SpaceLib: A Library for Shared-Memory Parallel Applications

PHILIP MACHANICK*
Department of Computer Science
University of Cape Town
Private Bag
Rondebosch
7700
e-mail: philip@concave.cs.wits.ac.za

This paper presents preliminary results from design and implementation of a library for shared-memory parallel applications, SpaceLib. SpaceLib is written in C++, and is designed to facilitate efficient use of caches, taking into account locality and the fact that caches are organized into fixed-sized blocks. Before the implementation of SpaceLib, restructuring of a particle-based wind tunnel simulation called MP3D showed that a cache-sensitive approach had significant advantages. Memory-system simulation showed big reductions in cache misses. Run times on several architectures showed that sensitivity to caches is increasingly important on more recent designs because processor speed is improving faster than DRAM (ordinary main memory) speed. MP3D has been re-implemented on top of SpaceLib with promising results; further research includes more detailed measurement and implementation of other applications.

1. Introduction

The current trend in new microprocessor announcements is a speed improvement of 1.5 to 2 times per year, while memory speeds are improving much more slowly—about 7% per year [Hennessy and Jouppi 1991].

This rapid improvement makes it increasingly attractive to base high-performance multiprocessor computers on off-the-shelf microprocessors. However some of the design decisions which are suitable for uniprocessors may conflict with requirements for designing high-performance multiprocessor systems.

This work focuses on shared-memory multiprocessors, and issues resulting from their implementation using off-the-shelf components.

One issue is the use of relatively large cache blocks. In a uniprocessor system, large blocks are a suitable technique for reducing the impact of the high latency of main memory. In a multiprocessor system using shared memory, large cache blocks cause *false sharing*. False sharing is the presence of unrelated data in the same block which—if the data belongs to different processors—results in unnecessary contention. Such issues will become increasingly significant as the speed difference between processors and memory grows.

If such problems are to be avoided without making undue demands on the programmer, there are a range of possible options open: tools to aid in finding performance bottlenecks, programming languages that minimize memory contention, compilers that automatically re-arrange memory, and cache management at a finer-grained level than the cache block are a few examples. The approach in this work is to develop an object-oriented library that can be used as a basis for a range of applications. The advantage of this approach is that it does not require building a complex infrastructure typically required of the other approaches listed; the drawback is that it requires substantially rewriting code. However this limitation is not a problem in the case of new problems.

This paper starts off by introducing a specific shared-memory parallel application used in preliminary work—a particle-based wind tunnel simulation called MP3D—and explains problems with the way an initial version of MP3D was implemented. Steps taken to improve MP3D (from the original MP3D–0 through to MP3D–2) are described, followed by a description of an object-oriented library in C++ to support such applications. Results from re-implementing MP3D (as MP3D–3) on top of this library are presented. Future work is presented followed by a conclusion summarizing work to date.

2. MP3D

MP3D is a particle-based wind tunnel simulation developed at NASA–Ames [McDonald and Baganoff 1988]. It is a time-stepped simulation; every particle (representing some large number of molecules) is moved each step. Space is divided into *cells* which are unit cubes used as the minimum unit of measurement, and for determining whether collisions have occurred. At each timestep, every particle in a cell is processed in two steps:

---

* On leave from Department of Computer Science, University of the Witwatersrand, 2050 Wits.

1. move including boundary interactions
2. collide (not all particles collide; those that do are randomly chosen, then randomly paired with collision partners)

MP3D was originally written for a vector architecture (Cray), and the structure of the original shared-memory implementation (referred to here as MP3D–0) reflects this.

Each particle has eight physical attributes, each of which is represented as a floating-point number. The particles are collectively represented in eight separate arrays, one for each attribute. This means that data representing one particle is widely spaced in memory. Each processor randomly contends for particles, so that at each timestep one particle may be processed by a different processor at either phase.

This organization has a number of problems on a cache-based architecture:

1. data is not *co-located* (related data is not close together so a large cache block has little or no prefetch effect)
2. data has no *processor affinity* (the cost of a cache miss is not amortized over as large a number of accesses as possible)
3. *false sharing* is not prevented but in fact occurs with relatively small blocks because of the lack of co-location

3. Improvements to MP3D

As a first attempt at addressing these problems, MP3D was restructured minimally to place particle data in C++ objects; with the option of padding and aligning these objects to fit cache block boundaries. This version is called MP3D–2.

To further improve the program's cache behaviour, a more complete restructuring was undertaken. Space was divided into *precincts*, each of which was allocated to a specific processor. More attention was paid to padding and alignment of data structures to cache block boundaries.

Memory system simulations showed that, depending on the block size, MP3D–1 improved on the miss ratio for shared data of MP3D–0 by a factor of 1.5 to 6.7. MP3D–2 improved on MP3D–0 by a factor of 11 to 21 (figure 1).

The effect these changes had on run times reflect the increasing significance of memory hierarchy sensitivity in more recent designs. Measurements were taken on two generations of Silicon Graphics and DEC Firefly shared-memory multiprocessors. In both cases, the more recent machine—with essentially the same memory system but faster processors—benefited more from the restructuring exercise. On 4 processors on the older model of Firefly, MP3D–0 achieved a speedup of 2.2, whereas on the newer model, speedup was only 1.2 (barely faster than on the 2-processor run where speedup was 1.1). For MP3D–1 speedup was much the same,

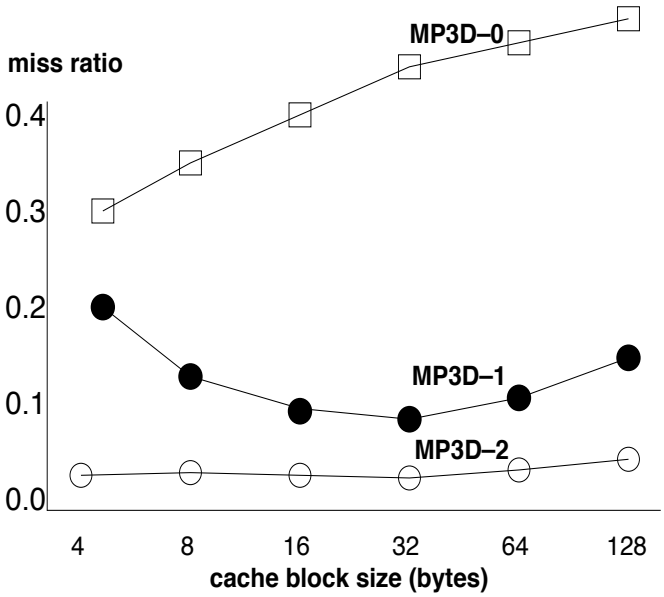

figure 1. Restructuring MP3D: memory simulation results

machine	number of processors	MP3D–0 time	MP3D–0 speedup	MP3D–1 time	MP3D–1 speedup	MP3D–2 time	MP3D–2 speedup
SGI 4D/240	1	677		597		678	
	2	503	1.3	410	1.5	369	1.8
	4	407	1.7	283	2.1	216	3.1
SGI 4D/380	1	544		437		569	
	2	455	1.2	315	1.4	312	1.8
	4	409	1.3	268	1.6	179	3.2
	8	431	1.3	265	1.6	141	4.0
MicroVax Firefly	1	12839		12700		9657	
	2	8096	1.6	8389	1.5	5116	1.9
	4	5818	2.2	5810	2.2	2732	3.5
CVAX Firefly	1	4854		4161		3303	
	2	4478	1.1	3964	1.0	1790	1.8
	4	4002	1.2	3632	1.1	1014	3.3

table 1. Performance of MP3D variations: run time in seconds and speedup

though overall run time was down. MP3D–2 on the other had achieved a speedup of 3.5 on the older machine, and 3.3 on the newer model. The speed improvement between the older Firefly using the MicroVax II processor and the newer CVAX is claimed to be 2.5 by the designers. MP3D–2 in fact achieved a speed improvement (using 4 processors) of 2.8, while MP3D–0 and MP3D–1 only managed an improvement of a factor of 1.5. Looked at in another way, MP3D–2 exceeded the designers' target performance improvement by 12%, while the other less well-structured versions were only able to use 60% of the designed performance improvement.

Similarly, on two generations of Silicon Graphics (older 4D/240 and the 4D/380), MP3D–2 more fully exploited the performance of the approximately 24% faster processors on the newer machine (table 1).

These results are reported in more detail elsewhere [Cheriton *et al*. 1990].

4. SpaceLib: MP3D–3 and Beyond

SpaceLib is implemented in C++, and is based on a layered model. The innermost layer is closest to the machine, while the outermost layer is the application layer. The layers from innermost outwards are:

1. Kernel—machine-specific shared memory allocation and deallocation, including padding and alignment to cache blocks; low-level synchronization primitives; messages used to convey units of work between precincts
2. Environment—reading in and interpreting a file describing the structure of space including how it is divided into precincts
3. Space—division of space into precincts and smaller *space units* (the smallest unit treated atomically for purposes of dispatching work); includes dispatching of work to precincts and synchronization

Synchronization in the original MP3D and the restructured versions up to MP3D–2 was by use of a barrier after each phase and at the end of the timestep.

SpaceLib introduces *distributed synchronization*, in which each precinct has a clock and ca be dispatched when its time is at or before that of its nearest neighbours in space. Each processor has a queue of precincts which it dispatches repeatedly in order, provided the condition on nearest neighbours is met. This synchronization model is designed to reduce sensitivity to short-term load imbalance. By contrast, when barriers

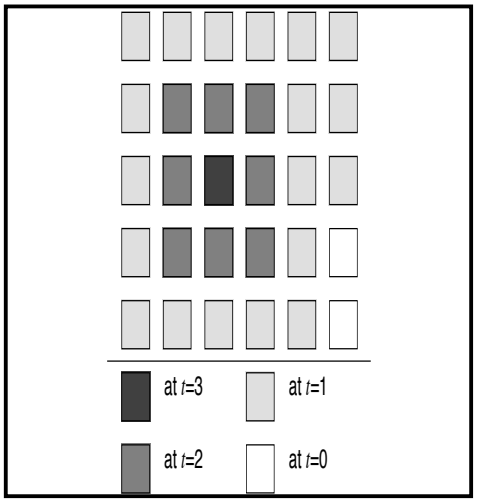

figure 2. Different precincts at different times

are employed, any processor with a slightly too much load holds up all the others. Also, distributed synchronization is a constant times the number of precincts (there are 26 nearest neighbours in three-dimensional space; fewer on boarders of space), whereas barriers require global communication and are inherently less scalable as the number of processors increases.

This does not do away with a requirement for load balancing but allows some slack time during which a relatively lightly loaded processor could be used for running a load balance algorithm. Load balance has not bee fully implemented in the current version of the library.

Figure 2 (simplified 2-dimensional representation) illustrates how some precincts may be able to proceed ahead of others. Precincts which are completely surrounded by others which are at their time or later (same or darker shading) can be dispatched; others have to wait. The darkest precinct at the centre is the furthest ahead in time and can only proceed once its neighbours have completed a timestep; the precincts at the edges which have no lighter-shaded neighbours can be dispatched.

MP3D–3 is implemented on top of SpaceLib, and is a complete rewrite rather than another restructuring.

The application layer mostly extends the space layer of the library, though some extensions of the environment layer are also needed (setting up space in a way specific to the application—in this case placing an object in the wind tunnel, and interpreting any additional command line arguments).

MP3D cells are implemented on top of the space unit class, and particles on top of the message class.

One unexpected discovery during testing and performance timing of MP3D–3 was the impact of TLB misses. The translation lookaside buffer is a small cache of page address translations which is consulted as a faster alternative to doing a page translation through page tables in main memory or conventional cache.

Since particles do not necessarily remain in close proximity to those that were allocated memory at the same time as they were, there is a high probability that most particles in a precinct are allocated in different pages. As a result of performance problems discussed in the next section, an option of keeping data in a given precinct in pages allocated for that precinct was also implemented. This is done by copying a message (in the case of MP3D, a particle) as it crosses a precinct boundary. This *message passing* variant is currently being re-implemented for greater efficiency.

5. MP3D–3 Preliminary Results

The major results available for MP3D–3 are synchronization cost gains and effects of TLB misses [Cheriton *et al*. 1993].

Performance has been measured in more detail but since the program is still under development, these results are not presented in detail. Compared with a time of 2.7μs to move a particle through a time step, MP3D–2 takes 6μs on a 1-million particle example; MP3D–3 has been measured to take under 5μs.

For purpose of measurement, SpaceLib includes an option of using a barrier instead of distributed synchronization. Measurements show that as the number of processors increases, the difference in synchronization cost of barriers is increasingly worse than that of distributed synchronization, as might be expected. The measurement shown in figure 3 is of total synchronization *waiting* time—the combined time over all processors spent waiting for other processors.

Note that the barrier cost not only climbs more steeply but the distributed synchronization cost increases less steeply as the number of processors increases. In all cases there is only static load balancing, so there is some aspect of load imbalance in the measured wait time; nonetheless the results support the claim that distributed synchronization scales better than barrier synchronization.

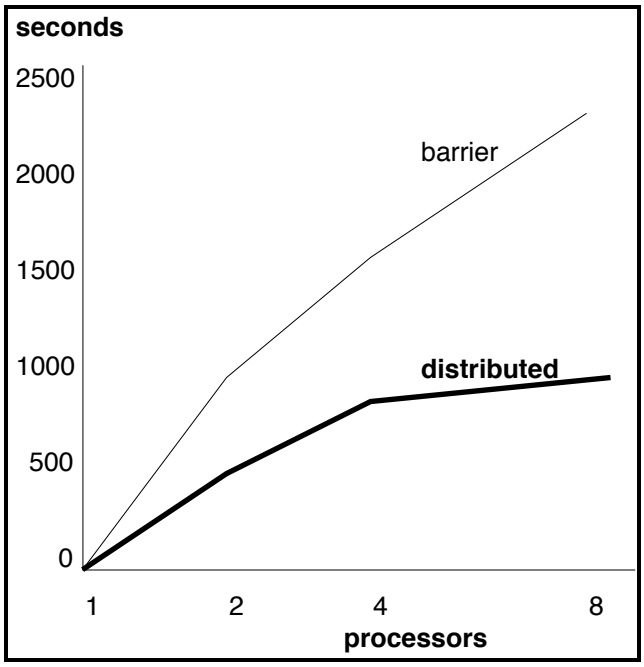

figure 3. Barriers *vs.* distributed: total synchronization wait time

If particles are randomly allocated (in keeping with MP3D–0), TLB misses are very frequent. If instead initial data is read from a file so particles in a given region of space are all allocated at the same time, there are very few TLB misses at first, but the number steadily increases. On average, a particle take 3 timesteps to cross a cell, and the wind tunnel is 131 cells long. After 500 timesteps the number of TLB misses in the two alternative initialization strategies start to converge. Not all particles go straight through, so only after 800 steps do the measures on the two cases fully converge.

Figure 4 illustrates the effect of TLB misses on execution time (time to push a particle through a timestep) on an 8-processor run with contiguous initialized from file data and randomly allocated data.

6. Future Work

Work in progress includes the message-passing code to reduce TLB misses. The initial attempt at this code—in which messages (particles) are copied every time they move to a new precinct—cost essentially as much execution time as it saved. As figure 4 illustrates, over 200 steps an initially contiguous allocation results in

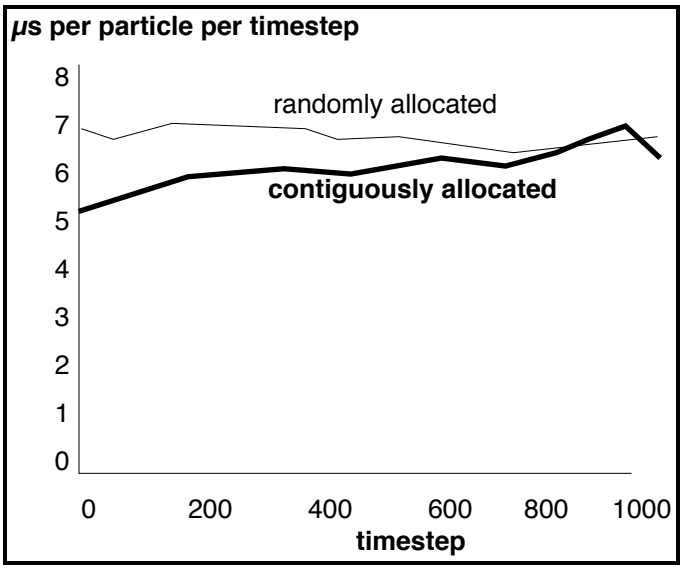

figure 4. Effect of TLB misses

relatively few TLB misses. Only doing copying after 200 steps (for this specific example) would be a good strategy; how to implement and parametrize this in general is currently under investigation.

Measurement of cache behaviour of SpaceLib applications will be done to compare with earlier work.

Other applications are also under investigation. The most likely next application to be implemented is the Barnes-Hut method for computing forces in an *n*-body problem [Barnes and Hut 1986]. This method divides space hierarchically using an octree. A leaf of the tree represents a cube in space containing only one body; an interior node represents a cube containing multiple masses represented as a point mass. The side of a cube and the contained masses are used to decide whether the current level of detail is sufficient, or whether further detail should be used lower down the tree. An octree is used because each cube is subdivided into 8 parts (or fewer since cubes containing empty space are discarded).

Both MP3D and Barnes-Hut are good representative applications of a relatively new programs which are relatively interesting in their overall approach and good candidates for implementation using data structures more sophisticated than arrays traditionally found in scientific applications. They are also part of the widely accepted Stanford SPLASH benchmark suite [Singh *et al*. 1992]. They are non-trivial and well accepted as benchmarks for shared-memory parallel machines.

7. Conclusions

Preliminary results for MP3D are promising. Performance of about half that of a Cray–2 has been achieved on an 8-processor Silicon Graphics 4D/380, a machine about 2.5% of the cost of the Cray. At the same time, this performance has been achieved on top of a library designed with potential for other applications in mind.

The real measure of the usefulness of this work will be in implementation of other applications with competitive performance. Although use of SpaceLib will usually require re-implementation of existing applications, the reusable library makes this a less daunting prospect than rewriting from scratch. As the issues addressed by this research become more significant—the growing gap between CPU and memory performance in particular—the cost of such re-implementation is likely to be seen more favourably in relation to benefits. More importantly, programmers developing new applications will have a base on which to develop using modern programming methods.

References

J Barnes and P Hut. A Hierarchical $O(N \log N)$ Force-Calculation Aglorithm, *Nature* vol. 324 no. 4 December 1986 pp 446–449.

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, *Proc. Int. Symp. on Shared Memory Multiprocessing*, Tokyo, April 1991, pp. 109–118.

D R Cheriton, H A Goosen, H Holbrook and P Machanick. Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: The Value of Distributed Synchronization, *Proc. 7th Workshop on Parallel and Distributed Simulation*, San Diego, May 1993, pp. 159–162.

J.L. Hennessy and P. Jouppi, Computer Technology and Architecture: An Evolving Interaction, *IEEE Computer*, vol. 24 no. 9, September, 1991, pp 18–29.

JD McDonald and D Baganoff. Vectorization of a Particle Simulation for Hypersonic Rarefied Flow, *Proc. AIAA Thermophysics, Plasmadynamics and Lasers Conference*, June 1988.

JP Singh, W–D Weber and A Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory, *Computer Architecture News* vol. 20 no. 1 March 1992, pp 5–44.

Acknowledgments

This work was partially supported by David Cheriton with funding from DARPA contract N00014–88–K–0619. Hugh Holbrook did the measurements of MP3D–3 reported here and is working on the message-passing code. Jeff McDonald of NASA-Ames supplied the original version of MP3D and assisted with validating later versions. Henk Goosen who is supervising the PhD which is proposed to result from this work also provided tools for memory system simulation and general encouragement and support.