

RESTRUCTURING A PARALLEL SIMULATION TO IMPROVE CACHE BEHAVIOR IN A SHARED-MEMORY MULTIPROCESSOR: THE VALUE OF DISTRIBUTED SYNCHRONIZATION

David R. Cheriton, Hendrik A. Goosen*, Hugh Holbrook, and Philip Machanick†
Computer Science Department
Stanford University

Abstract

Synchronization is a significant cost in many parallel programs, and can be a major bottleneck if it is handled in a centralized fashion using traditional shared-memory constructs such as barriers. In a parallel time-stepped simulation, the use of global synchronization primitives limits scalability, increases the sensitivity to load imbalance, and reduces the potential for exploiting locality to improve cache behavior.

This paper presents the results of an initial one-application study quantifying the costs and performance benefits of distributed, nearest neighbors synchronization. The application studied, MP3D, is a particle-based wind tunnel simulation. Our results for this one application on current shared-memory multiprocessors show a significant decrease in synchronization time using these techniques. We prototyped an application-independent library that implements distributed synchronization. The library allows a variety of parallel simulations to exploit these techniques without increasing the application programming beyond that of conventional approaches.

1 Introduction

Synchronization has a significant cost in many parallel programs. For example, Rose [Ros88] reports considerable savings from optimizing locking by exploiting application-specific semantics for a PCB board routing program. More recently, Karlin et al. [K⁺91] have shown significant speedup from adaptive locking, again illustrating the importance of synchronization.

Typically, locks are used to ensure mutually exclusive access to shared data structures. Parallel simulations require synchronization with respect to simulation time, in addition to mutual exclusion synchronization. Using a single centralized scheduler can result in excessive contention on this single resource. On a large-scale machine that uses fast processors and has high-latency global operations, application performance suffers. Global time synchronization also forces all processors to wait until the last processor completes its work as part of each time step. This waiting has a strong negative cumulative effect on efficiency over the execution of a program unless the load balancing is quite precise, especially on a moderately loaded multi-user machine.

Finally, global time synchronization across a parallel time-stepped simulation forces the program to update the entire model database before advancing to the next timestep. This is a serious problem on high-performance shared-memory multiprocessors that rely on caches for good performance: the model size normally dramatically exceeds the size of the caches, and therefore the model database must be brought into the processor caches on each timestep.

To make synchronization techniques scalable for future scalable shared-memory architectures, distributed approaches must be developed. Ideally, these should be packaged in reusable libraries suitable for a range of applications.

As a modest step in this direction, this paper presents the results of an initial study quantifying the performance benefits of distributed nearest neighbors synchronization. The one application studied and evaluated to date, MP3D [MB88], is a particle-based wind tunnel simulation that originally used global time synchronization.

For this study, the program was completely rewritten to explore distributed synchronization. We implemented neighbor-wise time synchronization between adjacent *precincts* of space, eliminating all global synchronization from the program. The new version of the program is also divided into a *library* layer and an *application* layer, as a preliminary attempt at providing application-independent run-time support for the techniques.

Our results for this one application on a current shared-memory multiprocessor show a significant decrease in synchronization overhead from these techniques, while adding only a small overhead to the execution time. As machines scale up, scalable synchronization should give substantial performance increases compared to global synchronization methods.

The next section describes the MP3D program structure, including some aspects of the supporting library and previous program versions of relevance to our performance evaluation discussion. Section 3 presents performance measurements of the revised program. A summary of related work is followed by conclusions and an outline of future research.

2 Program Structure

The major elements of MP3D are illustrated in Figure 1. Particles are generated from the reservoir, flow through the wind tunnel, collide with other particles, boundaries and the obstacle, and exit eventually to the meter, which feeds back to the reservoir to control the particle generation rate.

*On leave from the Computer Science Department, University of Cape Town, South Africa.

†On leave from the Computer Science Department, University of the Witwatersrand, Johannesburg, South Africa.

The wind tunnel space is divided into unit cubes called *cells*, which are the smallest unit of space used for taking measurements and detecting collisions between particles. In one timestep, all particles are moved, and some (depending on a statistical collision selection function) are collided [MB88]. No particle moves more than a cell length in a timestep, and collisions and moves only require local information (within a cell).

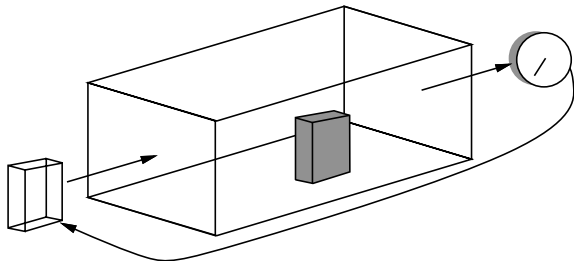


Figure 1: Major elements of the MP3D wind tunnel

2.1 MP3D Program Versions

MP3D has evolved through 3 versions. The original version, MP3D-0, was closely based on a Cray-2 implementation [MB88] that focused on efficient vector processing execution. The resulting data structures and algorithms suffered from poor locality, causing a large number of cache misses, both from replacement and from multiprocessor cache contention. MP3D-1 slightly improved on this by storing particle data in a cache block aligned C++ object, reducing both replacement misses and false sharing.

In MP3D-2, space is divided into contiguous regions called *precincts* which are the unit of load allocation. On each time step, each processor moves the particles in all the cells of the precincts allocated to it, resulting in better processor locality and increased performance, especially as processors become faster relative to memory.

MP3D-2 still uses the MP3D-0 global barrier synchronization of timesteps, provided by the ANL (Argonne National Laboratories) Parmacs macros [LO⁺87]. In MP3D-3, the use of barriers is replaced with time synchronization between neighboring precincts, as described below.

2.2 Distributed Synchronization

Each precinct has its own *local clock*, and each processor has a queue of precincts that it executes according to the following algorithm. A precinct p with local time t_p is scheduled when the minimum simulation time among all its neighbor precincts is at least t_p . Execution continues until $t_p >$ at least one neighbor precinct's simulation time. At that point, the next eligible precinct is scheduled.

Neighbor clocks can be checked very efficiently on a shared memory multiprocessor using an unsynchronized memory access to the neighbor's clock, given that the clock values increase monotonically (because there is no rollback). We assume that writing a new clock value is an atomic operation for the hardware. On a distributed memory machine, each precinct should send clock updates to its neighbors peri-

odically, analogous to the Bryant-Chandy-Misra [Fuj90] null messages.

We also implemented barrier synchronization as a runtime option in MP3D-3 for performance comparison with the distributed synchronization scheme.

2.3 Supporting Class Library

Space is divided into *precincts* which are the minimum unit of work for scheduling and load balance. Precincts are further divided into *space units* which are the minimum units that can be atomically processed for a specific application. *Messages* are an abstraction of information that may move between space units. Messages can be enqueued on a lockable queue within a space unit to allow transfer of information to other processors. Each precinct has a list of neighbors for implementing distributed synchronization. A *space directory* is used to find a space unit based on co-ordinates of a point in space.

Space units, precincts and messages are implemented as C++ classes, and each object is padded and aligned to cache block boundaries to minimize cache misses and maximize the prefetch effect of large cache blocks [CGM91].

The class library has substantially reduced the amount of MP3D-specific code and facilitates the incorporation of these techniques in other applications. The overhead introduced by the library layer seems to be modest. MP3D-2 and MP3D-3 have similar performance of slightly more than 6 microseconds per particle per timestep on 8 processors. However, it is difficult to make direct comparisons between MP3D-2 and MP3D-3, because we reduced the number of synchronization operations per timestep, and improved the layout of data.

3 Results

The revised program was measured running on a Silicon Graphics 4D/380 with eight 33MHz MIPS R3000 processors and 256Mbytes of RAM. Each CPU has a 64K data cache, 64K instruction cache, and 256K of unified second level cache. All measurements presented here were obtained by running on an otherwise unloaded system, using a 16MHz counter on the Ethernet board for detailed timing (62.5 ns resolution). Static load balancing is used, and is the same for all runs of the program. While this is a small-scale machine, it uses fast processors, and the memory latencies are approaching those expected for future scalable shared memory architectures, such as the ParaDiGM [CGB91] and DASH [L⁺92] architectures.

The workload for these measurements is a large 131 by 131 by 7 cell wind tunnel with a cylinder as test object. Unless otherwise stated, we used 1,000,000 particles in the wind tunnel, resulting in a total data set size in excess of 120MB. The wind tunnel was run for 1500 time steps, which is a typical scenario for real measurements.

First we compare the waiting time of the distributed synchronization scheme to that of barrier synchronization. Figure 2 shows the cumulative synchronization times for both schemes during an 8-processor run. The waiting time for the distributed synchronization scheme is about half of the barrier waiting time. In addition, the barrier waiting time in-

creases more rapidly as the simulation progresses, while the distributed synchronization waiting time is linear.

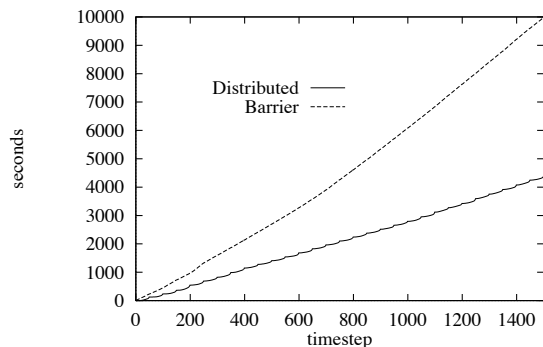


Figure 2: Cumulative synchronization time for all processors.

In Figure 3 we present measurements that show how the benefit from distributed synchronization increases as the amount of parallelism is varied from 1 processor to 8 processors. Observe that the cost of distributed synchroniza-

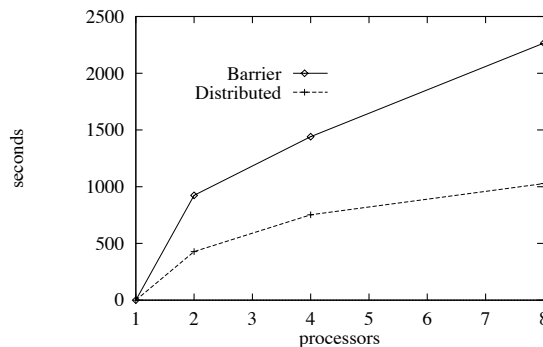


Figure 3: Total synchronization time vs increased parallelism.

tion is growing more slowly than the cost of barrier synchronization, at least on this small-scale multiprocessor. Measurements show that lower sensitivity to the fluctuating load balance is responsible for most of the improvement. In the case of barrier synchronization, all the processors have to wait for the slowest processor on each timestep. In the distributed scheme, processors only wait for neighbors, and therefore short-term load fluctuations are much less of a problem, as illustrated in Figure 4. This is a simplified one-dimensional model, so that each precinct has at most 2 neighbors (precincts 0 and 6 have only one neighbor each because they are at the ends). Assume that all precincts have finished processing timestep t , that precincts 1 through 5 have finished processing timestep $t+1$, but that precincts 0 and 6 are still busy processing timestep $t+1$. With barrier synchronization we have to wait for all the precincts to finish processing timestep $t+1$ before starting work on any of the precincts for timestep $t+2$. The distributed scheme, on the other hand, can continue scheduling precincts until timestep $t+3$: precincts 2, 3, and 4 can do work for timestep $t+2$,

and precinct 3 for timestep $t+3$. The work on timestep $t+3$ for precinct 3 can only begin after precincts 2, 3, and 4 have finished working on timestep $t+2$. A short-term load fluctuation that holds up precincts 0 and 6 does not immediately restrict us, and we also have time to recognize the imbalance and correct it before it impacts performance.

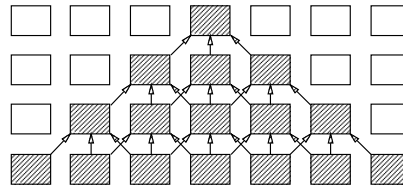


Figure 4: Tolerating load fluctuations through distributed synchronization.

There is additional overhead associated with distributed synchronization because we have to read the timestep counters of all neighboring precincts before proceeding. This overhead cancels out much of the benefit of distributed synchronization on small-scale parallel machines, such as the one we performed our measurements on. The total execution times of the two schemes are thus within 2% of each other.

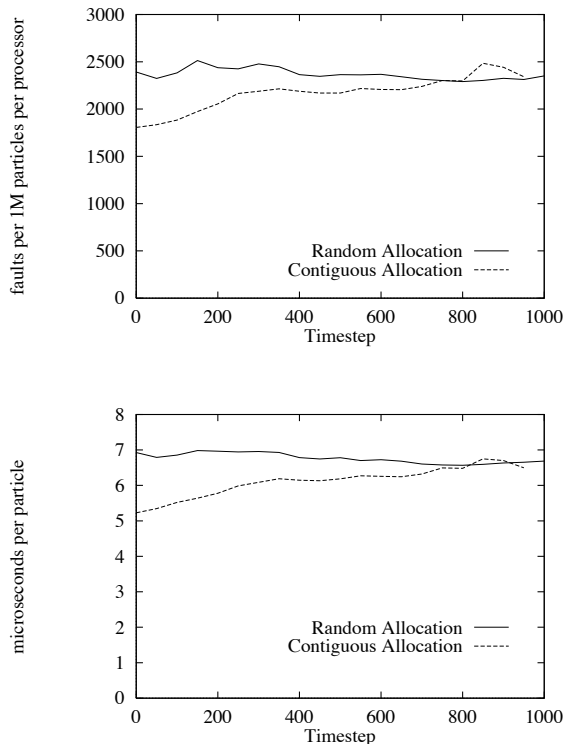


Figure 5: (a) L2 TLB misses per processor per timestep (b) Execution time per timestep, showing effect of increasing TLB misses.

An unexpected complication that our detailed measurements revealed is the substantial performance impact of

Translation Lookaside Buffer (TLB) misses. A TLB is required to translate virtual addresses to physical addresses before accessing a physically addressed cache. In our normal memory allocation scheme we pay no attention to the page locality of data, focusing instead on cache block alignment. To measure the effect of TLB misses we changed the memory allocation scheme so that all the data in a precinct (including particles) are allocated contiguously. Initially we expect relatively few TLB misses because each precinct uses data from only a few pages. Later, as particles execute a biased random walk through the wind tunnel, the particle data are scattered over a large number of pages, which should lead to an increased number of TLB misses.

Figure 5 (a) shows the number of second level TLB misses¹ per timestep per processor, as we compare an 8-processor run with contiguous allocation to one using arbitrary allocation. Figure 5 (b) shows that this has a significant effect on the execution time. As machines scale up in size and performance, TLB misses will become increasingly important, especially for large simulation, assuming the use of physically addressed caches.

4 Related Work

The synchronization scheme described here is a hybrid of Bryant-Chandy-Misra (BCM) [CM81] or *conservative* simulation, and traditional time-stepped simulations. In BCM, a parallel or distributed simulation is divided into logical processes (LPs) each of which may be on a separate processor. In a time-stepped simulation, on the other hand, the entire model state is advanced before any work is done on the next time-step. Neighbor-wise synchronization is similar to BCM if each precinct is regarded as an LP that communicates with its neighbors.² The primary contribution of our work has been to apply BCM synchronization to logical processes doing portions of the overall timestepped computation, rather than a conventional barrier synchronization scheme, and present performance measurements of this hybrid program structure.

5 Conclusions

Distributed synchronization on a relatively large-scale simulation was shown to provide a significant decrease in synchronization overhead. Most of the improvement comes from a reduced sensitivity to temporary load imbalances. We expect that the reduction in global communication will lead to a substantial performance improvement on large-scale machines, especially those (like Paradigm and DASH) that provide clusters with fast local communication.

Our experience with an application-independent library suggests that a variety of parallel simulations can exploit these techniques without increasing the application programming problem beyond that of traditional shared-memory approaches.

Further work is required to quantify the benefit of these techniques for other parallel applications and other architec-

¹Second level TLB misses happen when the page table entry that maps the referenced page is not present in the TLB.

²Particles cannot directly move between non-adjacent precincts because the precinct is much larger than the mean free path.

tures. Additional applications are also required to evaluate the generality of our class library approach, although our primary focus remains on physical simulations of 3 dimensional systems on shared memory multiprocessors. We regard the work reported here as a modest step in the direction of providing general, efficient and sophisticated execution support for parallel simulations on scalable multiprocessor systems. The results to date have been promising. Although much remains to be done, we are confident that the software structures and techniques we have described will contribute to realizing the full potential of scalable shared-memory multiprocessors for parallel simulation.

6 Acknowledgements

This work was sponsored in large part by the Defense Advanced Research Projects Agency under Contract N00014-88-K-0619. Hugh Holbrook is supported in part by a USAF Rome Laboratory Graduate Fellowship. Hendrik Goosen is supported in part by grants from the Foundation for Research Development (South Africa) and the UCT Research Committee. Jeff McDonald of NASA-AMES generously made the original version of MP3D available to us for this work, helped us understand the physics of the simulation and the details of the algorithm, and assisted with the validation of new variants.

References

- [CGB91] D.R. Cheriton, H.A. Goosen, and P.D. Boyle. ParaDiGM: A Highly Scalable Shared Memory Multicomputer Architecture. *IEEE Computer*, 24(2):33–46, February 1991.
- [CGM91] D.R. Cheriton, H.A. Goosen, and P. Machanick. Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: A First Experience. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*. Tokyo, April 1991.
- [CM81] K. M. Chandy and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *CACM*, 24(4):198–206, April 1981.
- [K⁺91] A. Karlin et al. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proc. Symp. on Oper. Sys. Principles*, pages 41–55. ACM, October 1991.
- [L⁺92] D. Lenoski et al. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.
- [Fuj90] R.M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, October 1990.
- [LO⁺87] E. Lusk, R. Overbeek, et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.
- [MB88] J.D. McDonald and D. Baganoff. Vectorization of a Particle Simulation Method for Hypersonic Rarefied Flow. In *AIAA Thermophysics, Plasmadynamics and Lasers Conference Proceedings*, June 1988.
- [Ros88] Jonathan Rose. Locusroute: A parallel global router for standard cells. In *Proc. 25th Design Automation Conference*, pages 189–195. IEEE, June 1988.