Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling

D. S. Henty
Edinburgh Parallel Computing Centre, James Clerk Maxwell Building, The King's Buildings,
The University of Edinburgh, Mayfield Road, Edinburgh EH9 3JZ,
Scotland, U.K.
email: d.henty@epcc.ed.ac.uk

Abstract

The current trend in HPC hardware is towards clusters of shared-memory (SMP) compute nodes. For applications developers the major question is how best to program these SMP clusters. To address this we study an algorithm from Discrete Element Modeling, parallelised using both the message-passing and shared-memory models simultaneously ("hybrid" parallelisation). The natural load-balancing methods are different in the two parallel models, the shared-memory method being in principle more efficient for very load-imbalanced problems. It is therefore possible that hybrid parallelism will be beneficial on SMP clusters. We benchmark MPI and OpenMP implementations of the algorithm on MPP, SMP and cluster architectures, and evaluate the effectiveness of hybrid parallelism. Although we observe cases where OpenMP is more efficient than MPI on a single SMP node, we conclude that our current OpenMP implementation is not yet efficient enough for hybrid parallelism to outperform pure message-passing on an SMP cluster.

1 Motivation

For a long time the HPC market was dominated by large, special-purpose MPPs with single-CPU nodes. More recently the shared-memory (SMP) architecture has grown in importance for HPC as advances in technology have allowed increasing numbers of CPUs to have access to a single memory space. In the future it appears that high-end HPC machines will have both MPP and SMP-like features. MPP manufacturers are replacing the single processors in their existing architectures with more powerful SMP nodes, while large SMPs are being clustered to go beyond the architectural limits on the number of CPUs in a single box. The fact that the machines being developed under the US ASCI program are of this type is clear evidence of the trend. Although large SMP clusters are straightforward to construct, the major question for the applications programmer is how to write portable parallel codes that run and scale efficiently.

Codes written in a parallel language like HPF must rely on the compiler for efficiency, whereas those purely using multi-threading will require special OS support in order to see a single system image and memory space across the whole SMP cluster. Although both these approaches are possible, we do not envisage them being widespread in the immediate future and they are not considered here.

Pure message-passing codes should port easily to any SMP cluster. It is necessary to use message-passing to communicate between processors in different boxes, but it is not immediately apparent that it is the most efficient parallelisation technique among processors in the same box. In this paper we examine the effectiveness of using a hybrid of both message-passing and shared-memory techniques within the same application. Although this approach clearly maps most closely to the underlying target architecture it remains to be shown that it is effective in practice.

Situations where this hybrid parallelisation might be of benefit include:

- where the MPI implementation is unoptimised for communications within an SMP
- replicated data algorithms where the hybrid approach might save memory by allowing one data copy per SMP rather than one for each individual CPU
- situations where there are more efficient shared-memory algorithms

Here we consider the final case and take advantage of the fact that load-balancing is more straightforward in shared-memory than in message-passing.

Some work has already been done in this area: a message-passing harness has been used to task-farm independent shared-memory jobs across the nodes of a cluster [1]; shared-memory parallelism has been introduced into a message-passing code in order to reduce the amount of data broadcasting and redistribution [2]; a simple Jacobi algorithm has been studied to evaluate the efficiency of hybrid parallelisation for the IBM teraflop system at SDSC [3]. Where direct comparisons are made between message-passing and hybrid implementations the results are somewhat inconclusive.

The work presented here is novel in that as well as combining the two parallelisation models we are using different load-balancing methods simultaneously in the same code and, unlike in [1], the message-passing tasks are closely coupled rather than being independent. We also determine the overheads associated with each of the parallel models for a general case rather than simply measuring the performance of a particular simulation.

0-7803-9802-5/2000/$10.00 (c) 2000 IEEE.

2 Background

The algorithm originates from a code developed in the Dept. of Physics at the University of Edinburgh [4] used to simulate the interaction of solid particles such as grains of sand. Such models are generically termed Discrete Element Models (DEMs) and are widely used when simpler continuous models are inadequate and a system requires to be simulated at the level of individual particles. Many poorly understood processes such as the way that particles pack together can be investigated using DEMs, where different models are defined by the precise form of the inter-particle force.

For the particular DEM in question the main interest for the physicists is how best to model friction. A typical simulation might involve letting particles fall under gravity onto a solid surface to form "sand-piles". Rather than the standard approach of using smooth particles with complex empirical frictional forces, the simulation uses complex particles with simple forces. These complex particles, or "grains", are collections of simpler basic particles stuck together with permanent bonds made of dissipative springs.

The idea is that the complicated macroscopic laws of friction will arise dynamically from the many microscopic collisions of these rough grains [5]. There is interest in the results for both two and three dimensions. These piles form and grow dynamically, and hence there is an ever-changing spatial distribution of clusters of particles; load-balance is clearly one of the key issues for any parallel implementation.

3 Methodology

There was an initial pilot project to parallelise the Physics DEM using message-passing, but this proved infeasible in the time available. Although parallelisation was straightforward in principle, the very large functionality of the code and the details of the data structures made it difficult in practice. Parallelisation using purely shared-memory directives is currently underway.

The basic aims are to evaluate the relative efficiencies of message-passing, shared-memory and hybrid implementations of this particular DEM algorithm. Rather than tackle the existing Physics DEM application with all its functionality and complexity, we choose to investigate the performance of a much smaller test code that implements precisely the same algorithm but has limited functionality. We simply simulate a collection of identical particles interacting via pairwise forces. Adding additional features would be straightforward but they would have little or no effect on the basic algorithm and its parallelisation, and would merely complicate the study.

The key features to be measured are the overheads of performing a load-balanced simulation of a clustered system using the various parallel models. We want to draw general conclusions, so rather than selecting a single complicated DEM simulation we choose to study a very simple, load-balanced test system. This enables us to predict the overheads that would be incurred in more general clustered systems, only requiring knowledge of the granularity of parallelism that would be required to achieve load-balance in each particular case.

We describe the basic algorithm and its serial implementation in Section 4 and cover the benchmarking procedure in Section 5. The design, performance and optimisation of the message-passing parallel implementation are detailed in Section 6; the shared-memory implementation is covered in Section 7, and we directly compare the performance of the two in Section 8. The motivation for and performance of the mixed-mode implementation are covered in Section 9; we draw conclusions in Section 10 and describe possible further work in Section 11.

4 The Algorithm

The DEM approach is basically the same as standard molecular dynamics [6] where the positions of the particles are evolved in time using many discrete time steps. The basic properties of a DEM are:

- there are many particles
- the inter-particle force is very short-ranged
- the time taken to compute the forces dominates the simulation
- particles are relatively stationary
- there is a potentially large and unpredictable clustering of particles

Our algorithm works by identifying all of the nearby pairs of particles, those closer than some cutoff distance r_c, by creating a list of pairwise links. In a real simulation these might represent either two particles in the same grain connected by permanent bonds, or two particles in different grains that are close enough that they might soon interact via contact forces. Since the particles move relatively slowly, this list only requires to be recalculated at infrequent intervals. The procedure is:

- create links between particles closer than cutoff r_c
- **repeat**
  - calculate forces across all links
  - update particles positions
- **until list is no longer valid**

The algorithm has one variable parameter, the cutoff distance r_c, which can take any value larger than the extent of the longest-ranged force, r_{max}. The larger the value of r_c then the less frequently the list must be recalculated. However, using a very large r_c will create many links between particles that never actually interact.

This is a standard approach in molecular dynamics. The important point for this work is that, because of the way it is implemented, the fundamental object in the code is a single list of links and the major time-consuming loop is over this list rather than over the particles themselves.

4.1 The Code

The serial test code, with which all the performance studies in this paper were performed, was written in Fortran 90 and works in an arbitrary number of dimensions D. The simulation domain is rectangular with either periodic boundary conditions or reflecting hard walls. The main loop in the code is to compute the force F_i on each particle i:

- **DO** $ilink$ **= 1, nlink**
  - compute f_{ij} between i and j at ends of $ilink$
  - update F_i and F_j accordingly using $f_{ji} = -f_{ij}$
- **END DO**

Note that there is a single link between particles i and j which means that the minimal number of force evaluations is carried out (one per distinct pair). The form of the algorithm remains the same, a single loop over a list of pairwise links, regardless of the details of the particular simulation.

When calculating the list of links a standard cell-based approach is used. The simulation domain is divided into a number of cubical cells, slightly larger than r_c in each dimension, and particles placed in their respective cells. When building links for each particle, only the positions of particles in the same cell and the $3^D - 1$ neighbours need to be checked. It is straightforward to avoid double-counting the links, eg by ensuring that links internal to a cell originate from the lowest-numbered particle and those between cells from the lowest-numbered cell.

Having computed F_i, we can then update the particle positions x_i (we use a standard second-order accurate scheme) taking into account the boundary conditions.

5 Benchmarking procedure

Benchmarks are performed on the following platforms

- 344-CPU Cray T3E-900 (450 MHz Alpha EV5.6)
- 8-CPU Sun HPC 3500 (400 MHz UltraSPARC-II)
- cluster of 5 Compaq ES40 SMPs (four 500 MHz Alpha EV6 CPUs per SMP, memory-channel interconnect)

The T3E and Sun are located at EPCC, the Compaq cluster in the University of St. Andrews.

We use simple load-balanced test cases in two and three dimensions. All simulations have a uniform, random distribution of one million identical elastic spheres of diameter $d = 0.05$ in an L^D box. With this choice of force, calculating each pairwise interaction requires one floating point inverse and one square root. As there are only contact interactions, the maximum force range $r_{max} = d$. We perform simulations with cutoff $r_c = 1.5 r_{max}$ and $r_c = 2.0 r_{max}$. For $D = 2$ we choose $L = 50$ and for $D = 3$ set $L = 5$, giving the same particle density in each case.

When reporting performance we use the time t taken per iteration to compute the force and update the positions; we exclude the link generation as this represents a small overhead in a real simulation. Times are averaged over 40 and 20 iterations for $D = 2$ and $D = 3$ respectively. Compiler optimisation was set at a reasonably high level using -fast on the Sun and Compaq, and -O3 on the T3E. Detailed platform-specific compiler tuning was not investigated.

6 Message-passing Parallelisation

The obvious message-passing parallelisation technique is to use MPI and domain decomposition. A general block-cyclic distribution was chosen to enable a clustered simulation to be load-balanced by adjusting the granularity appropriately. The core domain of each block is extended in the standard way to include a halo of width r_c in every dimension, and at each iteration we perform halo swaps with neighbouring processors. In the code, each individual block is effectively treated like a separate simulation with time-varying boundary conditions provided by the halo particles. The only difference from the serial algorithm is that we now have additional links between core and halo particles.

When building the list of links we must now distinguish between links that involve a halo particle and those that do not. Links from core to halo particles are effectively replicated between neighbouring processors and we must avoid double-counting, for example when calculating the total potential energy stored in the links. In practice, the links for each block are placed in a single list with all the core links first. The routine to update the positions is called twice, first with core links and then with halo links, and the energy associated with the latter is multiplied by a half.

For long periods of time while the same set of links remains valid, the template of the halos (ie the sets of boundary particles that must be exchanged) remains constant. For efficiency, we construct MPI indexed data-types for every block which describe the halo data to be sent in each dimension. Halo swaps are achieved by a series of matched send-recv calls between neighbouring blocks; the strided halo is received into contiguous storage immediately following the data for the core particles.

The same MPI types can be used for many iterations until the list of links becomes invalid because some particles have moved too far from their initial positions. At this point, particles that have moved outside the core region are moved to their new home process, the halos are recalculated and swapped, and a new list of links is constructed.

6.1 MPI Performance

We are mainly interested in the scaling of the time $t(P)$ with the number of MPI processes P and not absolute timings, so results on different platforms are normalised to the time taken to run on a small number of processors P_0 (it is not always possible to run on a single processor due to memory constraints on distributed-memory platforms). For completeness we report the relevant base timings in Table 1, scaled by P_0 to give the effective single-processor numbers. Some of the relatively poor performance of the T3E nodes can be ascribed to the fact that default integers occupy eight bytes on the T3E rather than the usual four. Processing the large list of links requires many integers to be read from memory, and the increased integer size places a heavier load on the T3E memory system.

For the parallel runs, the timings were stable and reproducible on all platforms expect the Compaq (CPQ) cluster where we observed variations of up to 5% even on an otherwise unloaded machine. The timings reported are the minimum obtained from at least three independent runs.

6.2 Block distribution

The results are shown in Figure 1 for a simple block distribution. Here we show the speedup obtained on P processors, compared to the reference time on P_0 processors, plotted as a function of P/P_0. These results come from using the smaller cutoff $r_c = 1.5 r_{max}$; with $r_c = 2.0 r_{max}$ the graph is qualitatively similar but the quantitative variation of performance with P is reduced.

Platform	D	r_c/r_{max}	P_0	$P_0 \times t(P_0)$
Sun	2	1.5	1	3.28
	2	2.0	1	4.13
	3	1.5	1	5.68
	3	2.0	1	9.05
T3E	2	1.5	8	3.84
	2	2.0	8	4.97
	3	1.5	8	7.60
	3	2.0	8	12.73
CPQ	2	1.5	1	1.80
	2	2.0	1	2.23
	3	1.5	1	3.20
	3	2.0	1	4.91

Table 1: Time per iteration (seconds) on P_0 processors

Platform	D	r_c/r_{max}	P_0	$P_0 \times t(P_0)$
Sun	2	1.5	1	2.45
	2	2.0	1	3.31
	3	1.5	1	4.58
	3	2.0	1	7.56
T3E	2	1.5	8	2.93
	2	2.0	8	3.90
	3	1.5	8	6.02
	3	2.0	8	10.60
CPQ	2	1.5	1	1.19
	2	2.0	1	1.57
	3	1.5	1	2.19
	3	2.0	1	3.74

Table 2: Time per iteration (seconds) with particle reordering

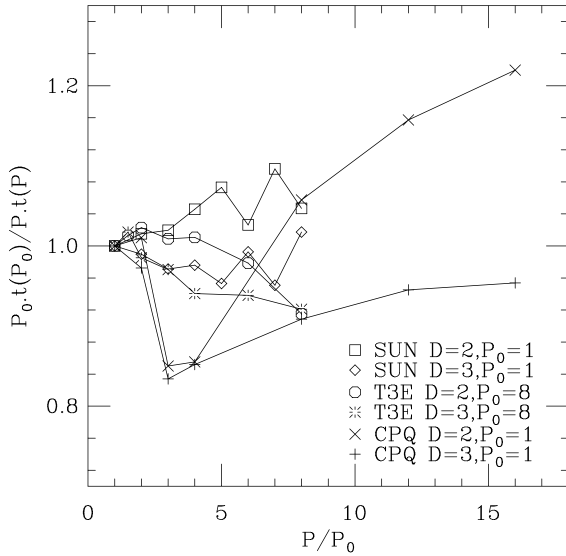

Figure 1: Scaling of performance for MPI block distribution on P processes using $r_c = 1.5r_{max}$

The results show surprisingly good scaling, with efficiencies actually in excess of one often achieved. This effect is typical of poor cache utilisation, with the code benefiting from the larger aggregate cache size as the number of processors is increased. The behaviour on the Compaq cluster is the most complicated. For $P \leq 4$ the runs were done within a single 4-CPU box, whereas for $P > 4$ we ran 4 processes on each box. Performance starts to increase dramatically when we start adding extra boxes, ie extra memory systems, indicating that the code is saturating the bandwidth to main memory on a single SMP.

Although not a key area for performance, the generation of the links scales almost linearly since it can be done independently on each processor once the particles have been redistributed and the halo data exchanged.

6.3 Cache optimisation

Despite the larger parallel overheads with increasing P, we observed in Section 6.2 that the code can show super-linear speedup. This points to poor cache utilisation.

The initial positions of the particles are generated randomly so the data access patterns are random, causing many cache misses. What is required is for particles to be ordered in a way that reflects spatial position. Unlike algorithms with fixed scattered data access (such as finite element simulations) we need to periodically update the ordering, as particles move and transfer between blocks and processes, to prevent the cache locality from degrading with time.

Fortunately, we already generate the required information when constructing the linked lists. When being binned into cells, a list of particle indices is constructed (particles in the same cell being contiguous in the list) along with the number of particles in each cell. We can re-use this same list to order the core particles so that they appear in cell-order within each block (leaving the halo particles untouched). As cells are numbered according to their spatial position, this achieves spatial locality of data. It is a very small overhead to do this each time the links are recalculated, which is precisely when the particles have moved a significant distance from their original positions.

This has a dramatic effect on the performance. The new base times per iteration are shown in Table 2 which show performance increases of up to 30% on the Sun and T3E, and 50% on the Compaq.

The new parallel scaling curve is shown in Figure 2. Although absolute performance is always better than without particle reordering, the parallel efficiencies are reduced as the code benefits much less from the increase in aggregate cache size. The general trend is as expected, with efficiency decreasing with increasing P except for $D = 2$ on the Compaq. Here we still see an increase in efficiency for $P > 4$ when we go outside a single SMP and add more memory systems.

We employ particle reordering for all subsequent simulations unless stated otherwise.

6.4 Load Balancing

To evaluate the efficiency of the block-cyclic distribution we perform the same simulations with finer granularities, ie increased numbers of blocks. To measure the parallel overheads we perform simulations on a fixed, large number of processors with increasing granularity. We would not actually expect performance to increase as there is no load-balance benefit in this particular test simulation where particles are evenly distributed.

In Figure 3 we show the scaling of performance of the

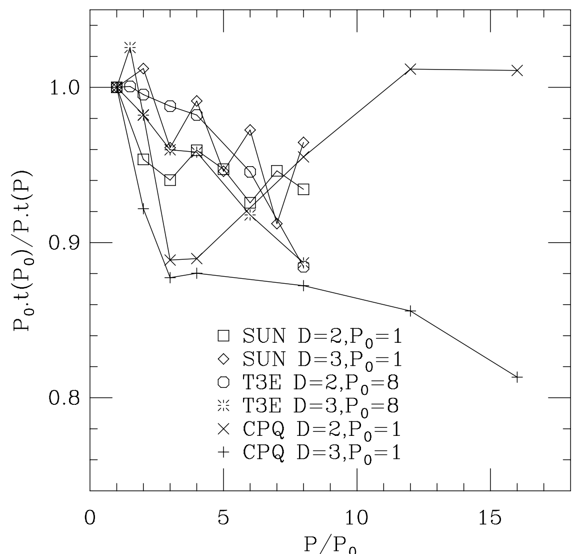

Figure 2: Scaling of MPI block distribution with particle reordering using $r_c = 1.5r_{max}$

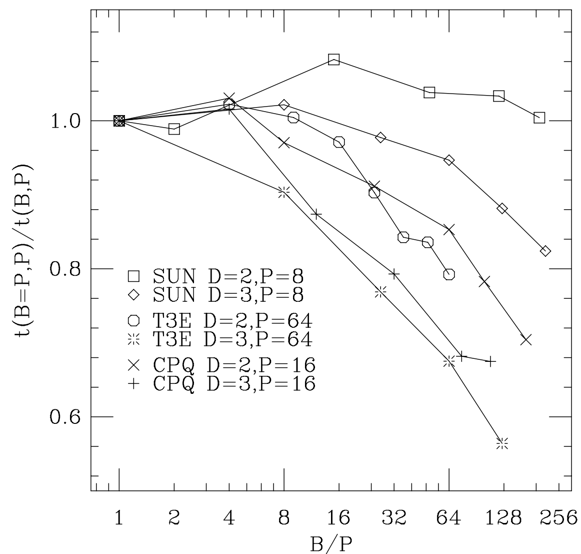

Figure 3: MPI performance vs number of blocks B for $r_c = 1.5r_{max}$

simulation as the number of blocks B is increased with a fixed number of MPI processes P. Results are plotted against the number of blocks per processor B/P, normalised to the timings for the block distribution with $B/P = 1$. We show data for the smaller value of r_c; for the larger cutoff, the results are very similar.

The results generally show the expected trend, with performance decreasing with increasing B, ie for finer-grained parallelism. On the Sun, however, we observe increased performance for $D = 2$. This is again a residual cache effect. Increasing the number of blocks B in the block-cyclic distribution for fixed P decreases the size of each individual block of data. This makes it more likely that the calculation for each block will fit entirely into cache. Without particle reordering, the effect is much more significant; for example, we then observe efficiencies substantially in excess of one on all platforms for $D = 2$.

Figure 3 shows that there can be a significant overhead to load-balancing clustered DEM calculations using a block-cyclic MPI distribution, particularly for $D = 3$. Not surprisingly, the overhead is largest when communication occurs over a real network (the T3E and Compaq) which is precisely the situation on any SMP cluster.

7 Shared-memory Parallelisation

In the shared-memory approach it is straightforward to decompose the problem directly over links as opposed to particle positions. The calculation is therefore automatically load-balanced with a simple block distribution of links amongst threads since the work is tied directly to the links rather than the particles.

The shared-memory parallelisation was done using standard OpenMP compiler directives to parallelise the major loops in the code. On the Sun we used the Kuck and Associates (KAI) Guide system version 3.7, which works by using source-to-source translation and inserting calls to a runtime library. Sun's native OpenMP compiler has just been released but was not available in time for this paper. On the DEC, OpenMP is part of the standard f90 compiler.

The force loop is parallelised over links, the update of positions is parallelised over particles, and link generation over cells. Load balance can be achieved in all cases using a static schedule, a simple block distribution of iterations amongst threads.

The main problem comes when updating the global force array F_i using the individual link forces f_{ij} (see Section 4.1). We need to protect against simultaneous updates of F_i and F_j by different threads. This can be done in several ways including:

- making every update atomic with an OpenMP directive ("atomic" method)
- identifying potential race condition and dealing with them appropriately ("selected atomic")
- accumulating into a thread-specific temporary array then performing a global sum across threads [7] ("array reduction")

The second option was implemented by scanning the linked list as soon as it is generated and identifying those particles with links belonging to more than one thread. This lookup table is consulted whenever the force array is updated, and the accumulation is protected by an atomic lock only if necessary. The table is valid for many force calculations until the linked list is next recalculated. Since there are relatively few multiple updates due to the short-ranged nature of the DEM forces, most of the accumulations do not in fact require protection. This approach minimises the overheads of locking, which can be significant on some platforms.

Several methods were used in implementing the array-reduction approach including:

- performing the global reduction in a critical region
- striping the global sum across threads so that each thread is always updating a different portion of the global array ("stripe method")

- having a global temporary array with an extra thread index. Each thread first accumulates into its own section of the array; the global reduction is then done in parallel over the main particle index ("transpose method").

The generation of the links was also parallelised. This entails parallel loops over particles (when binning into cells) and over cells (when generating the links). Both phases have inter-thread dependencies which were resolved using simple array-reduction methods. The link generation scales rather poorly, but as it is not particularly time-critical we did not spend time optimising it.

7.1 Performance

We measure the time per iteration with the various methods for updating F described above. The results for $D = 3$ are shown in Figure 4 for the Sun and in Figure 5 for the Compaq. Performing array reductions in a critical region gave extremely poor results which are not shown. The stripe and transpose methods gave almost identical performance in all cases so only one set of numbers is plotted. The results for $D = 2$ are very similar to those shown for $D = 3$.

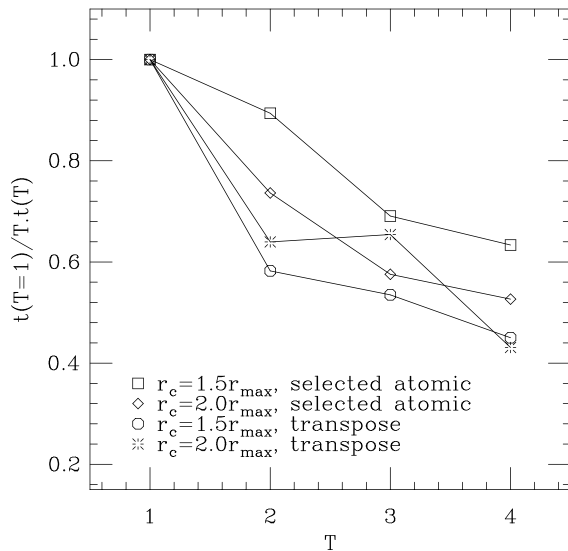

Figure 4: Scaling of performance with number of threads T for OpenMP code on the Sun, $D = 3$

The OpenMP code scales relatively poorly on the Sun, so we do not show results past $T = 4$. The Kuck compiler uses software locks for atomic updates which are very costly even when the minimum number possible is used via the selected atomic method. We do not show the performance for locking every update as this slows the calculation down by around an order of magnitude on four threads. The transpose method does not scale particularly well either. All array reduction techniques place a heavy demand on the memory system, and it appears that this is saturating the available bandwidth.

The UltraSPARC chip does possess an atomic update instruction; however, it is not accessible to the KAI source-to-source compiler. If Sun's native OpenMP compiler implements reasonably efficient atomic updates then scalable

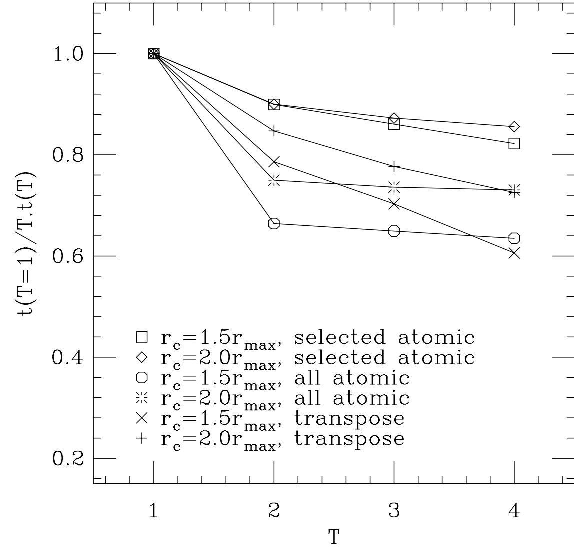

Figure 5: Scaling of performance with number of threads T for OpenMP code on the Compaq, $D = 3$

performance should be achievable using the selected atomic method.

The results are much more encouraging for the Compaq nodes. Here, the atomic updates are done in hardware and are much more efficient. Even so, they do have an associated cost and locking every update is slower than the transpose method on less than four threads. The selected atomic method is clearly the best with parallel efficiencies in excess of 80% on four threads.

The success of the selected atomic approach is an interesting result. An array reduction directive is planned in OpenMP 2.0 and the natural approach for an applications programmer might be to use it in loops such as the one considered here. A compiler implementation, however, could not currently use the efficient selected atomic technique. The approach is effective because the data access pattern is constant for a long time, but a compiler has no way of knowing this. We are currently proposing extensions to OpenMP for this case [8], similar to the SCHEDULE/REUSE extensions proposed for HPF [9].

Particle reordering (used above) has a significant effect on the performance of the OpenMP code. For $D = 3$ on the Sun the improvement was between 15% and 20%, whereas on the Compaq the figures were between 45% and 65%. In contrast to MPI where the increased performance is a purely serial effect and therefore decreases the overall parallel efficiency, the parallel scaling is also enhanced. On the Compaq, efficiencies for $T = 4$ are increased by between 5% and 10%. This is because data locality reduces the contention for cache lines between threads as well as making more efficient use of the cache within each thread.

8 Performance Comparison of MPI and OpenMP

To compare MPI and OpenMP directly we must time identical simulations using the two implementations on the same number of processors. There are no load balancing issues with OpenMP so the simulation time depends only on the

number of threads T. For MPI however, we would have to use a finer-grained block-cyclic decomposition (ie increase the number of blocks B) to load-balance a real simulation. By running our load-balanced test code we can determine the parallel overheads associated with running at each value of B. Since the performance typically decreases with increasing B, we can look for a crossover point where OpenMP outperforms MPI. This tells us that if a real simulation required more than this number of blocks to achieve acceptable load-balance in MPI, then the OpenMP implementation would be more efficient.

On the Sun, the OpenMP scaling is relatively poor and MPI is always faster than MPI for all reasonable values of B. On the Compaq, we choose to run on all four processors of a single SMP. We observe no crossover for $D = 2$, but for $D = 3$ there is a crossover at both values of the cutoff. We show the results in Figure 6.

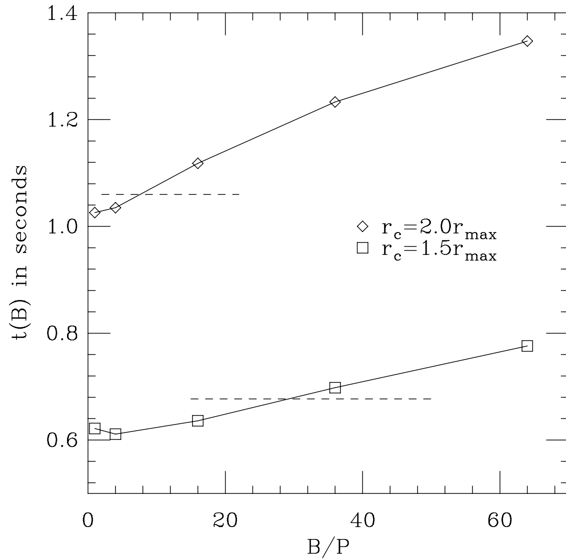

Figure 6: $P = 4$ MPI time vs B compared to OpenMP with $T = 4$ (dashed lines). Results from Compaq with $D = 3$

From the plots we conclude that, for $D = 3$, the OpenMP implementation can be more efficient on four Compaq processors. This occurs if load-balancing a real simulation requires more than 8 blocks per processor with $r_c = 2.0r_{max}$, or 30 blocks per processor with $r_c = 1.5r_{max}$.

9 Hybrid Parallelisation

It is relatively straightforward to have both MPI and OpenMP within the same code. The domain decomposition gives each MPI process a set of blocks with accompanying halos. The OpenMP parallelisation occurs lower down at the level of loops over the links or particles within each block, so MPI communications never take place within a parallel region. Global quantities such as the energy are reduced in parallel within a block by OpenMP, summed over blocks by the MPI process (the master thread) and then accumulated over processes by an MPI collective call.

All communications had already been isolated into a separate library to allow easy porting to message-passing systems other than MPI. It was therefore trivial to provide a dummy library so that a pure OpenMP code could be compiled in the absence of MPI (at runtime the communications routines are actually only called when $P > 1$ for efficiency). A single set of source files can therefore be compiled to produce efficient serial, OpenMP, MPI and hybrid codes.

9.1 Benefits of the hybrid approach

Analysing the performance of the hybrid code is more complicated than for either the pure MPI or OpenMP codes. Load-balancing between the SMPs in a cluster always requires the use of a block-cyclic MPI distribution regardless of whether or not OpenMP is used to load-balance within an SMP. In the hybrid scheme, the load balance characteristics are purely dependent on the MPI distribution and not on OpenMP. The load balance is therefore characterised by the number of blocks per MPI process, B/P, rather than the number of blocks per CPU (each SMP effectively behaves like a single CPU with increased performance).

For a pure MPI calculation it is necessary to achieve equal work for each individual CPU, which may require a very fine granularity and hence substantial performance penalty. However, if we only use MPI between nodes (one process per SMP) and parallelise using OpenMP within each SMP then the calculation is automatically load-balanced between CPUs within the same box. We still need a block-cyclic MPI distribution to balance the load between SMPs. However, global load balance may be achieved in the hybrid model with larger block sizes.

Imagine we are running a relatively coarse-grained MPI simulation that suffers from significant load imbalance. Is it more efficient to improve load balance (i.e. increase B/P) by using MPI with finer granularity (ie increase B), or to use OpenMP to load balance across CPUs within the same SMP (ie decrease P)? If our OpenMP parallelisation was 100% efficient then, since the MPI parallel overheads increase with B, the hybrid approach would always be more efficient. The question is whether our OpenMP parallelisation is efficient enough to take advantage of this effect in practice.

9.2 Performance

Tests were performed on the Compaq SMP cluster. We run pure MPI with $P = 16$, and the hybrid scheme with $P = 4$ (one process per SMP) and $T = 4$ (one thread per CPU). Runs were performed using a range of granularities, with the aim of looking for a performance crossover as already observed in Section 8. The fact that no crossover was previously observed for $D = 2$ does not in principle mean that the hybrid scheme will not be effective. Here we are running MPI across a network and the parallel overheads associated with increasing B will be greater than when running on a single SMP. The results are shown in Figures 7 and 8, plotted as a function of B/P which is the appropriate measure of granularity.

The results are somewhat disappointing, showing that the pure MPI code is always more efficient for a given granularity. Although the efficiency of the hybrid code is better for the larger cutoff, it still falls well short of the required performance for all B.

For all cases with $D = 2$ the hybrid code is significantly slower than the MPI code. For $D = 3$ and $r_c = 2.0r_{max}$, however, the situation starts off promisingly. The hybrid performance for $B/P = 1$ is very close to that of MPI, particularly for the larger cutoff. However, although the efficiency

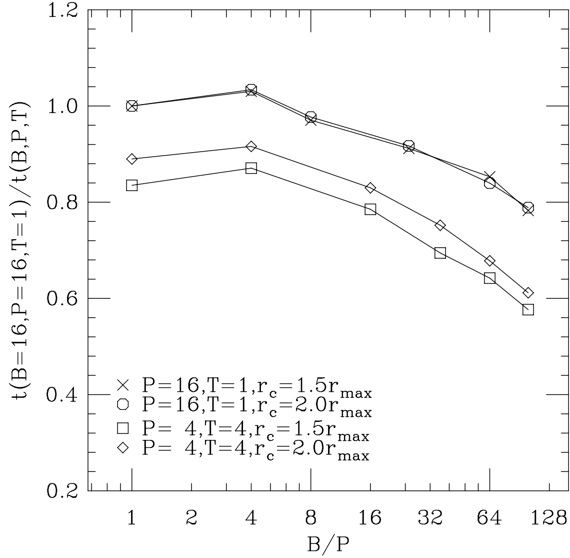

Figure 7: Efficiency of $D = 2$ MPI and hybrid models versus granularity B/P, normalised to MPI with $B/P = 1$

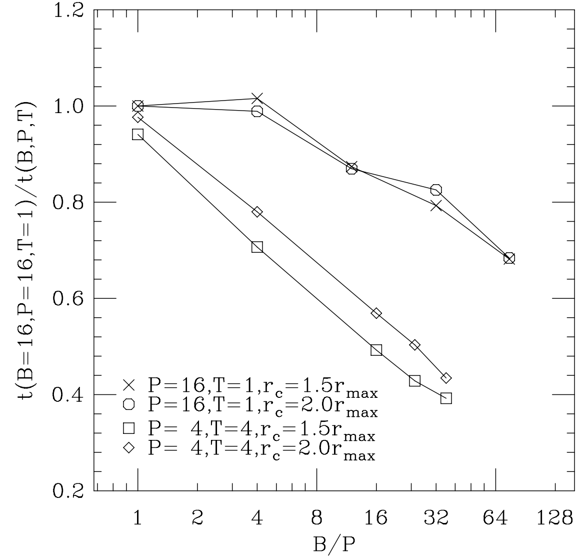

Figure 8: Efficiency of $D = 3$ MPI and hybrid models versus granularity B/P, normalised to MPI with $B/P = 1$

of the pure MPI code decreases with increasing B as seen previously, the efficiency of the hybrid code decreases even more rapidly. The degradation of performance with increasing B is actually worse in $D = 3$ than $D = 2$. Increasing B explicitly increases the MPI parallel overheads due to the finer granularity. However, it is clear that there is an even larger increase in the overheads of the OpenMP implementation when processing many small blocks rather than a few large ones.

9.3 OpenMP overheads

The previous OpenMP tests were done with $B = 1$. By using the selected atomic update method for the force calculation we were able to achieve reasonable parallel efficiency up to $T = 4$. In the hybrid code with $B > 1$ the force calculation and position update routines are called multiple times, once for each block. The possible reasons for the poor OpenMP performance are:

- thread creation and synchronisation overheads
- poor performance of the force calculation

The OpenMP parallelisation was done in a very straightforward fashion with most of the major loops over links or particles parallelised with simple `PARALLEL DO` directives. For each block, this causes thread creation at the beginning of the loop and synchronisation at the end. Hence this overhead will grow linearly with B. Some optimisations were performed for the simplest loops (eg those copying arrays by looping over particles) by having a single parallel region enclosing the outer loop over blocks and removing the unnecessary synchronisation at the end of each loop over particles. However, this is more problematic for the force calculation; for each block, all threads must now complete the parallel loop over links before positions can be updated using a parallel loop over particles.

For this code, parallel loop overheads are not the major cause of the poor performance. For example, the observed overheads scale sub-linearly with B rather than being proportional to B. We can estimate the overheads by counting the number and type of synchronisations and using the OpenMP Microbenchmark Suite [10] to measure the cost of each operation. Doing this for the DEM code gives a result of around 50 microseconds per block per processor. With a maximum value of B/P of 32, this amounts to a couple of milliseconds per iteration. The simple block MPI loop times, used to normalise Figures 7 and 8, are all well in excess of 100 milliseconds so this would only make a difference of a couple of percent. In addition, the $D = 2$ calculation is faster per iteration than $D = 3$ but has the same amount of thread synchronisation, so the relative effect should be larger. In reality, the $D = 2$ hybrid calculation scales better with B than $D = 3$.

The major source of poor performance for the hybrid code comes from the force calculation. Although we were able to achieve reasonable scaling with T for a single large block, the hybrid code uses thread parallelism over many smaller blocks. This means there are fewer particles in each block, and therefore more potential inter-thread conflicts when updating the force. We see a steep increase with B in the total number of atomic locks required during the force calculation, rising to around 50% at the finest granularity for $D = 3$. For $D = 2$, however, the maximum is around 25% which explains the better scaling with B. If an incorrect code is run that omits to lock the force updates (simulating a machine with an extremely efficient atomic lock), we actually observe superior performance of the hybrid code over MPI for $D = 3$ and small B ($B/P <= 2$). However, even without locking there is still a significant performance penalty for larger B. This is due to the large cache coherency overheads from multiple threads altering the same cache lines during the force calculation.

10 Conclusions

We have parallelised an algorithm from Discrete Element Modelling (DEM) using message-passing (MPI), shared-memory (OpenMP) and hybrid models. We load-balance with a block-cyclic spatial domain decomposition in MPI and a block distribution of links over threads in OpenMP. Some effort was required to achieve a scalable OpenMP code, good scaling only being obtained when actual inter-thread dependencies are determined in advance rather than synchronising for each of the large number of potential dependencies. Good cache utilisation is, not surprisingly, critical to performance. Running on a small SMP cluster we demonstrated that, on a single SMP node, OpenMP will outperform MPI for sufficiently clustered simulations in three dimensions. However, on an SMP cluster, the overheads of our OpenMP implementation mean that it is currently always more efficient to use MPI than the hybrid model. The fine granularity of the thread parallelism required in the hybrid model results in much poorer performance than observed in the pure OpenMP code. Overall load balance is better achieved by using a finer granularity of MPI distribution rather than load-balancing within each SMP using OpenMP.

11 Further Work

We are currently making detailed profiles of the hybrid code to quantify the OpenMP overheads for the case of multiple blocks. To this end we are making use of the OMPItrace and Paraver tools from CEPBA [11] to produce and analyse accurate traces of performance. We also plan to reduce the OpenMP overheads in the hybrid code by having a single parallel loop over all links in all blocks rather than one loop per block. This will have the desired effect of reducing inter-thread dependencies, but requires a significant reorganisation of the data structures.

Acknowledgements

This research was partially supported by the Improving the Human Potential Programme, Access to Research Infrastructures, under contract HPRI-1999-CT-00071 “Access to CESCA and CEPBA Large Scale Facilities” established between The European Community and CESCA-CEPBA. I would like to thank Josep Lluis Larriba for arranging my visit to CEPBA, and Jesus Labarta and Eduarde Ayguade for their technical assistance during my stay. I also acknowledge the use of the PPARC-funded Compaq MHD Cluster in St. Andrews, and would like to thank Tony Arber and Keith Bennett for arranging exclusive access for benchmarking.

References

[1] S.W. Bova, C. Breshears, C. Cuicchi, Z. Demirbilek and H.A. Gabb. *Dual-level Parallel Analysis of Harbor Wave Response Using MPI and OpenMP*, The International Journal of High Performance Computing Applications, to appear Spring 2000.

[2] L.A. Smith and P. Kent, *Development and Performance of a Mixed OpenMP/MPI Quantum Monte Carlo Code*, Proceedings of the First European Workshop on OpenMP, Lund, Sweden, Sept. 1999, pp. 6-9.

[3] *Hybrid MPI/OpenMP programming for the SDSC teraflop system*, Online Volume III Issue 14 - July 7, 1999, available at `http://www.npaci.edu/online/v3.14/SCAN.html`.

[4] J.P.J. Wittmer and M.E. Cates, Dept. of Physics and Astronomy, University of Edinburgh, work in progress.

[5] T. Poschel and V. Buchholtz, *Molecular-Dynamics of Arbitrarily-Shaped Granular Particles*, Journal de Physique I, 1995, Vol.5, No.11, pp.1431-1455

[6] D.M. Beazley and P.S. Lomdahl, *Message-Passing Multi-Cell Molecular Dynamics on the Connection Machine 5*, Parall. Comp. 20 (1994) 173-195.

[7] D. Hisley, G. Agrawal, P. Satya-narayana and L. Pollock, *Porting and Performance Evaluation of Irregular Codes using OpenMP*, Proceedings of the First European Workshop on OpenMP, Lund, Sweden, Sept. 1999, pp. 47-59

[8] E. Ayguade and D.S. Henty, submitted to the Second European Workshop on OpenMP, Edinburgh, UK, Sept. 2000.

[9] S. Benkner, *HPF+ High Performance Fortran for Advanced Industrial Applications*, Technical Report TR 98-04, Institute for Software Technology and Parallel Systems, University of Vienna, May 1998.

[10] J.M. Bull, *Measuring Synchronisation and Scheduling Overheads in OpenMP*, Proceedings of the First European Workshop on OpenMP, Lund, Sweden, Sept. 1999, pp. 199-105.

[11] J. Labarta, S. Girona, V. Pillet, T. Cortes and L. Gregoris, *DiP: A Parallel Program Development Environment*, 2nd International EuroPar Conference (EuroPar 96), Lyon (France), August 1996.