical feasibility of the model for the use in real world applications. One of these attempts was the SB-PRAM Project, which included the development of a real PRAM hardware, a high-level PRAM programming language and a compiler. It offered programmers the ability to implement algorithms designed for a PRAM in a natural way. Today, the hardware based prototype no longer exists, but a simulation software is still available. Even though the simulated hardware contains a huge amount of inherent parallelism, it turned out to be hard to provide an efficient parallel implementation of the simulation. In this article a promising new approach for this problem, its implementation and evaluation is presented. Experiments have shown the high potential of its efficiency and discover even more potential that can be exploited by future work.

Keywords: PRAM, Simulator, parallel, Multicore-Architecture

1 Introduction

The parallel random-access machine (PRAM) is an abstract register machine used in computer science to model the algorithmic performance of parallel algorithms. Although being used as theoretical model for many years, only few attempts have been made to upraise the model to practical usability. One of these attempts was the SB-PRAM Project. It was intended to be a "proof of concept" that the theoretical model of the PRAM is technically feasible for use in real world applications. The project included the development of a real PRAM hardware, the high-level PRAM programming language Fork, and a Fork compiler [KKT01]. In addition, a simulation of the hardware has been provided in the form of a software called `pramsim`.

The simulation of PRAM algorithms can provide a practical access to the theory of parallel algorithms. Even more, scientific developers of parallel algorithms are able to exemplaryly verify if the result and the assumptions about runtime complexity are correct. However, the PRAM simulation is running very slow compared to programs natively running on present desktop computer systems. Therefore, a significant runtime improvement is required in order to be able to observe more than toy size problems. The inherent parallelism of the simulation suggests parallel computation as a solution. However, there were several

[1] based on the master thesis: "New Parallelization Approaches for PRAM Simulations"

[2] ACS / EON ERC RWTH Aachen University, Mathieustraße 10, 52074 Aachen, nberr@eonerc.rwth-aachen.de

previous attempts to parallelize the simulator with different approaches that did not achieve any significant performance improvement, or were restricted to certain classes of PRAM programs.

This article describes the basic approach followed in order to achieve significantly more parallel performance. Details about the implementation and measured results of the parallel performance are given. It will conclude on the work done and will offer some ideas that potentially can lead to even more parallel performance.

2 Background and Related Work

The PRAM-Model The PRAM was introduced as generalization from the random-access machine (RAM) by Fortune and Wyllie in 1978 [FW78, KKT01, Vi10]. In theory, a PRAM employs an unbounded number of processors, synchronized by a common clock, all having unit time access to a common shared memory, and a private local memory in addition. If the model is used to discuss the performance of a concrete parallel algorithm, the actual number of processors used often is assumed to be set to a certain number p, typically calculated in relation to the size of the input n, e. g. $p := \sqrt{n}$.

At each time step a processor can write into the shared memory, read from shared memory, or perform some computation on its local memory. If a program is processed in parallel, each single instruction will be done in parallel by all PRAM processors synchronously. Thus, either all processors are reading from shared memory, all are writing to the shared memory or all are doing some computation based only on local (private) memory. While synchronous computing on processor local data does not cause any possible conflicts, accesses to the same memory cell do. There is a variety of rules to resolve such conflicts that are typically divided into three main variants: exclusive-read exclusive-write (EREW), concurrent-read exclusive-write (CREW) and concurrent-read concurrent-write (CRCW) [KKT01]. The former does not allow any concurrent access to a shared memory cell, the latter allows concurrent read as well as write accesses.

The CRCW variant requires additional rules for the conflict resolution, such as *Weak* (only a certain predefined value is allowed for simultaneous writing), *Common* (all processors writing to the same location have to write the same value), *Arbitrary* (only one arbitrary processor's value will be written to the memory), *Priority* (the value is determined by a predefined processor priority) and *Combining* (a reduction function, such as minimum or maximum, is applied to all written values) [KKT01]. It has been shown that an implementation of a PRAM supporting at least the Priority and the Combined CRCW-PRAM variants will also be able to generate valid results when running algorithms designed for most of the other variants [KKT01].

The SB-PRAM Project The SB-PRAM Project was intended to be a "proof of concept" that the theoretical model of the PRAM is technically feasible for use in real world applications [FGK97]. It included everything needed to program and run such an

application: a programming language, a compiler, a hardware and an operating system. The SB-PRAM hardware is a scalable shared memory architecture with uniform access time. It basically emulates the Priority CRCW-PRAM model. Additionally, powerful multi-prefix operations, like multi-prefix addition (*mpadd*), have been implemented, which practically empowers the SB-PRAM to support nearly all relevant models. To take full advantage of the PRAM-like programming model supported by the hardware on the low level of assembly, the C based language Fork has been developed to enable the programming on a higher level. In order to enable software developers programming the SB-PRAM without having access to the real hardware, a simulation of the hardware has been provided in the form of a software called `pramsim`. It is available for free[3] and is part of a collection of compiler and system tools for the SB-PRAM. Besides the usage for research purposes, `pramsim` has enabled the practical experience of PRAM programming also as a complementation of classical theory courses on PRAM algorithms [Ke04].

Fork Fork is a high-level programming language for PRAMs [KKT01]. It was developed as part of the SB-PRAM project introduced in Section 2. Like other shared memory based parallel programming languages, Fork offers explicitly assignable *shared* (`sh`) and *private* (`pr`) variables with comparable restrictions, except for the additional concept of a group level shared memory. Fork offers two different program execution modes that are associated with source code regions: the *asynchronous* and the *synchronous* mode.

In *synchronous execution mode* processors remain synchronous on the statement level (which is passed through the compiler's output on instruction level), i. e., the instruction counters of the processors are equal at each machine cycle. This is called the *synchronicity invariant*. If a conditional jump occurs that evaluates differently for parts of the participating processors, they are split into two groups. The *synchronicity invariant* remains guaranteed for the processors belonging to the same group. This also implies,that all expressions consisting only of shared objects always evaluate to the same value for all processors of the group. If groups are joined again, their processors have to be synchronized in order to maintain the synchronicity invariant.

In *asynchronous execution mode*, the synchronicity invariant is not enforced. The instruction pointers of the processors may differ at any time and there are no implicit synchronization points, hence the use of explicit synchronization is required to guarantee that accesses to shared objects occur at the points in time intended by the algorithm. Programming *asynchronous* regions is very similar to the programming model used in today's popular and well known thread based shared memory programming extensions like OpenMP. An exemplary use of the Fork language for both, synchronous and asynchronous algorithms, is illustrated in Appendix A in Section 6.

Previous Work on the Parallelization of the SB-PRAM Simulator Although the SB-PRAM simulator `pramsim` is simulating a massively parallel computer architecture, it is coded as a pure sequential program. It appears likely that a parallel version of

[3] `http://www.ida.liu.se/~chrke/fork.html`

the simulator would be easy to develop by exploiting the inherent parallelism of the simulation. However, there were several previous attempts to parallelize the simulator with different approaches that did not achieve any significant performance improvement, or were restricted to certain classes of PRAM programs.

There are several publications related to the topic of PRAM simulation that can be classified into three categories. The first category includes approaches based on the assembly model of the SB-PRAM processor directly without any knowledge about any semantics from a higher level [Cl07, CLB12, Bl07]. All these approaches have in common that all read and write operations in parallel simulations must lead to the same result as in sequential simulation. In order to achieve this, they mainly followed the idea of read optimistic (continue unsynchronized operation until a checkpoint, rollback in case of conflicts) or conservative (block operation if a conflict may occur until all processes are blocked) approaches. They were applied to the original simulator code or even to a full parallel discrete event simulation (PDES) implementation. The attempts were based on Message-Passing implementations and were only successful for algorithms that tend to cause very few conflicts, i. e. make use of shared memory write accesses very sparingly.

The second category of approaches is based not only on the semantics of the SB-PRAM assembly model, but rather includes additional knowledge about the higher programming language level. One of this kind is introduced in [KKW09]. The idea is based on the Fork semantic of synchronous groups. As already stated in Section 2, the *synchronicity invariant* is only guaranteed for processors belonging to the same group. Thus processors that split into different groups can be simulated by different threads without any synchronization until they reunite again. In order to avoid overhead in detecting group splitting from within the simulator, the Fork compiler was extended to add hints for the simulator in the form of a new (pseudo) instruction `shint`. Even though the implementation was based on shared memory, there was no significant performance gain in using multiple threads. However, besides the mechanism of *shints*, this work also provided a slim version of the sequential `pramsim` that introduced some other simplifications to the code, representing a good basis for further work on the parallelization of the simulation, based on the combination of assembly model and fork semantics.

The third category of approaches completely gets away from the idea of simulating a PRAM and is based only on the semantics of Fork. An example of this kind is the source-to-source compilation approach presented in [BKK12] which was very successful. It relies completely on creating general-purpose computing on graphics processor units (GPGPU) compatible code from synchronous fork code regions but does not maintain the capability to simulate and analyze the runtime behavior of PRAM algorithms. In addition there is no direct support for the Priority CRCW collision resolution or multi-prefix operations.

With respect to the previous work done and to the value of being able to analyze the runtime behavior of a huge class of PRAM algorithms it was decided to follow new ideas based on the second approach.

3 Concept and Implementation

The Simple and the Extended Approach The simple approach is mentioned here to provide a basis for extensions to the approach and to illustrate the main problem of fine-grained parallelism. It is one of the first category of approaches introduced in Section 2, since it is based only on the assembly model of the SB-PRAM.

The concept is realized by SPMD fashioned shared memory programming: the simulation loop presented in Figure 1 is run by several threads having a unique *id*. The complete set of physical PRAM processors (PPs) is divided as equally as possible into disjunct subsets PP_{id}. As soon as all PPs have processed one instruction, the memory processing phase is done by a single thread (for later reference this procedure will be called `SimulateMem`). Hence, the actual parallel processing is limited to the simulation of the instruction in lines 2 to 4. Furthermore, synchronization is needed in every simulation step to make sure that instruction- and memory-processing does not overlap.

```
 1: while not end of simulation, with thread 0..n-1 as id parallel do
 2:     for pp ∈ PP_id do
 3:         execute one instruction of physical processor pp
 4:     end for
 5:     barrier
 6:     if id = 0 then
 7:         for pp ∈ PP do
 8:             perform the memory request of physical processor pp, if any
 9:         end for
10:     end if
11:     barrier
12: end while
```

Fig. 1: Simple Parallel Pramsim Simulation Loop (based on [Bl07])

The basic idea of extending the simple approach is coarsening the synchronization granularity by executing several instead of just one instruction per simulation round. This principle is illustrated in Figure 2. The execution of the instruction stream by a single PP is continued until a certain instruction or condition is met (for later reference this procedure will be called `SimulateMultipleStepsPP`). The occurrence of such an event is called *synchronization point* since the resulting simulation will be synchronized only at these points. One of the approaches outlined in Section 2 – [Bl07] – already introduced this principle, but was based only on the semantics of the assembly model, i. e., any access to the global memory represented a synchronization point.

A new approach combines this principle with some knowledge about Fork semantics and has been inspired by the bulk-synchronous parallel model [Va90]. The basic idea is to think of the simulation as a sequence of segments being connected by global synchronization phases. Inside such segments, it is assumed that no relevant information is propagated between PPs, i. e., the simulation can be run in parallel without further constraints. Information that has to be accessible by other PPs is written to the global memory only during the global synchronization phases.

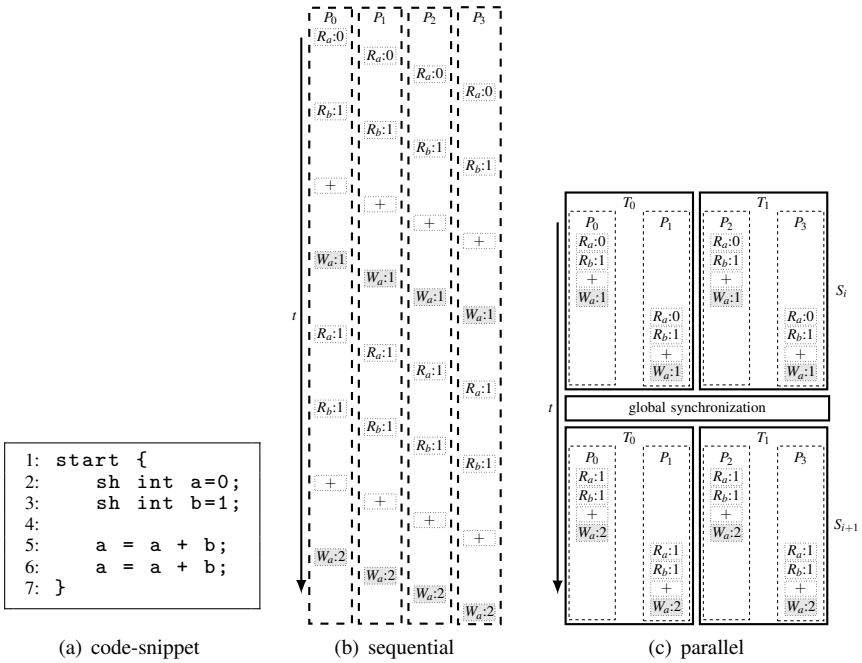
```
 1: while not end of simulation, with thread 0..n-1 as id parallel do
 2:     for pp ∈ PP_id do
 3:         execute instructions of pp until a synchronization point is reached
 4:     end for
 5:     barrier
 6:     if id = 0 then
 7:         for pp ∈ PP do
 8:             perform the memory request of pp according to certain constraints
 9:         end for
10:     end if
11:     barrier
12: end while
```

Fig. 2: Extended Simple Parallel Pramsim Simulation Loop

A simple variant of this approach is to consider any write access to the global memory as a relevant propagation of information. In this case, a segment – the PP simulation in lines 2 to 4 (Figure 2) – is run in parallel for each PP until its instruction stream reaches a global write instruction. The global synchronization phase in lines 5 to 11 actually performs all write operations requested in a strictly sequential order by a single thread while the other threads stay idle. Then the next segment is processed.

For a better understanding, this scenario is illustrated using the example code-snippet shown in Figure 3(a). The example shows a part of a simulation where all PPs are part of a single synchronous group, actually running the same code sequence. The result for a and b at the end of the code-snippet is expected to be the same as if only one processor was used. In the present case actually $a = 2$ and $b = 1$. Figure 3(b) shows this scenario for the sequential simulation of four PPs $P_0, ..., P_3$ and Figure 3(c) shows the parallel version of this simulation using two threads T_0 and T_1. The instructions are noted in a simplified format. There are read (R), write (W) and arithmetic ($+$) instructions. Read and write instructions are denoted by a subscript, the name of the variable, and followed by the value actually read or written. Just as expected, the result at the end of the simulation part is $a = 2$ and $b = 1$ for both, the sequential and the parallel simulation.

As mentioned before, the *global synchronization* divides the simulation into consequtive segments. This is visualized in Figure 3(c) for the two segments S_i and S_{i+1}. The write instructions within these segments are treated as write requests while read instructions are processed directly. The processing of write requests is delayed until the next global synchronization. Therefore, the read operations R_a for P_1 and P_3 in S_i result in the value of a as at the beginning of S_i. Moreover, any read operation for any PP results in the value of the accessed variable as at the beginning of the segment, hence is guaranteed to be the same for all PPs. For the special case of a single synchronous group this also guarantees the result of sequential and parallel simulation being exactly the same, since all PPs are executing the same sequence of instructions in all segments.



(a) code-snippet (b) sequential (c) parallel

Fig. 3: Simulation of a Single Synchronous Group

Coarsening the Synchronization Granularity The coarsening of the synchronization granularity is the main source of optimization potential. The objective is to maximize the number of instructions that can be simulated in the function `SimulateMultipleStepsPP` without breaking the rules for the correct processing of Fork programs. This can be done by the successive extension of the meaning of "relevant propagation of information". Therefore, the implementation includes a function that decides when the simulation of a single PP has to be stopped for the actual segment. The function can be configured to make decisions according to the following levels of granularity. The number of write accesses actually not considered being a relevant propagation of information (RPI) increases with the level.

Level 1 Any write operation to the global memory is considered as RPI.

Level 2 Extending Level 1, write requests to the private partition of the global memory are not considered being an RPI. Accesses to the private partition can be easily identified by the highest bit of the virtual address.

Level 3 Extending Level 2, accesses to the actual private stack-frame of the accessing PP are detected and not considered as RPI. It was investigated that the private stack-frames of the PPs are not located in the private partition of the global memory. Therefore, accesses to the private stack-frame have to be detected by comparing the virtual address to the current private frame-pointer (fpp) and the current private stack-pointer (spp) of the accessing PP.

Level 4 Extending Level 3, the check for private accesses is extended to the whole stack, not just the current stack-frame. This could only be achieved by using `shint` instrumentation of the fork-lib startup routine. It is used by the simulator in order to trigger the recording of the first stack-frame address.

Level 5 Extending Level 4, the fifth level of granularity introduces the aspect of determining if an explicitly globally visible information actually is considered to be relevant or not. In an asynchronous region this relevance is marked and enforced by the use of explicit synchronization. For that purpose, the implementation distinguished between PPs actually running in asynchronous execution mode and PPs running in synchronous execution mode. In order to support the simulator to make this distinction, the Fork compiler has been extended to insert `shint` instructions when asynchronous program regions are entered or left.

It should be noted that the current implementation of the fifth level is not completely Fork compliant: if a synchronous group and at least one PP running in asynchronous mode are executed simultaneously, it may happen that not all PPs of the synchronous group read the same value from the same memory location at the same read instruction. However, the implementation has been developed in order to evaluate its possibilities and nevertheless can be safely used for simulations consisting only of phases where either all PPs are in synchronous mode or all PPs are in asynchronous mode.

Barrier The Fork barrier has the most important function for the whole concept to work correctly. since it has to ensure that all PPs waiting in the barrier will continue the execution at the beginning of the same segment. This constraint is derived from the idea of the *exact barrier* introduced in [KKT01]. The exact barrier is actually stricter than a standard barrier. It ensures that all PPs of the same group will continue the execution of the next instruction after the barrier at the same CPU cycle. This functionality is absolutely essential for the reunion of groups or switching between asynchronous and synchronous execution mode. Please refer to [KKT01, p. 142-147] for more detailed information about the concept and the implementation of the exact barrier.

Several problems arise if the barrier is executed in the parallel simulation without modifications. The first problem of the parallel simulation concept is that the simulation of a PP is only stopped if a write request occurs. This means a PP running into the barrier will enter a loop that does not include any write requests and then will be simulated forever, since no synchronization point can be reached if all threads running the simulation are actually busy waiting for a PP stuck in the barrier. Therefore the concept is modified to

include read requests from inside the barrier code into the list of conditions that will stop the simulation of the actual PP. Such requests are detected by `shints` added to the Fork library at the beginning and at the end of the barrier function.

The next problem is that read and write accesses inside the barrier will occur in the same synchronization phase. According to the basic implementation of parallelism in this section, `SimulateMem` will process all memory accesses that are pending which now includes read accesses. This causes a serious problem: not all PPs will be released at the same time. Since memory processing is done sequentially in the priority order of the PPs, a PP entering the barrier last will trigger the release only for those PPs processed afterwards. The other PPs will not be released until the next synchronization phase. The problem of simultaneous read and write accesses was prevented for the original SB-PRAM by dividing read and write instructions into odd and even processor cycles. With this concept in mind, it was decided to process read accesses only in the PPs simulation phase and write accesses detected as relevant propagation of information only in the synchronization phase. For the special case of the read access inside the barrier, the processing is delayed to the beginning of the next segment.

The final problem is that the releasing PPs will be one global synchronization behind the already waiting PPs. The solution is derived from the early wave delay technique used for the original SB-PRAM, which used two additional nop instructions to counteract a similar problem. The second nop is replaced by a special `shint`. This `shint` then triggers an additional synchronization phase for the PPs being early.

The handling of the barrier as described in this chapter has a major advantage over the handling by the sequential simulation. In the parallel simulation the barrier loop is only simulated once per segment. The actual length of the instruction stream processed by certain PPs in the same segment may be very different, i. e., this handling introduces a noticeable performance improvement, especially in sequential program regions processed only by one PP.

Runtime Estimation of Simulated Algorithms There are two major purposes for the simulation of PRAM algorithms: testing if the algorithm works correctly and observing its runtime behavior. The former purpose remains fully served by the parallel simulation, since it is Fork compliant. Serving the latter purpose may be only possible to a limited extent. The observation of the runtime behavior typically is done by using the program `trv` in order to visualize trace file outputs generated by a profiling run of of the algorithm [KKT01, Ke04]. Since the tracing ability was not subject of this work it can be considered as not present for the parallel simulation. Nevertheless, a basic form of observing runtime behavior is still possible: runtime estimation.

The Fork library offers read access to the processor cycle counter of the PPs via the `getct` function. This can be used to calculate the wall clock time of a certain part of the program, i. e., to determine the exact runtime of a parallel algorithm for a certain problem size and processor count. On the original SB-PRAM and in the sequential simulation, the cycle counts of all PPs are equal at every point in time. Hence, it does not matter which PP

actually obtains the result of `getct`. However, the individual progress of the PPs diverges during the parallel simulation. At the end of the program, the values of the cycle counters may be completely different. and possibly none of them represents the real time taken by the execution of the algorithm. In addition, the barrier implementation, as described in Section 3, introduces the new feature of temporarily suppressing the simulation of waiting PPs. This also implies it will be missing some processor cycles.

From a different point of view, the barrier itself can be used as global time reference. The barrier semantic guarantees all PPs belonging to the same group to leave the barrier at the same time – or from the view of the simulation: they have the same cycle count. In the sequential simulation, the waiting PPs are released when the last PP enters the barrier. In the parallel simulation "the last" PP actually means the PP that has made the most progress before entering the barrier. This point of view offers a conclusive image of the runtime behavior. The PPs may have diverging cycle counts during the simulation, but from their point of view this count represents the actual time, if it is synchronized to a global time reference whenever a barrier is left. This global time reference is determined by the maximum of all cycle counts of all PPs that entered the barrier and belong to the same group.

Therefore, the handling of the barrier is extended. The `shint` handler for the *barrier enter* event is extended by the procedure of determining the maximum cycle count. The `shint` handler for the *barrier leave* event is extended by the procedure of setting the cycle to this maximum plus the instruction length of the barrier itself. Of course, this has to be done with respect to the actual group topology. This is done by using the shared group-pointer (gps) as the key for a group individual maximum entry in a tree-map. This highly increases the accuracy of the simulation concerning the observation of the runtime behavior based on the processor cycle count.

4 Evaluation

Evaluation Setup and Definitions Subject of the evaluation was the runtime of original and parallel implementation of `pramsim` for several different algorithms as well as comparison of the algorithm's result (correctness) and the result of the estimated PRAM runtime (simulated SB-PRAM hardware processing time). All measurements were done on the same shared memory computer system utilizing up to 16 cpu cores. For comparison purposes the *relative* and the *real speedup* was calculated. The *relative speedup* was calculated by dividing the runtime of the parallel implementation using only one thread by the runtime of the same implementation using 16 threads. The *real speedup* was calculated by dividing the runtime of the original implementation by the runtime of the parallel implementation using 16 threads.

Summary of Results Obtained The Fork compiler package contains several example algorithms with very different characteristics regarding the number and the frequency of change between asynchronous and synchronous regions and groups. Extensive evaluations

using a set of these algorithms have shown that the parallel simulation of all these algorithms could achieve an *real speedup* between 12 and 31, depending on the algorithm. For some of them this was achieved without any modification whereas some others have been modified slightly. The outputs of the algorithms simulated by original and parallel simulation were identical for all cases – with exceptions at Level 5, as expected. The overall accuracy of the runtime estimation had an acceptable relative deviation between original and parallel simulation of at most 1%. Algorithms with dominating synchronous regions even do better up to exact results. A more detailed view on results and investigations is presented in Appendix B in Section 6, including the identification of further optimization potential.

5 Conclusions and Future Work

The SB-PRAM simulator in conjunction with the Fork programming language represents a good basis for the simulation of PRAM algorithms. In the past, there were several attempts to implement an efficient parallel version of this simulator which did not achieve any significant performance improvement or were restricted to certain classes of PRAM programs. The attempt described in this article introduced and enabled the concept of successive coarsening of the synchronization granularity. The basic mechanism of the concept is simulating single PPs until a situation occurs that requires synchronization. The actual decision if and when such a situation is considered to be present can be done by several extensible criteria. As part of this work, several levels of synchronization granularity have been introduced, implemented, and evaluated. This was mostly done with the assistance of instrumentation instructions inserted by an extension to the Fork compiler. The resulting parallel simulation showed a reasonable parallel performance without any restrictions to certain classes of PRAM algorithms while maintaining the possibility of runtime analysis. Even more, the concept still leaves room for even more improvement, which could be addressed in future work by the introduction of new compiler optimizations and/or an efficient parallel implementation of the `mpadd` instruction.

References

[BKK12] Brenner, Jürgen; Keller, Jörg; Kessler, Christoph: Executing PRAM Programs on GPUs. Procedia Computer Science, 9(0):1799 – 1806, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.

[Bl07] Blaar, Holger; Keller, Jörg; Kessler, Christoph; Wesarg, Bert: Emulating a PRAM on a Parallel Computer. In: PARS-2007 21. PARS-Workshop, Hamburg, Germany. GI Gesellschaft für Informatik eV, 2007.

[Cl07] Clauß, Carsten: Paralleler PRAM-Simulator. Master's thesis, FernUniversität in Hagen, Germany, January 2007.

[CLB12] Clauss, Carsten; Lankes, Stefan; Bemmerl, Thomas: Mapping the PRAM model onto the Intel SCC many-core processor. In: High Performance Computing and Simulation (HPCS), 2012 International Conference on. IEEE, pp. 395–402, 2012.

[FGK97] Formella, A; Grün, T.; Kessler, C.W.: The SB-PRAM: concept, design and construction. In: Massively Parallel Programming Models, 1997. Proceedings. Third Working Conference on. pp. 163–172, Nov 1997.

[FW78] Fortune, Steven; Wyllie, James: Parallelism in Random Access Machines. In: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing. STOC '78, ACM, New York, NY, USA, pp. 114–118, 1978.

[Ke04] Kessler, Christoph: A Practical Access to the Theory of Parallel Algorithms. SIGCSE Bull., 36(1):397–401, March 2004.

[KKT01] Keller, J.; Kessler, C.W.; Träff, J.: Practical PRAM programming. Wiley series on parallel and distributed computing. J. Wiley, 2001.

[KKW09] Keller, Jörg; Kessler, Christoph; Wesarg, Bert: Efficient Simulation of Fork Programs on Multicore Machines. In: Proc. 22nd PARS-Workshop. PARS'09, 2009.

[Va90] Valiant, Leslie G.: A Bridging Model for Parallel Computation. Commun. ACM, 33(8):103–111, August 1990.

[Vi10] Vishkin, U.: , Thinking in Parallel: Some Basic Data-Parallel Algorithms and Techniques, 2010. In use as class notes since 1993.

6 Appendix

A: Fork Example Algorithms An example PRAM algorithm in asynchronous mode was already mentioned and evaluated in [Cl07, CLB12]: a Jacobi over-relaxation solver for the well-known two-dimensional Laplace problem. The core of this solver is shown in Figure 4. The solver iterates K times over a two dimensional $N \times N$ mesh. As one can see, the work for the physical PRAM processors (PPs) is shared statically in lines 8 and 14 by the `forall` loop (`#` evaluates the actual number of PPs in the current group). If N is divisible by the number of processes, then the load will be evenly divided, since all PPs will process the same number of instructions. The iteration is divided into two phases: the calculation of the values for the next iteration and a copy phase. Both phases reside inside two asynchronous regions, each marked by a `farm` statement. The outer `start` statement marks a synchronous region, i. e. the asynchronous regions are synchronized by an implied barrier at the end of the region.

Since Fork was designed as a PRAM programming language, it offers much more potential to express parallel algorithms than just the OpenMP like fashion presented in Figure 4. The previously introduced algorithm solving the Laplace problem by Jacobi over-relaxation does not make use of real PRAM specific capabilities. Figure 5 outlines a CREW-PRAM variant of the algorithm utilizing $P := N^2$ processors. As one can see, the copy phase is not necessary any more. Additionally, the two nested inner loops are replaced by a complete static distribution of cells to processors (identified by the actual group processor id `$`). The complete algorithm is now running in synchronous execution mode, hence the synchronicity invariant applies. It guarantees for each iteration that first all PPs will simultaneously read the same, old values stored in `src`, then simultaneously compute the new result and, finally, simultaneously write their result to a distinct memory location inside the `src` array.

```
1:  start {
2:     pr int i, j, k;
3:
4:     // iterate K times
5:     for(k=0; k<K; k++) {
6:
7:        // calculate new destination values
8:        farm  forall(i,1,N-1,#)
9:           for(j=1; j<N-1; j++)
10:             dst[i][j] = 0.25 * (src[i-1][j] + src[i+1][j]
11:                                 + src[i][j-1] + src[i][j+1]);
12:
13:       // copy destination to source values
14:       farm forall(i,1,N-1,#)
15:          for(j=1; j<N-1; j++)
16:             src[i][j] = dst[i][j];
17:    }
18: }
```

Fig. 4: Core of Laplace Solver (based on [Cl07, CLB12])

```
1:  start {
2:     pr int i, j, k;
3:
4:     i = $ / SQRT_P + 1;
5:     j = $ % SQRT_P + 1;
6:
7:     // iterate K times
8:     for(k=0; k<K; k++) {
9:        src[i][j] = 0.25 * (src[i-1][j] + src[i+1][j]
10:                           + src[i][j-1] + src[i][j+1]);
11:    }
12: }
```

Fig. 5: Core of Synchronous Laplace Solver

B: Closer Inspection of the Introduced Laplace Algorithm In order to enable deeper insight into the design of the parallel SB-PRAM simulation, the performance of the Laplace solver algorithms, as described in Figure 4 and Figure 5 in Appendix A, are discussed in more detail.

For the case of the asynchronous solver (Figure 4) there will be the following points of synchronization depending on the level of granularity: Level 1 and 2 will result in any write to private or shared variables (see lines 5, 8, 9, 10, 14, 15 and 16) being a trigger for a synchronization phase. Level 3 and 4 will only trigger synchronization phases for writes into shared variables (lines 10 and 16). Level 5 will only synchronize at implicit points located at the end of the two farm statements outside the inner loops. This corresponds to the increasing *relative speedup* of the parallel simulation of a 64 PP PRAM ($p = n$) shown in Table 1 (second column).

For the case of the synchronous solver (Figure 5) synchronization will take place at the following locations: Level 1 and 2 will result in synchronization at any write to private or

shared variables (see lines 4, 5, 8 and 9). Level 3 and 4 will only trigger synchronization phases for writes into shared variables (line 9 only). Level 5 will have no benefit since there are no asynchronous regions. This corresponds to the *relative speedup* of the parallel simulation of a 4096 PP PRAM ($p = n^2$) also shown in Table 1 (third column). Obviously, the relative speedup is limited to the amount of Level 4 Graining – which seems to be similar for both asynchronous and synchronous Laplace solver. However, since the number of visible synchronization phases obviously is far less for the synchronous solver (only the `k` loop is present), a further examination of the presence of additional, implicit synchronization was done. It was investigated through revision of the Fork compiler and its assembly output that a major source of synchronization is caused by an `mpadd` instruction that is inserted into the assembly for each loop iteration. This is done by the Fork compiler since after any conditional branch the number of participating PPs in the current synchronous group may have changed. It is one of the mechanisms the compiler uses in order to enforce the synchronicity invariant.

Since `mpadd` is the most expensive instruction – from the perspective of the parallel simulation – a third variant of the Laplace solver was created. It was called semi-synchronous because it based on explicitly telling the Fork compiler that he should run the complete loop in Figure 5 line 8 to 11 in asynchronous execution mode by simply adding a `farm` to the beginning of line 8. As the compiler does not enforce the synchronicity invariant for asynchronous regions, it does not insert the `mpadd` instruction. However, even running in asynchronous execution mode – where PP instruction pointers may differ – does not mean that the instruction pointers of PPs necessarily have to be different. In this special case, all instruction sequences for all PPs will be the same. Hence, the output of the algorithm will be the same as the output of the synchronous variant. This applies even for the parallel simulation of the algorithm except for Graining Level 5. As one can see in Table 1 (fourth column), this results in a noticeable boost of the relative speedup from about 8 up to 12.

The insight of this experiment discovers the potential of further improvements. Future work may focus on the development of an efficient `mpadd` implementation. Even investigations on compiler based optimization is possible. One could focus on path prediction techniques in order to avoid the `mpadd` instruction (branch out only in last iteration). Also one could define `k` as a shared variable and try to implement detection and handling for parallel operations that will result in equal results for all PPs (since all PPs operate on the same `k` the results of increment and comparison will be equal, thus have to be computed only once).

Graining	Laplace async.	Laplace sync.	Laplace semisync.
Level 1 & 2	4.3	5.9	7.4
Level 3 & 4	8.2	8.2	12.1
Level 5	13.6	8.2	(invalid result) 15.4

Tab. 1: Relative Speedup for Parallel Simulation of Laplace Algorithms