Efficient Simulation of Fork Programs on Multicore Machines

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

[1] FernUniversität in Hagen, Dept. of Math. and Computer Science, 58084 Hagen, Germany
joerg.keller@fernuni-hagen.de
[2] Linköpings Universitet, Dept. of Computer and Inf. Science, 58183 Linköping, Sweden
chrke@ida.liu.se
[3] Techn. Univ. Dresden, Center for Inf. Services and High Performance Computing, 01062 Dresden, Germany
bert.wesarg@tu-dresden.de

Abstract. The SB-PRAM project resulted in the PRAM programming language Fork and in a complete tool chain from algorithms, libraries and a compiler to a prototype hardware and a sequential simulator of that prototype. With the prototype becoming defunct, the necessity arose to parallelize the simulator to be able to simulate larger Fork programs in a reasonable time on a multicore machine. We present an approach to increase the granularity of parallelism in the simulation and hence increase the speedup. The compiler forwards structural information of the Fork programming model via the code to the simulator. The simulator exploits this information at runtime to reduce synchronization overhead. We report on preliminary experimental results with a synthetic benchmark that indicate that our approach is advantageous.

Key words: PRAM simulation, granularity of parallelism, multicore programming

1 Introduction

The parallel random access machine (PRAM) is an important concept to study the inherent parallelism of problems and to devise parallel algorithms without consideration of or tweaking to architectural particulars of a target machine. While PRAM algorithms were considered unpractical for a long time, the SB-PRAM project provided the programming language Fork together with a compiler, a runtime system and the PAD library of basic algorithms on top of the SB-PRAM machine, a parallel architecture to efficiently execute Fork programs. In addition, the SB-PRAM simulator program provides support for Fork program development and debugging. See [1] for a detailed introduction into all of these issues.

Yet, as typically happens with prototypes, the SB-PRAM hardware became outdated quite fast and finally got disassembled in 2006. While the simulator remained, it is a sequential code and simulation of larger program instances would take days. As teaching of parallel algorithms and programming has become popular again because of multicore CPUs being rather the rule than the exception these days, and practical assignments should go beyond toy problems [2], we decided to provide a more powerful platform for Fork programming: a parallelized simulator program to emulate the SB-PRAM instruction set architecture on a contemporary shared memory parallel machine. While there are other PRAM languages such as *ll* [3], *e* [4] and XMTC [5], they either do not mention a simulator, or only provide a sequential simulator, so that we could hardly build on related work, except for parallel discrete event simulation in general [6].

In a previous attempt to parallelize this simulator [7] we explored the inherent parallelism between the PRAM processors but had to learn that this parallelism is very fine-grained, down to the level where the synchronization overhead is overall dominant. We were moderately successful in enlarging this granularity but could only achieve speedups on quite artificial program code.

In the present work we present a new approach to reduce the synchronization overhead by exploiting structural information available in the Fork programming model. Each Fork

program maintains a logical partitioning of the processors into groups. Initially all started processors run synchronously in a root group. Upon arrival of a `fork` statement or control flow statements such as `if` with a condition that depends on the processor ID (e.g. by using processor-local variables) the participating processors are split into subgroups, two in the latter case, more in the former case. At the end of the statement, the subgroups synchronize and join again to form the original group. The (sub)groups form a tree starting with the root group of all started processors. Only the leaf groups (which contain disjoint subsets of processors) of this tree are active. Fork only guarantees the synchronicity among the processors of each leaf group.

We have instrumented the Fork compiler to emit new instructions to communicate with the simulator in a one way fashion. These annotations signal the simulator when a group splits and when its subgroups will join again. With this information the simulator is capable to maintain the group tree by itself at runtime. The simulator uses this group tree to execute leaf node groups with low overhead. All PRAM processors in one leaf node group are simulated sequentially on one core round-robin instruction by instruction to maintain the synchronicity property of the Fork language, therefore avoiding any synchronization overhead of this fine-grained parallelism. Different leaf node groups can be executed in parallel without any synchronization because they are independent. Only in the case of group merges a synchronization is necessary. A side effect of this group monitoring approach is the avoidance of busy waiting when subgroups join again, so that already a 1-threaded version of our approach achieves a speedup over the previous simulator. With multi-threaded versions, we achieve a speedup over the previous simulator, and achieve a small speedup over the 1-threaded version for frequent situations. We explore the performance advantages with the help of a parameterizable synthetic benchmark.

The remainder of this article is organized as follows. In Sect. 2 we briefly summarize challenges in the simulation of PRAM programs that motivate our simulator design. In Sect. 3 we report on preliminary experimental results. Section 4 concludes.

2 Challenges in Parallel PRAM Simulation

The PRAM model assumes that all processors are executing their instructions synchronously, so this also is the semantics of the SB-PRAM instruction set architecture which is the target of the Fork compiler. In Fork all processors in the beginning execute the same instructions in an SPMD style like in MPI, i.e. they form a synchronous processor group. With the `fork`-statement, processors can be split into subgroups by evaluating an expression which is directly or indirectly dependent on the processor IDs. A similar situation with two subgroups appears when a group executes an `if`-statement with a condition somehow dependent on the processor IDs. Other situations where a group splits exist, such as a loop where the number of iterations is dependent on the processor IDs. The processors of each subgroup are synchronous, while the different subgroups are asynchronous. As a subgroup can be split again, at any time the groups form a group tree of which only the leaf node groups are currently active, and those leaves contain a partitioning of all PRAM processors.

When a group has split into subgroups, the subgroups unite again into the previous group after they have executed their block of instructions, such as `then` and `else` branches in the case of an `if`-statement. As the subgroups are asynchronous, they must pass a barrier to unite again. As the SB-PRAM was considered a single-user machine, a busy-waiting has been implemented.

In order to simulate the SB-PRAM sequentially, the processors are simulated round-robin, one instruction at a time. This guarantees the synchronous semantics but does not directly lead to a parallel simulation.

Note that there is also an asynchronous mode within Fork entered by the `farm`-statement. In this mode, all processors execute statements independently, i.e. they may become asynchronous. The user is responsible for taking appropriate action for coordination and avoiding race conditions, as is the case with other shared memory environments such as POSIX threads. Therefore, simulation on a multicore can be done by distributing the PRAM processors onto the cores and simulating larger blocks of instructions for each PRAM processor before switching to the next. As the asynchronous mode does not provide major challenges, we concentrate on the simulation of the synchronous mode.

In a previous attempt to parallelize the simulation for the synchronous mode [7], we have surveyed a number of possible approaches and finally implemented the time-warp optimistic simulation technique to perform a parallel simulation of all processors, taking access to shared memory as the events to be ordered by the time-warp simulation. However, the granularity of the available parallelism was so fine that only a moderate speedup could be achieved with a combination of techniques to coarsen that granularity, and on quite artificial benchmarks.

Therefore, we try a different approach here: The processors of each leaf node group are simulated sequentially as in the sequential simulator. The different leaf node groups can be simulated in parallel without overhead on different cores of our multicore platform as they are asynchronous. The PRAM shared memory is realized by allocation on the heap of the shared memory in the multicore platform. Access by PRAM processors of each group is serialized and hence the PRAM semantics is maintained. Access by different groups is not coordinated. The reasons are explained below. When a group is split into subgroups, the group tree representation in the simulator is extended, and the processors of the group are split onto the subgroups. As long as there are more threads than active leaf node groups, each group is assigned to a thread of its own. If there are more groups than threads, we use a runqueue of active groups served by the threads. When a group is joining again with another group, it is taken out of the runqueue until the other group has arrived as well. Thus, the busy waiting overhead is avoided. The threads are not terminated when groups join, but are re-used as we use a form of thread pooling. Also, there is no involuntary migration of groups between threads. One simulator thread executes a processor group until it either merges with its siblings or it splits itself into subgroups. Therefore the thread overhead is restricted to the split and merge of groups.

Unfortunately, the overhead for maintaining the group tree inside the simulator is not negligible in all cases. For example, if there are many leaf node groups that regularly merge again, then the threads regularly access the runqueue so that its lock has contention. Yet, in typical code with some structure the overhead is small as the results in Sect. 3 indicate.

In order to perform the operations above, the simulator must know when groups split and unite again. To achieve this, we enhanced the Fork compiler to instrument the generated code to provide this information to the simulator. The instrumentation comes in the form of additional instructions inserted by the compiler. We assigned a new instruction opcode to create the instrumentation instruction, because no existing instruction should be overloaded. This approach has the advantage that only the compiler backend had to be modified. The parallel simulator will not execute these instructions but act as described above.

The PRAM instruction set comprises read-modify-write instructions which are used as synchronization primitives between processor groups. These instructions were simulated with non-atomic instructions in the sequential simulator. In the multi-threaded simulator these instructions are implemented with atomic instructions provided by the host architecture. Using atomic memory access is not always required by the simulated code, but there is currently no information available whether the affected memory location will be accessed by multiple groups or not. This can lead to performance regressions in the simulator.

While two subgroups are asynchronous, they still both may access the same shared variable, such as in the following example code.

```
sh int a;
if (...private condition...) { // subgroup 1
   a = ...;
   ... = ... a ...;
} else { // subgroup 2
   ... = ... a ...;
}
```

Here subgroup 1 first writes into shared variable a, then a is read and the new value is used in the second assignment. Subgroup 2 reads a and uses it in the assignment. As Fork makes no assumption about the subgroups' progress relatively to each other, two cases can occur: either all processors in subgroup 2 read the old value of a, or all processors in subgroup 2 read the new value of a. Typically, we assume that an algorithm does not depend on which case actually occurs; we call such algorithms *robust*. Algorithms that are not robust may lead to race conditions depending on the simulation, so we do not consider them further.

Textbooks on PRAM algorithms such as [8,9,10,11] do not encounter this problem, as they do not treat hierarchical partitioning of processors in detail and consider all processors as one group executing one code synchronously, even when `if` statements with private conditions appear. Still, the algorithms presented there normally are robust.

3 Experimental Results

We use a synthetic benchmark to explore the benefits of our implementation. Although the code is rather simple, it covers a wide range of PRAM algorithms. The pseudo-code looks as follows:

```
times (4) {
    fork (#subgroups) {
        do_work(load +/- groupdev);
    }
    do_work(load * loadratio);
}
```

First, the p PRAM processors are split into an adjustable number of subgroups which all have the same size (assuming that #`subgroups` divides p.) Each subgroup then executes routine `do_work` with an adjustable `load` modified by variation `groupdev`. Then the subgroups synchronize and the root group (consisting of all p processors) executes routine `do_work` with the load scaled by parameter `loadratio`. The whole benchmark is repeated four times to even out runtime effects from other processes.

The parameter `load` serves to model the amount of useful work in contrast to the overhead of group splitting and joining. The parameter `groupdev` serves to model the runtime differences of different subgroups. The parameter `loadratio` serves to model the fraction of the simulation that is purely sequential because all PRAM processors form a single group.

All experiments simulate $p = 4096$ PRAM processors. We compare the sequential simulator with our approach using 1, 4 and 8 threads respectively, running on a quadcore processor (AMD Phenom II X4 920 processor, 4×2.8 GHz, 4×512 kB L2-cache, 6MB shared L3-cache.)

Fig. 1 depicts a situation with a very low workload of 16, no load deviation, and a strong sequential part with a load ratio of 50%, i.e. a situation quite uncomfortable for our approach. Still all versions achieve a speedup of about 1.4 for up to 256 subgroups, although we only see a small speedup of the multi-threaded versions over the single-threaded version for 2 and 4 subgroups, which however is a frequent case resulting from an if-statement (or two nested if-statements) with private condition. For large numbers of subgroups, the overhead in group tree administration strongly increases without providing appropriate benefit because the large number of subgroups leads to negative cache effects. The 8-threaded version is only slightly faster than the 4-threaded version in some situations, which indicates that already one thread per core is able to keep that core busy.

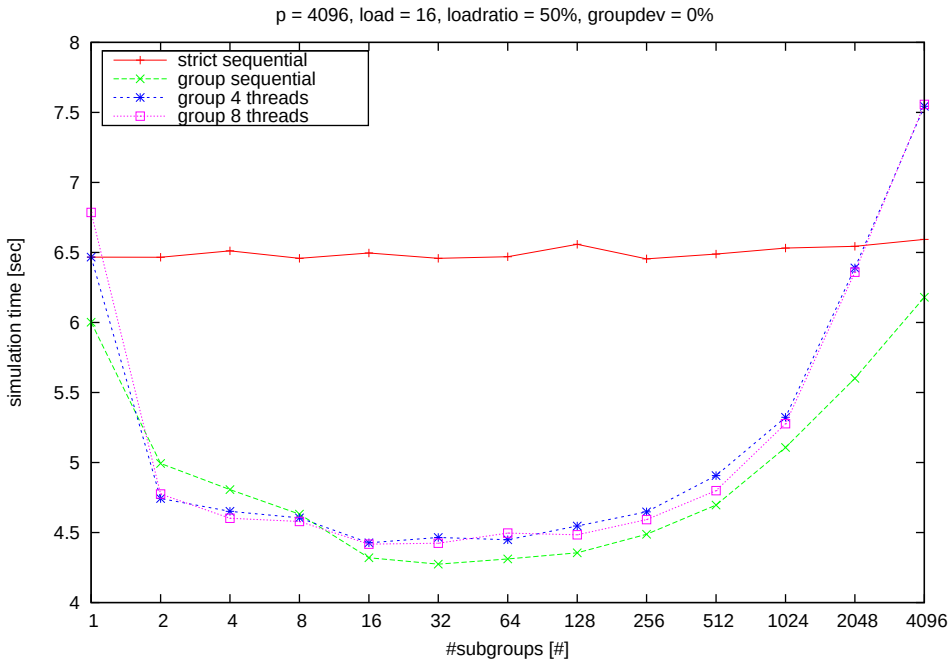

Fig. 1. Runtimes with low even workload and strong sequential part.

In Fig. 2 the load increases to 256. The situation is not really different from the previous one, which indicates that the group tree administration is not a bottleneck. In Fig. 3 we decrease the load ratio to 1% and see a sharp increase to a speedup of 2 over the sequential simulator, and a notable increase in the speedup of the multi-threaded versions over the single-threaded version for 2, 4 and 8 subgroups. In Fig. 4 we let the load deviate by up to 25%. We immediately notice that the runtime of the sequential simulator is increasing with the number of subgroups, because the load is not balanced anymore, and groups waiting for synchronization do a busy-wait. The speedup over the sequential version is larger than in any previous figure, because the groups waiting for a join with other groups are simply taken out of the run-queue until all groups participating in the join have arrived there. The speedup of the multi-threaded versions over the single threaded version for 2, 4 and 8 subgroups is still small but visible.

4 Conclusions

We have presented a new approach to accelerating the execution of Fork programs on multicore processors by exploiting the processor group structure in Fork. Our experimental

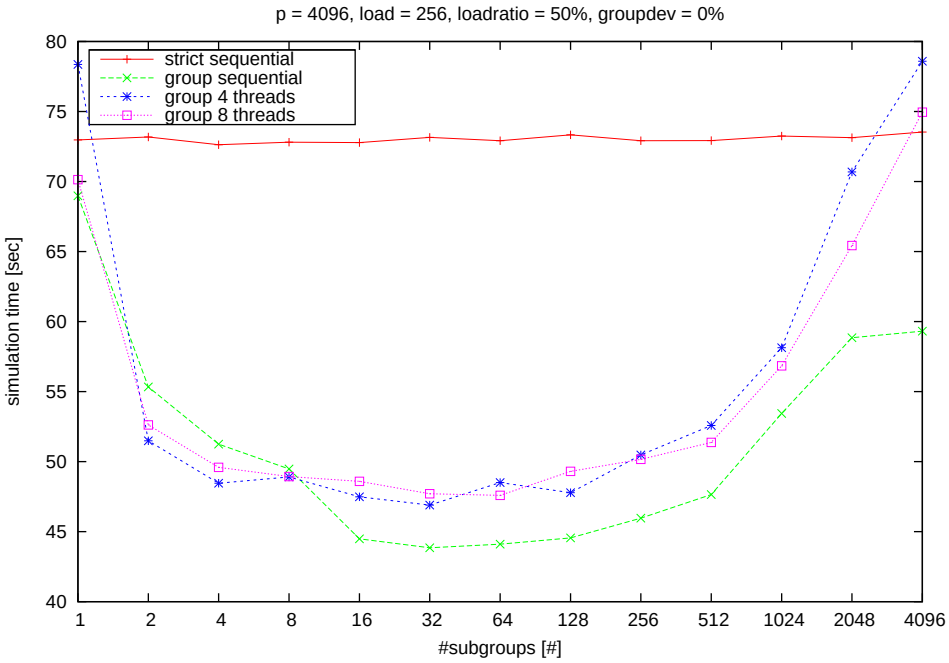

Fig. 2. Runtimes with workload increased compared to Fig. 1.

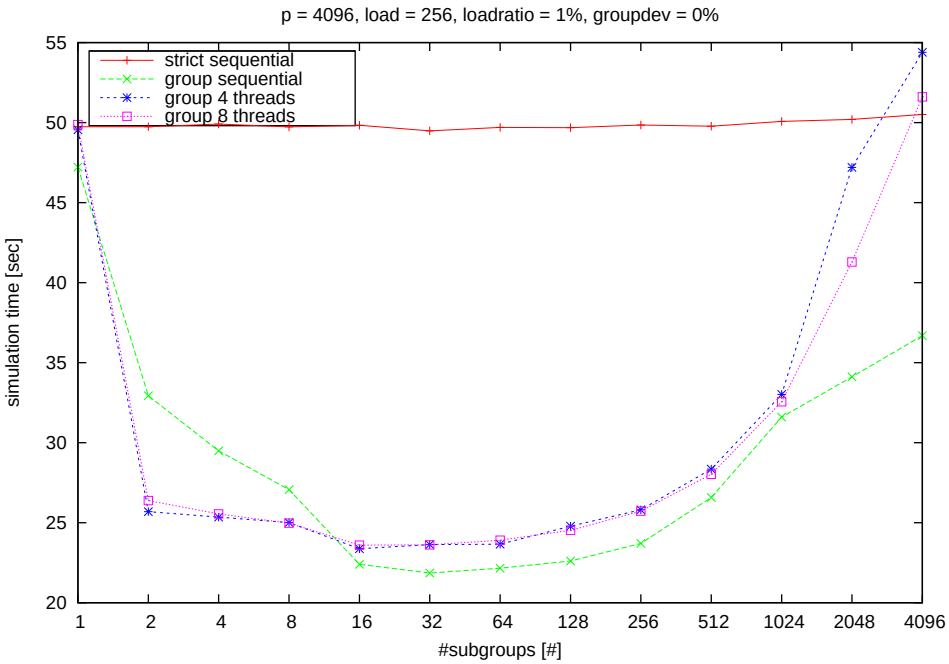

Fig. 3. Runtimes with sequential part decreased compared to Fig. 2.

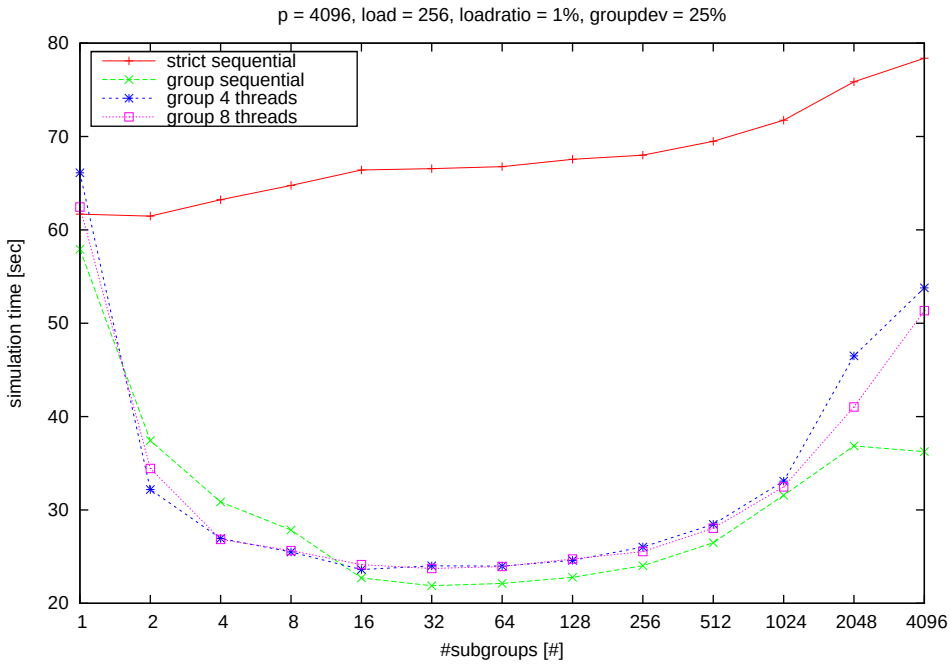

Fig. 4. Runtimes with load deviation increased compared to Fig. 3.

results indicate that already a single-threaded version shortens simulation time by a factor of 1.5 to 2. The multi-threaded versions achieve a small but noticeable speedup over the single-threaded version for the important case of a small number of subgroups.

For future work, another acceleration could be achieved by having the instruction stream for all processors of one group only decoded once and only execute the same instruction on different processors with different register contents as in a SIMD style.

It would also be nice to have a forecast on the simulation speedup to be expected when compiling the program. While this is possible in simple cases where the group tree structure is quite regular and not depending on input data, the general case would involve more or less a complete runtime analysis of the parallel program and thus seems out of reach.

References

1. Keller, J., Keßler, C.W., Träff, J.L.: Practical PRAM Programming. Wiley & Sons (2001)
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. León, C., Sande, F., Rodríguez, C., García, F.: A PRAM oriented language. In: Proc. EUROMICRO PDP'95 Workshop on Parallel and Distributed Processing. (1995) 182–191
4. Forsell, M.: e—a language for thread-level parallel programming on synchronous shared memory NOCs. WSEAS Transactions on Computers **3**(3) (2004) 807–812
5. Wen, X., Vishkin, U.: FPGA-based prototype of a PRAM-on-chip processor. In: CF '08: Proceedings of the 5th conference on Computing frontiers. (2008) 55–66
6. Fujimoto, R.M.: Parallel discrete event simulation. Communications of the ACM **33**(10) (1990) 30–53
7. Wesarg, B., Blaar, H., Keller, J., Kessler, C.: Emulating a PRAM on a parallel computer. In: Proc. 21st Workshop on Parallel Algorithms and Architectures (PARS 2007). (2007)
8. Akl, S.G.: The Design and Analysis of Parallel Algorithms. Prentice Hall, Englewood Cliffs, NJ (1989)
9. Gibbons, A., Rytter, W.: Efficient Parallel Algorithms. Cambridge University Press (1988)
10. JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley, Reading, MA (1992)
11. Karp, R.M., Ramachandran, V.L.: A survey of parallel algorithms for shared–memory machines. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science, Vol. A. Elsevier (1990) 869–941