Parallel Hybrid Particle Simulations Using MPI and OpenMP*

M. Hipp and W. Rosenstiel

Wilhelm-Schickard-Institut für Informatik
Department of Computer Engineering, Universität Tübingen
Sand 13, D-72076 Tübingen, Germany
{hippm,rosen}@informatik.uni-tuebingen.de

Abstract. We present a library for the parallel computation of particle simulations called ParaSPH. It is portable and performs well on a variety of parallel architectures with shared and distributed memory. We give details of the parallelization for hybrid architectures (clustered SMPs) using MPI and OpenMP and discuss implementation issues, performance results and memory consumption of the code on two parallel architectures, a Linux Cluster and a Hitachi SR8000-F1. We show the advantage of hybrid parallelization over pure message-passing especially for large node numbers for which we gain a maximum speedup of about 350 for hybrid parallelization compared to 120 for message-passing.

1 Introduction

The Collaborative Research Center (CRC) 382 works in the field of Computational Physics with a main focus on astrophysical simulations. Our group is responsible for the parallel computing. Some methods used in the CRC 382 are particle based and thus we develop efficient parallel particle libraries to support these applications. [6]

There is always a need for larger simulations needing more memory and computing power. A parallelization combining threads and message passing is a promising way to reduce the parallel overhead in respect of memory and performance on the increasing number of hybrid architectures over pure message-passing parallelization.

2 Hybrid Architectures

For development and performance analysis we are working on two parallel systems. A Hitachi SR8000 installed at the HLRB in Munich and a Linux Cluster installed locally at Tübingen University.

* This project is funded by the DFG within CRC 382: *Verfahren und Algorithmen zur Simulation physikalischer Prozesse auf Höchstleistungsrechnern* (Methods and algorithms to simulate physical processes on supercomputers).

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 189–197, 2004.

The Hitachi SR8000 consists of 8-way SMP nodes coupled by a fast communication network. The installation in Munich has 168 nodes with a total FPU performance of 2 TFlops.

The Linux cluster is built of commodity hardware and has two partitions. One partition consists of 96 2-way SMP Intel Pentium 3 nodes with 650 MHz processor speed and the second partition consists of 32 2-way AMD Athlon nodes with 1667 MHz processor speed. All nodes from both partitions are connected with a switched full-bisection-bandwidth Myrinet network. The peak FPU performance of the Linux cluster is 338 GFlops.

3 Motivation

The majority of parallel machines in the TOP500 lists are so called hybrid parallel architectures, a combination of N-way shared memory nodes with message passing communication between the nodes.

Hybrid parallelization is the combination of a thread based programming model for parallelization on shared memory nodes together with message-passing based parallelization between the nodes. The standard library for message passing is MPI[8] OpenMP[9] has become the standard for thread based programming for scientific applications.

Obviously, the share of common data structures on shared memory nodes can reduce the amount of required memory. More important is the reduction of the communication. Every parallel implementation has a maximum speedup limited by its serial parts. In our non trivial particle codes the major serial part is the (often collective) communication. A hybrid implementation reduces the amount of transferred data because the communication of shared data on the SMP nodes is implicit.

But the communication itself is faster, too. A simple test shows this for the MPI-*Allgather* call. We distribute a 200 MByte data array on the Hitachi SR8000 between all nodes. K is the number of nodes.

In the first test (pure-MPI) a MPI process runs on every SR8000 processor. The $K \times 8$ processors send and receive $200/(K \times 8)$ MByte to/from each *processor*. In the second test (Hybrid) the same amount of data is just sent between the master threads of each node and thus each call sends and receives $200/K$ MByte to/from each *node* (other non-master threads are idle).

The numbers in the table are the time for 50 MPI-*Allgather* calls in seconds on the SR8000-F1 at the HLRB in Munich.

	K=1	K=2	K=4	K=8	K=16	K=32
Pure-MPI	16	51	97	126	172	200
Hybrid	10	15	20	21	22	21

One can see a big difference between the hybrid communication (comparable to the hybrid programming model with implicit intra-node communication) and the *pure-MPI* communication model with explicit MPI intra-node communication. Because the inter-node communication should be independent of the

parallelization strategy for a good implemented *Allgather* call, for $K > 1$ one would expect for $K > 1$ the same 16 to 10 ratio of the $K = 1$ case (showing the time of the intra-node communication and the overhead for 8 times more messages). Instead, for the $K = 32$ run, the hybrid programming model is 8 times faster.

3.1 OpenMP

We chose OpenMP to keep the implementation portable. Compilers for OpenMP are available on all important hybrid platforms including Hitachi SR8000, NEC SX5/6 and Linux (Intel C++ or Portland Group compilers). OpenMP has advantages over explicit thread programming, for example POSIX threads. First, it annotates sequential code with compiler directives (pragmas) allowing an incremental and portable parallelization. Non-OpenMP compilers ignore the directives. Second, POSIX threads require to implement parallel sections in separate functions and functions with more than one argument need wrappers, since the POSIX threads API supports only one argument. OpenMP allows joining and forking threads at arbitrary positions in the source code.

3.2 HPF

A complete different approach is using a HPF compiler such as the Vienna Fortran Compiler, which is able to generate hybrid code. If a code is parallelized with HPF and performs well on a message-passing architecture this can be an interessting option. Since our code-base was written in C and there was a MPI parallelisation, this was no option for us.

4 The SPH Method

An important particle method used in the CRC 382 is Smoothed Particle Hydrodynamics (SPH). SPH is a grid-free numerical Lagrangian method for solving the system of hydrodynamic equations for compressible and viscous fluids. It was introduced in 1977 by Gingold [3] and Lucy[7]. It has become a widely used numerical method for astrophysical simulations. Later, the method was also applied to other problems such as the simulation of a fluid jet to model the primary break-up of a diesel jet as it is injected into the cylinder of an engine[2].

Rather than being solved on a grid, the equations are solved at the positions of the pseudo particles, representing a mass element with certain physical quantities while moving according to the equations of motion. Due to the mesh-less nature of the method, SPH is well suited for free-boundary problems and can handle large density gradients.

Each particle interacts with a limited number of particles in its neighborhood. Because the particle positions and therefore the neighbor interactions change after each time-step one cannot find a perfect load balancing and domain decomposition for the parallelization in advance. Instead, a reasonable fast and

communication optimized domain decomposition, which must be applied after every time-step is crucial. The general parallelization strategies are explained in more detail in [5].

5 Hybrid Implementation

The hybrid implementation is an extension of the ParaSPH library for particle simulations. ParaSPH is written in C and parallelized with MPI. The library separates the parallelization from the physics and numeric code. The interface between the library and the application is optimized for particle simulations. The library provides an iterator concept to step through all particles and their neighbors and later communicates the results.

In parallel mode, the library transparently distributes the work amongst all processors. Every local iterator processes only a subset of all particles. The code performs well on machines with a fast message passing network. We tested the code on Cray T3E, Hitachi SR8000, IBM SP and a Linux cluster.

For the hybrid implementation, OpenMP is used for the inner intra-node parallelization. MPI is still used for inter-node communication. To achieve a better portability, because not all MPI implementations are thread safe, only the master thread calls the MPI library. The performance penalty is small for our applications, since the expensive communication calls are collective operations.

Experiments showed, that it is necessary to optimize the load balancing. Therefore, we introduced two different balancing strategies. The standard load balancer for distributed memory is used only for a coarse load balancing between the nodes. For the fine balancing on the node, the user can choose between two new balancers, a fixed load balancing and a dynamic master-worker load balancing. The master-worker algorithm promises the best load balancing for inhomogeneous problems, because of its inherent load-steeling. The disadvantage is the worse cache utilization, because the data is not bound to a specific CPU for successive runs over the particle list. For SPH simulations with a fixed number of neighbour interactions, the static load balancing is faster. One may consider the dynamic load balancer for computations with a variable number of interactions and large density gradients.

From OpenMP, we used 15 *parallel* pragmas, one *barrier* pragma and one *threadprivate* pragma. We also need two additional locks to protect internal structures.

The first version, ParaSPH frequently called sections with a *critical* region and showed a bad performance on the SR8000. Explicit locking instead of critical regions improved the performance only a little. Lock-free implementations of the static load balancer and iterator fixed the problem. We setup up an independent particle list with its own iterator for each thread and omit shared counters and pointers.

Another problem was a compiler flaw in the Hitachi OpenMP implementation, if the program uses a local constant value like

```
{ /* begin local code section */
   double const aValue = 2.0/3.0;
   ... some code using aValue ...
}
```

The compiler generated a shared variable together with an initialization for every run through the local code section instead of using an immediate value or a register. The cache trashing leads to a two times slower code.

On the Linux platform, we used Intel's C++ compiler. We had to replace the *threadprivate* directives by explicit memory allocation for every thread, because the *threadprivate* directive triggers a compiler bug.

No additional code change was necessary to instrument 95% of the parallel code with OpenMP. But 5% remaining serial code limits the speedup to about 6 on one Hitachi node with its 8 CPUs per node. So, we redesigned parts of the code to increase the parallelization ratio. The strategy was to first identify the hot spots in the remaining 5% and find a lock-free implementation. Most parts need minor changes to omit necessary locks or heavy usage of shared data structures. For the remaining parts with no lock-free alternative, we tried to optimize the serial code itself to reduce the run time. With these changes we gain a parallelization ratio of about 98% compared to the *pure-MPI* version.

6 Results

Our standard application – a typical astrophysical problem of an accretion disk – is a 2-dimensional SPH simulation with 300 000 particles and 80 interaction partners per particle. This medium sized simulation requires about 900 MBytes of memory on one node. One integration step needs about 74 seconds on one Hitachi node (8 processors, *pure-MPI* mode). Large production runs use more particles and are often calculated in three dimensions. They usually need 1 000 or more integration steps resulting in several days of computation time on eight CPUs and weeks on one CPU.

6.1 Hitachi SR8000 Single Node Performance

First we compared the performance of the hybrid and *pure-MPI* code on one SR8000-F1 node. We used the hybrid version for this test, although there is no message-passing. The application was about 25% slower (95 sec. compared to 74 sec.).

We found three main reasons for the performance impact of the hybrid version.

Serial Parts. The code is not fully annotated with OpenMP instructions. There is a small serial part, which is not present in the pure MPI version. This causes a performance decrease especially on machines with a great number of processors per node like the SR8000. In hybrid mode, the serial part is a fixed overhead over *pure-MPI* parallelization. For the SR8000, the remaining 2% serial code in ParaSPH result in a performance decrease of about 14%.

OpenMP Overhead. Similar to the serial overhead is the OpenMP overhead itself (thread creation, locking of critical sections) together with some additional parallelization work (for example the fine load balancing described above). Flaws in the OpenMP compiler may additionally decrease performance and are difficult to find. Usually, the generated assembler code has to be verified after identifying the problem with a profiler.

Cache. The parts computing the physical quantities have a near perfect hybrid parallelization (no serial code) together with a near perfect load balancing (about 99%) But unfortunately, the total computing speed of the physical part is much slower in hybrid mode. When we monitor the number of data load/stores and the data cache-misses we get a near equal number of load/stores but 3 to 4 times higher data cache-misses. The reason is not yet investigated. There is no simple reason, since the intra-node load balancer tries to schedule the same load to the same thread for successive runs over the particle list and concurrent memory access of the same data exists only for reading (causing no cache-trashing). Only data writes from different threads may fall into the same cache line resulting in a higher cache miss rate.

6.2 Linux Cluster Single Node Performance

We used gcc for the *pure-MPI* version and Intel icc for the OpenMP version. The Intel icc is about 4% slower than the gcc in *pure-MPI* mode.

So, we expect only a little difference between the *pure-MPI* and the hybrid version for the Linux Cluster with only two processors per node. The reality was different. The hybrid version is 30% slower on one node than *pure-MPI*.

The reason may be a again a much worse cache utilization discussed in the section above. Additionally, the Linux kernel may frequently reschedule the threads on different processors while MPI processes are better stuck to one CPU.

6.3 Parallel Speedup

Figure 1 shows the speedup comparison of the *pure-MPI* and the hybrid parallelization for different processor numbers. Now the inter-node communication becomes important. On the SR8000 with 8 processors per node the hybrid speedup for large node numbers is much better. There is only little difference between the two strategies on the Linux cluster. "Physics" is the computation of the physical quantities including their communication. It shows a linear hybrid speedup until 256 CPUs. Since it contains communication, there is no linear speedup for larger node numbers. For *pure-MPI*, the "Physics" is the dominant part while for hybrid the parallelization overhead (without communication) is dominant. On the Linux cluster one can see a super-linear speedup for the "Physics" curve. The reason is the increasing cache efficiency. The smaller per-processor problem sizes for large processor numbers result in a better cache reuse. The effect is smaller for *pure-MPI*, since one node cache efficiency is higher.

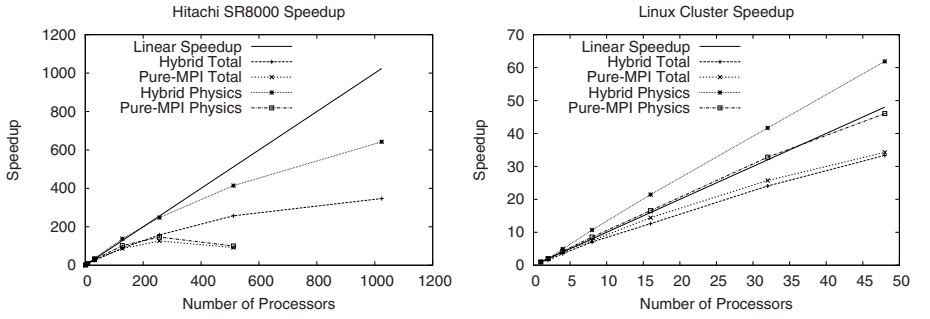

Fig. 1. Speedups using different parallelization strategies on Hitachi SR8000 (left) and the Linux Cluster (right).

The reduced communication is the reason why the hybrid version is faster than the *pure-MPI* version for large node numbers (see 2). The time consuming call in the communication part is a MPI_Allgather and (after a code change) several calls to MPI_Allgatherv together with some post processing to reorganize the received data. In the Allgather-test in section 3 we sent the same amount of data independent of the number of nodes. For the SPH method, the amount of data sent to the neighbors even increase with the number of nodes, because the interaction area between the domains increases compared to the domain size. On the SR8000 the hybrid communication time is near constant for large node numbers, while the *pure-MPI* communication significantly increases for more than 128 processors. On the Linux Cluster, there is only little difference between hybrid and *pure-MPI* parallelization.

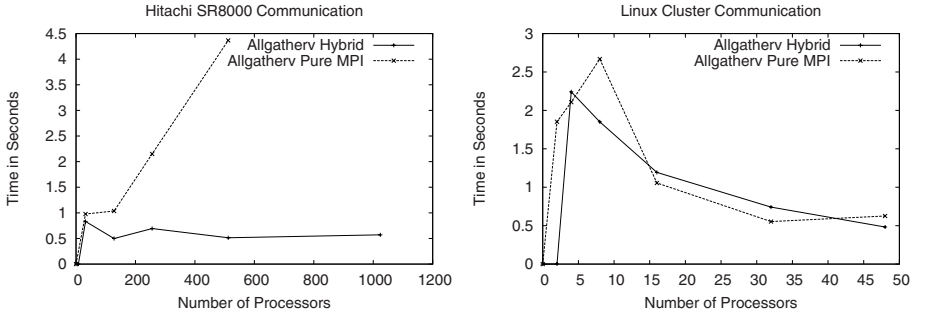

Fig. 2. The communication is the reason for the limited speedup in *pure-MPI* mode for large node numbers.

6.4 Memory Consumption

The SPH method with its neighbour interactions shares a lot of data between processor domains. The advantage of reducing the number of domains for the memory consumption is obviously. While there is only little thread overhead, a *pure-MPI* run on one SR8000 node takes about the same amount of memory as a hybrid job on 8 nodes. The exact reduction depends on several simulation parameters. A good approximation is that we can double our problem size by switching to hybrid mode.

7 Related Works

The comparison of a *pure-MPI* with a hybrid OpenMP/MPI programming model is done in [1] for the NAS benchmark on a IBM SP. To achive a good hybrid implementation the NAS benchmark is profiled to find loops for OpenMP parallelization. Comparing only computation time, the *pure-MPI* version is near always faster. There are only advantages for the hybrid version, if communication time dominates the benchmark.

In [10] there is an extensive comparison of different hybrid programming models. The author compares hybrid-masteronly, hybrid-multiple and *pure-MPI* communication on several parallel platforms. The author concludes, that on some platforms (for example IBM SP) only a hybrid-multiple programming style can achive full inter-node bandwith because a single CPU cannot saturate the network unless the MPI library itself uses a thread based model to optimize the communication.

Henty [4] presents the results of a hybrid implementation of a Discrete Element Modelling (DEM) code. The author made similar experiences, that a scalable hybrid implementation is not achieved without effort. Worse cache utilization and locking in a thread based parallelization reduces overall performance. Therefor, the *pure-MPI* parallelization of the DEM code is better than a hybrid model for a cluster of SMPs. The load balancing advantages of a hybrid code cannot compensate the penalties.

8 Conclusion

A simple instrumentation of a MPI parallelized code with OpenMP usually results in a worse performance, because it is not always possible to find a simple thread based parallelization for a message-passing based code. Since locks are very expensive, at least on the Hitachi SR8000, it is important to have a lock-free implementation. We gain a parallelization rate of about 98% with simple OpenMP instrumentation together with minor code rewrites and explicit optimization of serial parts.

With these enhancements, a hybrid parallelization is possible. It has serveral advantages over *pure-MPI* such as reduced memory consumption and can increased performance for larger node numbers on machines with bigger SMP

nodes such as the 8-way Hitachi SR8000, while there is no performance advantage or even a disadvantage over *pure-MPI* parallelization for small node numbers or machines with only 2-way SMP nodes. The reasons are a worse cache efficiency, thread scheduling problems together with compiler flaws and inefficient implementations of some OpenMP features.

For large node numbers the parallelization profits from the reduced communication overhead in hybrid mode. For the SPH method, one can achieve a speedup of about 257 on 512 CPUs and 346 on 1024 CPUs while the *pure-MPI* version is limited to a speedup of about 120 on 256 CPUs on the Hitachi SR8000.

The future work is to further investigate the reason for the reduced cache utilization of the hybrid parallelization and to provide a hybrid version with good performance for small node numbers, too.

References

1. Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. In *Proc. Supercomputing '00, Dallas, TX*, 2000.
2. S. Ganzenmüller, M. Hipp, S. Kunze, S. Pinkenburg, M. Ritt, W. Rosenstiel, H. Ruder, and C. Schäfer. Efficient and object oriented libraries for particle simulations. In E. Krause, W. Jäger, and M. Resch, editors, *High Performance Computing in Science and Engineering 2003*, pages 441–453. Springer-Verlag, 2003.
3. R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics: Theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181:375–389, 1977.
4. D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Proc. Supercomputing '00, Dallas, TX*, 2000.
5. M. Hipp, S. Kunze, M. Ritt, W. Rosenstiel, and H. Ruder. Fast parallel particle simulations on distributed memory architectures. In E. Krause and W. Jäger, editors, *High Performance Computing in Science and Engineering 2001*, pages 485–499. Springer-Verlag, 2001.
6. S. Hüttemann, M. Hipp, M. Ritt, and W. Rosenstiel. Object oriented concepts for parallel smoothed particle hydrodynamics simulations. In *Proc. of the Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'99)*, 1999.
7. L. B. Lucy. A numerical approach to the testing of the fission hypothesis. *The Astronomical Journal*, 82(12):1013–1024, 1977.
8. Message Passing Interface Forum. MPI: A message passing interface. In *Proc. Supercomputing '93*, pages 878–883. IEEE Computer Society, 1993.
9. OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface, March 2002. `http://www.openmp.org`.
10. Rolf Rabenseifner. Hybrid Parallel Programming on HPC Platforms. In *Proc. European Workshop on OpenMP '03*, 2003.