Hybrid Message-Passing and Shared-Memory Programming in a Molecular Dynamics Application on Multicore Clusters

Martin J. Chorley*[1], David W. Walker[1], and Martyn F. Guest[2]

[1]School of Computer Science, Cardiff University, Cardiff, UK
[2]ARCCA, Cardiff University, Cardiff, UK

April 6, 2009

Abstract

Hybrid programming, whereby shared memory and message passing programming techniques are combined within a single parallel application, has often been discussed as a method for increasing code performance on clusters of symmetric multiprocessors (SMPs). This paper examines whether the hybrid model brings any performance benefits for clusters based on multicore processors. A molecular dynamics application has been parallelized using both MPI and hybrid MPI/OpenMP programming models. The performance of this application has been examined on two high-end multicore clusters using both Infiniband and Gigabit Ethernet interconnects. The hybrid model has been found to perform well on the higher latency Gigabit Ethernet connection, but offers no performance benefit on low latency Infiniband interconnects. The changes in performance are attributed to the differing communication profiles of the hybrid and MPI codes.

1 Introduction

High performance computing (HPC) is a fast changing area, where technologies and architectures are constantly advancing [1]. In recent years a significant trend within the HPC community has been away from large Massively Parallel Processing (MPP) machines, based on proprietary chip technology and software, towards clusters of standard PCs or workstations using off-the-shelf components and open source software. The November 2008 Top500 ranking (http://www.top500.org) of the world's supercomputers shows that 410 are classed as having a cluster architecture, whereas the November 2002 list shows just 81 clusters in the Top500. The increased availability and decreased cost of commodity-off-the-shelf (COTS) hardware and software means that clusters

*m.j.chorley@cs.cf.ac.uk

are also gaining in popularity in smaller computing centres and research departments as well as the larger centres that tend to feature in the Top500 list.

Recent advances in the computing field, such as 64-bit architectures, multicore and multi-threading processors [2] and software multi-threading [3], can all be used to benefit the Scientific Computing community. Multicore processors can increase application performance above the increase from multi-processor systems [4]. The introduction of multicore processors has enabled a significant trend towards clusters with many thousands of processing cores. The machines in the upper reaches of the Top500 list now contain not just tens of thousands but hundreds of thousands of processor cores per system. As multicore chips become more widespread there is a growing need to understand how to efficiently harness the power available.

These technological advances are leading to the dividing lines between HPC architectures becoming blurred. Previously, HPC machines were generally divisible into two classes: systems of distributed memory nodes in which each node is a processor with its own distinct memory, and systems in which nodes access a single shared memory. When the nodes are identical processors, the latter type of system is often termed a symmetric multiprocessor (SMP). Systems in which the nodes themselves are identical SMP systems introduce another level of complication to the field. The introduction of multicore processors has further increased the complexity of HPC architectures. As the number of processing cores within a processor increases, individual nodes within a distributed memory cluster have become more like SMP machines, with large amounts of memory shared between multiple processing cores, while the overall cluster still retains the global characteristics of a distributed memory machine. This results in HPC cluster architectures being more complicated, containing several levels of memory hierarchy across and within the cluster. In a multicore cluster parallelism exists at the global level between the nodes of the cluster, and across the multiple processors and cores within a single node. At the global level the memory of the cluster is seen as being distributed across many nodes. At the node level, the memory is shared between the processors and processor cores that are part of the same node. At the chip level, cache memory may be shared between some processing cores within a chip, but not others. This complex hierarchy of processing elements and memory sharing presents challenges to application developers that need to be overcome if the full processing power of the cluster is to be harnessed to achieve high efficiency.

The distinction between SMP and multicore systems is an important one. In an SMP system, a number of distinct physical processors access shared memory through a shared bus. Each processor has its own caches and connection to the bus. Multicore processors have several processing cores on the same physical chip. These processors will often share cache at some level (L2 or L3), while having their own separate L1 caches. The cores often also share a connection to the bus. This cache sharing can have performance implications for parallel codes [5].

1.1 Hybrid Parallel Programming

MPI[6] has become the *de facto* standard for message-passing parallel programming, offering a standard library interface that promotes portability of parallel code whilst allowing vendors to optimize the communication code to suit par-

ticular hardware. Shared memory programming has often been accomplished using libraries (such as pThreads). These low-level approaches offer a fine level of control over the parallelism of an application, but can be complex to program. More recently the higher level compiler directive approach taken by OpenMP [7] has become the preferred standard for shared memory programming. OpenMP offers a simple yet powerful method of specifying work sharing between threads, leaving much of the low-level parallelization to the compiler.

The hybrid shared-memory and message-passing programming model is often discussed as a method for achieving better application performance on clusters of shared memory systems [8, 9, 10]. Combining OpenMP and MPI (and therefore the shared-memory and message-passing models) is a logical step to make when moving from clusters of distributed memory systems to clusters of shared memory systems. However previous work has not reached a consensus as to it's effectiveness.

Although it is possible to classify hybrid codes to a fine level of detail according to the placements of MPI instructions and shared memory threaded regions [9], hybrid codes are most easily classified into two simple distinct styles. The first style adds fine-grained OpenMP shared memory parallelization on top of an MPI message-passing code, often at the level of main work loops. This approach allows the shared-memory code to provide an extra level of parallelization around the main work loops of the application in a hierarchical fashion that most closely matches the underlying hierarchical parallelism of a cluster of SMP or multicore nodes. This is the approach we use in this paper. The second style uses a flat SPMD approach, spawning OpenMP threads at the beginning of the application at the same time as (or close to) the spawning of MPI processes, providing a coarser granularity of parallelism at the thread level with only one level of domain decomposition. This approach has not been used in this paper.

1.2 Previous Work

Previous work has been done in considering the hybrid model of parallel programming, and in combining MPI and OpenMP in particular. Cappello and Etiemble have compared a hybrid MPI/OpenMP version of the NAS benchmarks with the pure MPI versions [8], and found that performance depends on several parameters such as memory access patterns and hardware performance. Henty considers the specific case of a Discrete Element Modelling code in [11], finding that the OpenMP overheads result in the pure MPI code outperforming the hybrid code, and that the fine-grain parallelism required by the hybrid model results in poorer performance than in a pure OpenMP code. In [10], Smith and Bull find that in certain situations the hybrid model can offer better performance than pure MPI codes, but that it is not ideal for all applications. Lusk and Chan have examined the interactions between MPI processes and OpenMP threads in [12], and illustrate a tool that may be used to examine the operation of a hybrid application. Jost et al. also look at one of the NAS parallel benchmarks [13], finding that the hybrid model has benefits on slower connection fabrics. One of the best known examples of a hybrid MPI/OpenMP code is the plane wave Car Parrinello code, CPMD [14]. The code has been extensively used in the study of material properties, and has been parallelised in a hybrid fashion based on a distributed-memory coarse-grain algorithm with the addition of loop level parallelism using OpenMP compiler directives and

multi-threaded libraries (BLAS and FFT). Good performance of the code has been achieved on distributed computers with shared memory nodes and several thousands of CPUs [15, 16]

1.3 Contribution

The work presented in this paper investigates the performance of a hybrid molecular dynamics application, and compares the performance of the code to the same code parallelized using pure MPI. In this paper, we consider and examine the performance of both versions of the code on two high-end multicore systems. Much of the previous work on the hybrid message-passing/shared-memory model focuses on SMP systems or clusters of SMP systems. While multicore systems share many characteristics with SMP systems, there are significant differences as discussed in Section 1 that may affect code performance. These differences make the study of the hybrid model on multicore systems a novel direction. The results from the hybrid molecular dynamics application used in this paper provide further knowledge to the discussion of the hybrid programming model and its suitability for use on multicore clusters. As well as considering the hybrid model on multicore clusters, this paper also examines the effect that the choice of interconnection fabric has on the performance of a pure MPI code compared with a hybrid MPI/OpenMP code, examining both high-end HPC Infiniband and Infinipath interconnects, and more standard Gigabit Ethernet connections.

The rest of this paper is organized as follows: Section 2 describes the hybrid Molecular Dynamics (MD) application used in this work, with focus on the portions of code affected by using the hybrid model. Section 3 describes the hardware used for performance testing, and describes the methodology used, while Sections 4 and 5 present performance results. Conclusions from the work are presented in Section 6, and future work is considered in Section 7.

2 Hybrid Application

The molecular dynamics application used in this research simulates a three-dimensional fluid using a shifted Lennard-Jones potential to model the short-range interactions between particles. This simulation is carried out in a periodic three-dimensional space, which is divided into sub-cells each containing a number of particles stored in a linked list.

The MPI message-passing version of the code performs a three-dimensional domain decomposition of the space to be simulated and distributes a block of sub-cells to each MPI process. Each MPI process is responsible for the same set of sub-cells throughout the simulation, although particles may migrate from one cell to another. Although no load balancing is carried out as the simulation progresses, load imbalance is not a significant concern because at short distances particle repel each other, but at longer distances they attract, and hence particles are quite homogeneously distributed in space.

The application contains several routines that are called during each simulation time step. The `forces` routine contains the main work loop of the application, which loops through the sub-cells in each process and updates the forces for each particle. The `movout` routine contains the communication code

used for halo-swapping and particle migration. There are also several smaller routines, such as `sum`, which contains collective communications for summing macroscopic quantities (the virial and the kinetic and potential energies of the system), the `hloop` routine which checks to see if the code is still equilibrating, and the `movea` and `moveb` routines which each perform part of the velocity Verlet algorithm.

2.1 Hybrid Version

The hybrid version of the Molecular Dynamics code used in this paper was created by adding OpenMP parallelization on top of the original MPI parallel code. Parallel directives were added around the main `forces` loop of the application, with the main particle arrays being shared between threads. Threads are spawned at the beginning of the `forces` routine, and the iterations over the sub-cells in the main loop are divided between the threads using the default OpenMP scheduling. A *reduction* clause is used to manage the global sums performed to evaluate the virial and the potential and kinetic energies, rather than using atomic updates within the loop, as these variables are written to by each of the threads during the `forces` update loop. Experimentation with the OpenMP scheduling options for the main forces loop did not reveal any significant performance difference between schedules, thus the default *static* schedule was used for performance tests.

In addition to the `forces` loop, OpenMP parallelization has been applied to the loops over all particles in both the `movea` and `moveb` routine. In all other routines the hybrid code will be running with less overall parallelization than the MPI code, so a decrease in performance of the routines without any form of OpenMP parallelization may be seen.

No requirements are placed on the MPI implementation as to the level of threading support required. Although many implementations of the MPI standard provide a level of thread-safe operation, this must usually be requested at the MPI initialization stage. The thread-safe MPI implementation then guarantees that communication may be carried out in any thread without side effects. The hybrid code used in this work restricts all message-passing communication to the master thread of the application as all such communication occurs outside of OpenMP parallelized regions. This allows the code to be run without multiple threads using MPI communications, thus no level of thread safety is required from the MPI implementation.

2.2 Communication Routines

One expectation of hybrid codes is that they may exhibit smaller runtimes as shared memory communication between threads may be faster than message passing between processes. In this work we have not found this to be a factor in the performance differences. The MPI implementations used for performance testing on the systems include communication devices that are able to use shared memory for intra-node communication. This is a common feature of recent MPI implementations, which may result in few differences between shared memory and message passing code when running on one node. However, the hybrid code does have a different communication profile from the pure MPI code which

results in different timings for the two codes when inter-node communication is involved.

The MD code has two distinct phases of communication. The first occurs in the `movout` routine, where six sets of point-to-point messages are used to send boundary data and migrating particles to the eight nearest neighbour processes. The second phase occurs in the `sum` routine, where collective communications are used to communicate global system quantities (the kinetic and potential energies and virial) to each process.

The first communication phase is clearly affected by the number of processes and the number of cells in each process. A pure MPI code running on a cluster of multicore nodes will have many processes with small numbers of cells. The communication phase for these processes will involve smaller message sizes, but more messages. A hybrid code running on the same cluster will have fewer processes, with larger numbers of cells. The communication phases for these processes will therefore involve larger message sizes, but fewer messages. This difference in communication profile results in the communication times depending heavily on the interconnect used. There is a relationship between the number of messages and the interconnect latency, and the message size and the bandwidth.

Studying this first communication phase in terms of the amount of communication allows us to see the difference between the Hybrid and pure MPI communication profiles. Assuming that the particles are distributed evenly between subcells within a process, we calculate the number of particles to be communicated per process and the number that must be communicated over all processes per timestep (Fig. 1). The Hybrid code clearly has more particles to communicate per process, but as the numbers of processes are reduced compared to the MPI code, the overall number of particles to be communicated is far less. This simple analysis does not take process placement into account and the fact that some of the MPI code communication will be intra-node rather than all inter-node as in the Hybrid case, but does provide an insight into the differing communication profiles.

The second communication phase in the MD code involves collective communications to sum global system characteristics. The size of messages does not change between a pure MPI and a hybrid MPI/OpenMP code, however, collective communications are affected by changes in the numbers of processes involved in the communication, as most collective operations take longer as the number of processes involved increases. The hybrid code uses less MPI processes to run on the same number of cores as the MPI code, so performs better in this communication phase.

3 Performance Testing

Performance analysis of both the Hybrid and pure MPI versions of the MD application was carried out on two systems, Merlin and CSEEM64T. Both are clusters of multicore nodes each with two available connection fabrics: a high-speed, low-latency 'typical' HPC interconnect, and a Gigabit Ethernet network. Merlin is a production HPC system in use at Cardiff University, while CSEEM64T is a benchmarking system at Daresbury Laboratory.

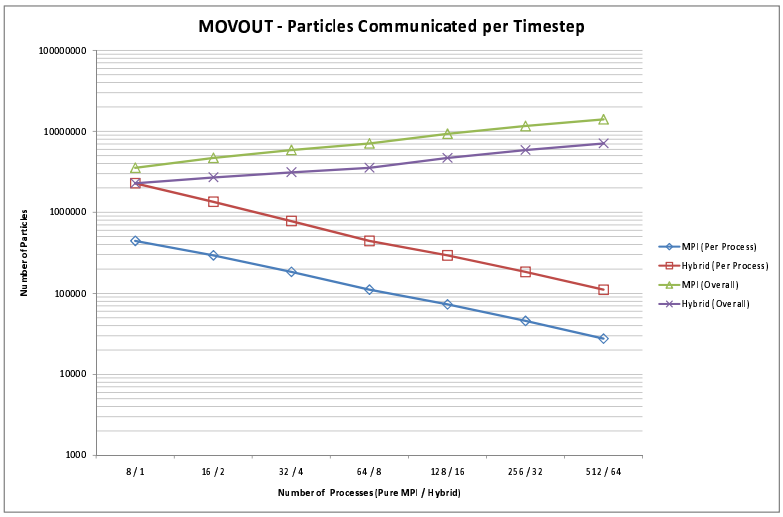


Figure 1: Movout Analysis - Particles Communicated per Timestep

3.0.1 Merlin

Merlin is the main HPC cluster at the Advanced Research Computing facility at Cardiff University. It consists of 256 compute nodes, each containing two quad-core Xeon E5472 Harpertown processors running at 3.0Ghz, with 16GB RAM. Each processor contains four cores with a 32kb instruction cache and a 32kb L1 data cache. Each pair of cores shares a 6MB L2 cache. The nodes are connected by a 20GB/s Infiniband interconnect with 1.8 microsecond latency, as well as a Gigabit Ethernet network. Each node has 1 Infiniband link, and 1 Gigabit Ethernet link. The compute nodes run Red Hat Enterprise Linux 5, with version 10 of the Intel C++ compilers. Bull MPI is used over both the Gigabit Ethernet and Infiniband interconnects. The Gigabit Ethernet interconnect is not a dedicated communication network and is also used for node and job management, which has an impact on some of the results. This use of a non-dedicated network most likely accounts for the poor scaling of some of the code on Merlin seen in Section 5.

3.0.2 CSEEM64T

CSEEM64T consists of 32 compute nodes, each containing two dual-core Xeon 5160 Woodcrest processors running at 3.0Ghz, with 8GB RAM. Each processor is dual-core, with each core containing a 32kb instruction and a 32kb L1 data cache, with a 4MB L2 cache shared between both cores. The nodes are connected by a 20GB/s Infinipath interconnect, as well as a Gigabit Ethernet network. The compute nodes run SUSE Linux 10.1, with version 10 of the Intel C++ compilers. Intel MPI is used over the Gigabit Ethernet network, while Infinipath MPI is used over the Infinipath interconnect.

3.1 Methodology

On each cluster three different sizes of simulation were tested: small, medium and large. The small simulation contains 16,384,000 particles, the medium 28,311,552 particles and the large 44,957,696 particles. Each size was run for 500 timesteps and each was tested three times on a range of core counts. The fastest time of the three runs was used for analysis. The molecular dynamics simulation uses a 3D domain decomposition, so at each core count the most even distribution across all three dimensions was used.

When running the performance tests a number of MPI processes were started on each node and the `OMP_NUM_THREADS` environment variable used to spawn the correct number of threads to use the rest of the cores in the node, giving (MPI)×(OpenMP) cores used per node. Each simulation size and processor core count was tested with three combinations of MPI processes and OpenMP threads, as illustrated in Fig. 2 and described below:

1. MPI - One MPI process started for each core in a node, no OpenMP threads: (4×1 on CSEEM64T, 8×1 on Merlin). (Figure 2(a))
2. Hybrid 1 - One MPI process started on each node, all other cores filled with OpenMP threads: (1×4 on CSEEM64T, 1×8 on Merlin). (Figure 2(b))
3. Hybrid 2 - Two MPI processes started on each node, all other cores filled with OpenMP threads: (2×2 on CSEEM64T, 2×4 on Merlin). (Figure 2(c))

When using less than the full number of cores on a node, the full node was reserved via the job queueing system to ensure exclusive access while the performance testing was carried out.

4 Multicore Performance

The use of multicore processors in modern HPC systems has created issues [17] that must be considered when looking at application performance. Among these are the issues of memory bandwidth (the ability to get data from memory to the processing cores), and the effects of cache sharing (where multiple cores share one or more levels of cache). Some brief experiments have been done to assess the effects of these on the MD code used in this paper.

4.1 Memory Bandwidth

Memory bandwidth issues are easily exposed in a code by comparing performance of the application with fully populated and under populated nodes. By running the MPI code on a number of fully populated nodes on Merlin, (using 8 processes per node, ppn=8), then again on twice the number of nodes using half the cores (ppn=4), we can see if any performance differences occur due to the reduced memory bandwidth on a fully populated node. The results of this test are shown in Fig. 3. They clearly show that there is little if any performance difference between a fully populated and under populated node, demonstrating that memory bandwidth is not an issue with this MD code, so not a factor when examining performance results.

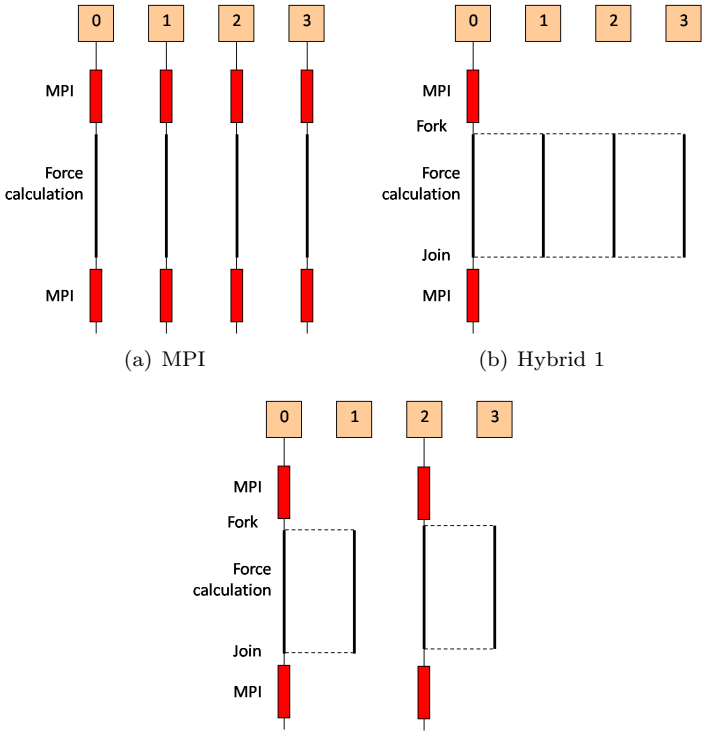


Figure 2: MPI and Hybrid versions.

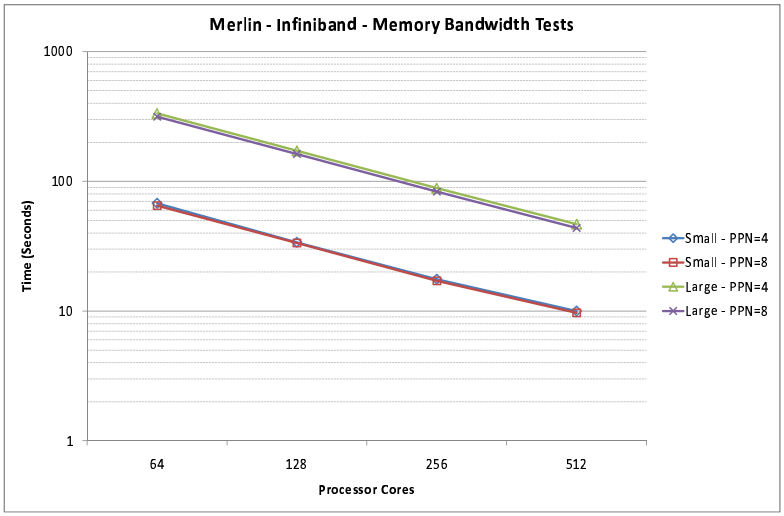


Figure 3: Merlin, Memory Bandwidth Tests

4.2 Cache Sharing

We have examined the effect of cache sharing for both the MPI and Hybrid codes on Merlin. As each node in this cluster contains two quad-core processors, with two pairs of cores on each physical processor sharing the L2 cache, running 4 processes or threads on one processor (using all 4 cores, so the L2 cache is shared) and then on two processors (using one of each pair of cores, so each core has exclusive access to the cache), will expose any performance difference caused by cache sharing. The timing results for the two codes, and difference between the exclusive and shared cache timings are presented in Table 1.

Small Simulation						
	MPI			Hybrid		
	Shared	Exclusive	Diff.(%)	Shared	Exclusive	Diff.(%)
Total	2436.699	2328.641	4.43 %	3622.872	3577.680	1.25 %
Forces	2211.631	2168.351	1.96 %	3305.688	3275.290	0.92 %
Large Simulation						
	MPI			Hybrid		
	Shared	Exclusive	Diff.(%)	Shared	Exclusive	Diff.(%)
Total	8410.663	8103.421	3.65 %	16935.252	16751.750	1.08 %
Forces	7791.354	7691.7058	1.28 %	16061.832	15921.399	0.87 %

Table 1: Timing Effects of Cache Sharing. Times in seconds.

The MPI code is affected more by the cache sharing than the Hybrid code, as the difference between exclusive cache timing and shared cache timing is larger for the total time of the MPI code. This is to be expected as the MPI code runs with four processes continually, whereas the Hybrid code only runs one process until the threads are spawned in the `forces` routine. The Hybrid code is only sharing cache between cores during the execution of this routine, at all other points only one MPI process is running, which will have exclusive access to the cache. It is therefore reasonable to expect that the Hybrid code will be affected less by the cache sharing overall. Examining the portion of the total difference that can be attributed to the `forces` routine, shows that it makes up a far greater proportion of the total difference in the Hybrid code than the MPI code, which is in line with these expectations.

Examining only the `forces` routine timing does not show any clear difference between the two codes. The small simulation shows a larger difference between shared and exclusive cache in the MPI code, while the large simulation shows the bigger difference in the Hybrid code. It is not therefore possible to state that either is affected more or less by the cache sharing than the other. The difference does show that cache sharing has a negative impact on the performance of the code, but that it is not responsible for the large performance difference between the MPI and Hybrid codes when running on small numbers of cores.

5 Performance Results

Throughout the performance results two simple patterns can plainly be observed.

First, the main work portion of the code (the `forces` routine) is always slower in the hybrid code than the pure MPI code. This may be the result of the increased overheads in this section of code related to the spawning, synchronising and joining of the OpenMP threads. The reduction clause needed to ensure the kinetic and virial measures are updated correctly adds an additional overhead and an extra synchronisation point that is not present in the pure MPI code, where all synchronisation occurs outside the `forces` routine in the communication portions of the code.

Second, the hybrid model offers little (if any) performance improvement over the pure MPI code when using a modern low-latency HPC interconnect, such as Infiniband or Infinipath. The pure MPI code in general runs much faster than the hybrid code when using these connections. The communication profile of the pure MPI code suits these low latency high bandwidth interconnect well. However, when using a low latency interconnect such as Gigabit Ethernet, the opposite is seen. When using large numbers of cores over such an interconnect, the hybrid code performs better than the pure MPI code. This is probably because a larger number of MPI processes results in more network traffic and congestion when Gigabit Ethernet is used.

These results do not show any performance gains from the use of shared memory over message passing within nodes. As already discussed, the MPI implementations used for testing include communication devices allowing the intra-node communications to be carried out through shared memory without having to use explicit message passing over the network interconnect. This allows the intra-node communications in the MPI code to be as fast as the shared memory accesses in the hybrid code.

5.1 Merlin

On the Merlin cluster, (two quad-core processors per node), it is clear that when using the Infiniband connection the MPI code is faster than either of the Hybrid versions. The Hybrid 1 (1×8) version is slower than the Hybrid 2 (2×4) approach, while both are slower than the pure MPI code (Fig. 4).

The performance gaps between the three code variants remain fairly consistent as the number of cores increases, demonstrating that the scalability of the three codes is very similar. The performance of the Hybrid 1 and Hybrid 2 codes is also very similar throughout all core counts, something which is not seen when using the Gigabit Ethernet interconnect (Fig. 5).

Examining the breakdown of runtime between routines for the large simulation on both 512 and 128 cores (Tab. 2) shows that the main differences between the Hybrid 1 and pure MPI codes occur in the `forces` and `movout` routines. Both routines have a longer runtime in the Hybrid code when using the Infiniband connection. This pattern, where the `forces` routine has a longer runtime in the hybrid code than in the pure MPI code, is repeated throughout the performance results on both clusters using both interconnects, as already discussed.

The Gigabit Ethernet results on Merlin show that the scalability of both the pure MPI and hybrid codes is an issue when using larger numbers of processors, (Fig. 5), but that in general the hybrid codes outperform the pure MPI code. Above 128 cores there is no performance improvement for either the Hybrid 1 or pure MPI code. Above 192 cores the pure MPI code performance steadily

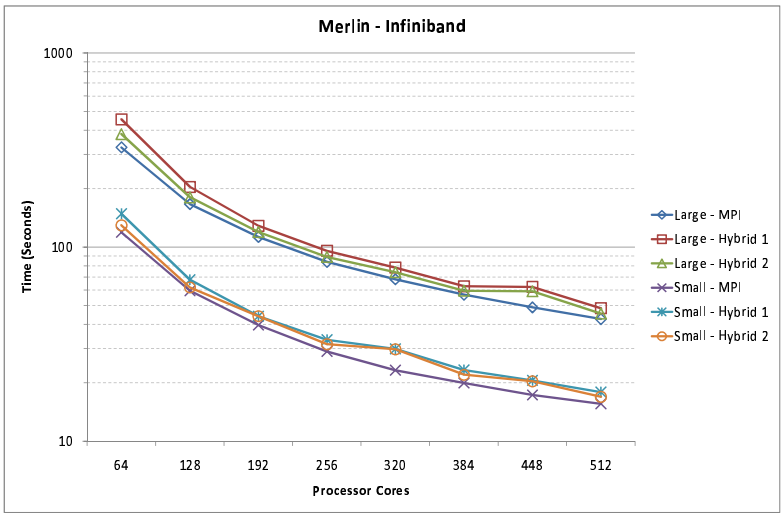


Figure 4: Merlin, Infiniband, Overall Timing Results

512 Cores		
	MPI	Hybrid
Forces	34.06927	36.18235
Movout	2.37228	6.98469
MoveA	0.70046	0.94867
MoveB	0.39769	0.6112
Sum	2.09433	0.40785
Hloop	0.07061	0.07222
Startup	2.87173	3.18326

128 Cores		
	MPI	Hybrid
Forces	137.7295	155.33234
Movout	8.75808	28.76653
MoveA	8.57629	8.33893
MoveB	4.25237	5.21269
Sum	2.02575	1.25527
Hloop	1.97784	1.99515
Startup	3.04294	3.42531

Table 2: Routine Breakdown, Large Simulation, Infiniband, Merlin. Times in seconds.

worsens, while the Hybrid 1 performance stays relatively steady. The Hybrid 2 code fluctuates in performance more erratically than either the Hybrid 1 or MPI codes, but delivers the best performance at 192, 256, 384 and 512 cores.

The routine breakdown (Tab. 3) on Gigabit Ethernet shows that the main difference in timings between the pure MPI and hybrid codes comes again in the `movout` routine, which has a far longer runtime in the pure MPI code on 512

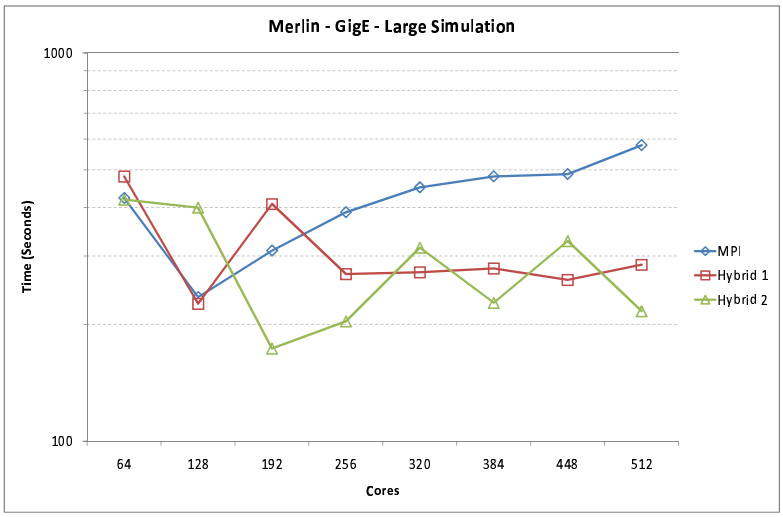


Figure 5: Gigabit Ethernet, Merlin, Large Simulation

cores than the hybrid code. There is also a large difference in the `sum` routine, which uses collective communications, as the pure MPI performs worse with the collectives over the Gigabit Ethernet, while the hybrid code fares better due to the reduced number of MPI processes involved in the communication.

512 Cores		
	MPI	Hybrid
Forces	34.20658	36.07332
Movout	415.74687	234.70416
MoveA	0.66362	0.96054
MoveB	0.31689	0.5956
Sum	101.97926	1.44475
Hloop	0.11349	0.07668
Startup	25.75003	9.93624

Table 3: Routine Breakdown, Large Simulation, Gigabit Ethernet, Merlin. Times in seconds.

5.2 CSEEM64T

On the CSEEM64T cluster a similar pattern to that seen on the Merlin cluster is observed. Using the low-latency, high-bandwidth Infinipath interconnect the pure MPI code outperforms the hybrid codes at all problem sizes. Unlike on Merlin however, there are large differences between the Hybrid 1 and Hybrid 2 results when using the Infinipath connection. The Hybrid 2 results are often much slower than the pure MPI – up to twice as slow in some cases – and also much slower than the Hybrid 1 timings (Fig. 6). This plot shows the code running at all problem sizes on 64 cores, and it is plain to see the performance

gap between MPI and the Hybrid codes.

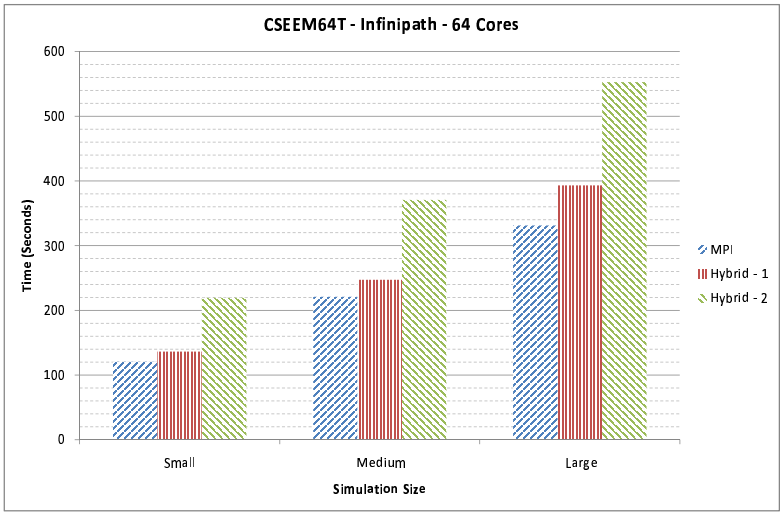


Figure 6: CSEEM64T, Infinipath, 64 Core Timings

Using the Gigabit Ethernet connection the results (Fig. 7) are quite different, with the hybrid codes performing better than the pure MPI at all problem sizes. The Hybrid 1 & Hybrid 2 runtimes are very similar, with Hybrid 1 having a slightly slower performance than Hybrid 2, while both are consistently faster than the pure MPI timings.

An examination of the routine timings breakdown over both interconnects (Tab. 4) shows two things. First, the `forces` routine is again slower in the hybrid code than in the pure MPI code. Second, the main differences between the two codes occurs in the timing of the `movout` routine, as on the Merlin cluster. Using the Infinipath connection the `movout` routine is much faster in the pure MPI code than the hybrid, while this is reversed when using the Gigabit Ethernet connection. On the Infinipath interconnect, the pure MPI code `movout` routine is around twice as fast as that of the hybrid code, while on Gigabit Ethernet the hybrid code `movout` routine is twice as fast as the pure MPI code `movout` routine.

On CSEEM64T, for both Infinipath and Gigabit Ethernet connections, the differences between the pure MPI and Hybrid 1 codes are in the region of 20-60 seconds, depending on the simulation size. Using the Infinipath connection the pure MPI code is faster, using the Gigabit Ethernet connection the Hybrid 1 code is faster.

5.3 Interconnects

The communication phases of the code are where the major differences between the MPI and Hybrid timing profiles occur. The interconnect used for communication has a significant effect on this phase of the code.

Figure 8 shows the time spent in the `movout` routine for the pure MPI, Hybrid 1 and Hybrid 2 codes on both CSEEM64T and Merlin, for both interconnects

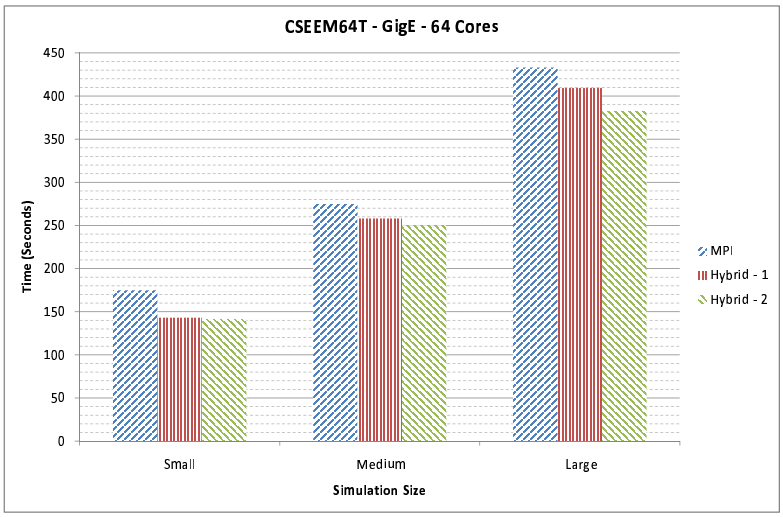
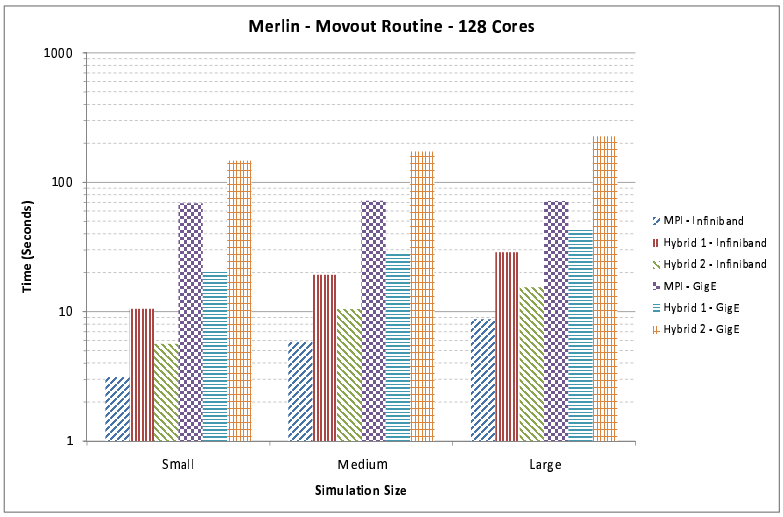


Figure 7: CSEEM64T, Gigabit Ethernet, 64 Core Timings

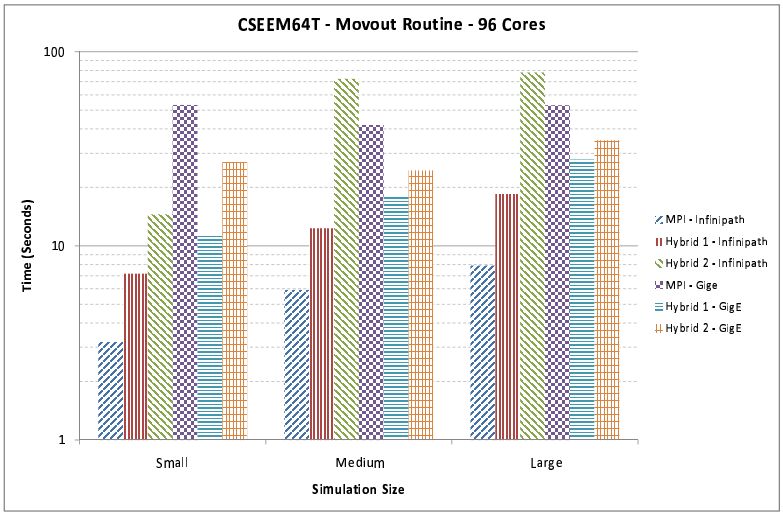
Infinipath		
	MPI	Hybrid
Forces	193.77336	207.99323
Movout	7.90706	18.5059
MoveA	7.377	12.5858
MoveB	3.64191	5.08047
Sum	0.90146	0.26923
Hloop	2.07302	2.3668
Startup	4.10984	3.794
Gigabit Ethernet		
	MPI	Hybrid
Forces	194.5966	207.18975
Movout	53.00865	27.87451
MoveA	7.42247	12.57763
MoveB	3.48986	5.05727
Sum	1.3424	1.29131
Hloop	2.06386	2.3614
Startup	9.39962	9.38928

Table 4: CSEEM64T Routine Breakdown, Large Simulation. Times in seconds.

on each system. As expected, the difference between the standard Gigabit Ethernet interconnect and the high-end Infinipath and Infiniband interconnects can be quite large, with the GigE interconnect consistently much slower than the faster HPC interconnects. For the small simulation on Merlin, the MPI `movout` code running on Gigabit Ethernet is around 172 times as slow as for the Infiniband connection. On CSEEM64T for the small simulation, the Gigabit Ethernet is about 17 times as slow. For the large simulation, the difference is



(a) Merlin - 128 Cores

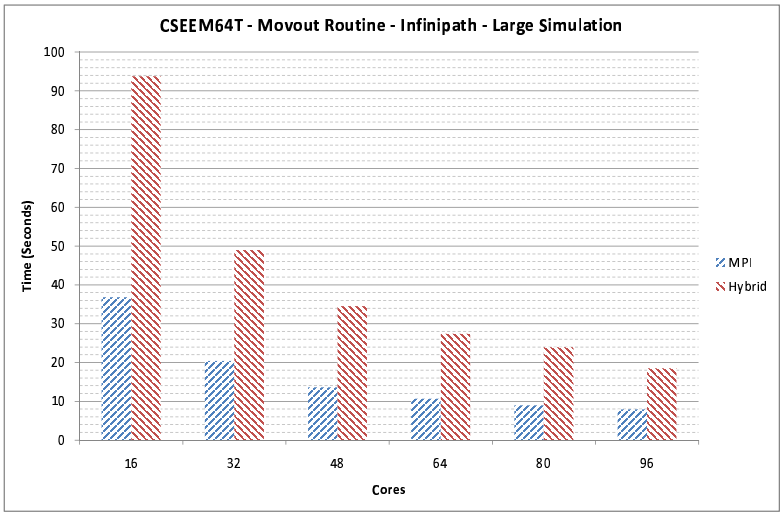


(b) CSEEM64T - 96 Cores

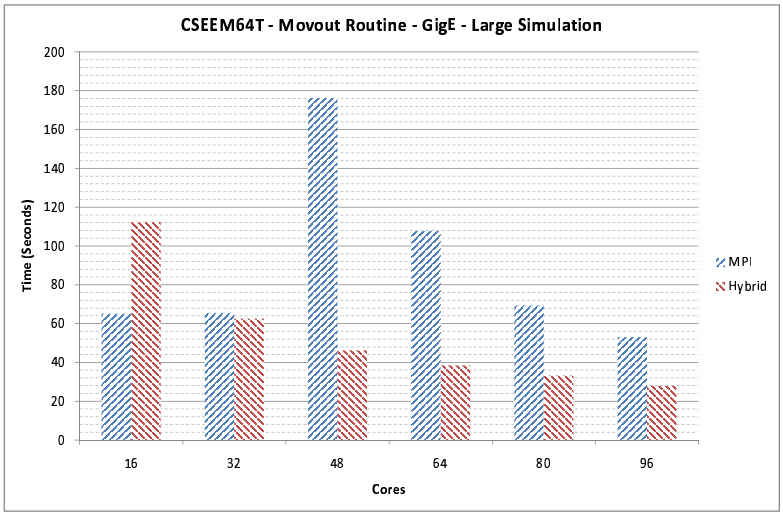
Figure 8: Movout Routine Timings, Large Simulation, Interconnect Comparison

about 68 times on Merlin, and 6 times on CSEEM64T. The Hybrid code fares a little better than this however. For all size simulations on Merlin, the difference between the Infiniband and Gigabit Ethernet times for the `movout` routine is only around ten seconds. On CSEEM64T, this difference is even less, being somewhere between three and ten seconds. The MPI code spends less time in the communication phase than the hybrid codes when using the Infiniband and Infinipath interconnects, but the reverse is true when using the Gigabit Ethernet connection. On Merlin, in the Hybrid 1 code the `movout` routine is much faster than for the pure MPI code on this interconnect, while on CSEEM64T both the Hybrid 1 and 2 `movout` code are faster. Of interest is the fact that on Merlin

at 128 cores using the GigE connection, the Hybrid 2 code is much slower than both the MPI and Hybrid 1 codes. This correlates with the poor scaling of the code seen in Fig. 5, where Hybrid 2 is much slower than both Hybrid 1 and the MPI code at this core count. This may be caused by a number of factors, perhaps due to the placement of processes at this core count resulting in more inter-node communication for the Hybrid 2 code.



(a) Infinipath



(b) Gigabit Ethernet

Figure 9: Movout Routine Timings, CSEEM64T, Large Simulation

This difference in the `movout` routines on the two interconnects is also shown well in Fig. 9. This clearly shows the scaling of the `movout` routine timing as the number of cores increases on the CSEEM64T cluster. For Infinipath, the MPI `movout` code is consistently faster than the Hybrid 1 `movout` routine, but

on GigE this is only true at 16 cores. For all other core counts the Hybrid 1 code spends less time in the `movout` routine than does the pure MPI code.

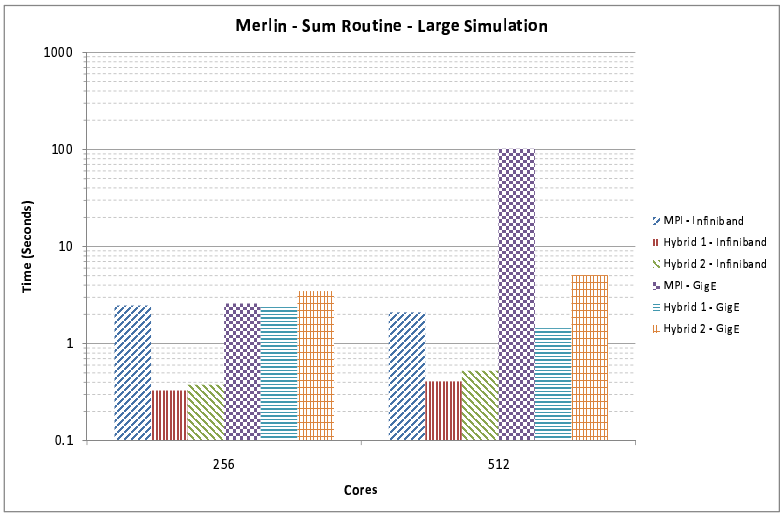


Figure 10: Sum Routine Timings, Large Simulation, Merlin

The interconnect used also affects the timing of the collective communications in the `sum` routine (Fig. 10). The Hybrid 1 code is consistently faster than the pure MPI code on both interconnects, while the Hybrid 2 code is faster on all but Gigabit Ethernet on 256 cores. The most striking result is for 512 cores using the Gigabit Ethernet interconnect, where the Hybrid 1 code is 98.52% faster than the MPI code. It is also interesting to see that at both core counts, the Hybrid 1 and 2 codes are more than twice as fast as the pure MPI code on the Infiniband interconnect, where the hybrid code overall is much slower than the pure MPI. However this routine, and global communications as a whole, are not a significant part of the runtime, so this routine does not have a major effect on the total time of the application.

6 Conclusion

The hybrid model has often been discussed as a possible model for improving code performance on clusters of SMP's or clusters of multicore nodes. While it has not been found that the shared memory sections of the code bring any performance benefits over using MPI processes on a multicore node, it was found that the communication profile differs between the pure MPI and hybrid codes sufficiently for there to be significant performance differences between the two.

No benefit has been found from shared memory threading on a multicore node over using MPI processes for the work portions of the code. With this particular molecular dynamics code the extra overhead from starting and synchronizing threads results in the shared memory code performing slower than the pure message passing code in the main work sections. Cache sharing has been shown to have less of an effect on the hybrid code than the pure MPI, but

this may be due to the specific implementation of the MD algorithm used in this code, and cannot be generalized to the hybrid model itself.

The MPI and hybrid codes exhibit different communication profiles whose runtimes can be affected depending on the communication interconnect being used. This work has shown that with a molecular dynamics simulation using modern multicore clusters with high-end low-latency interconnects, the hybrid model does not offer any improvement over a pure MPI code. The extra overheads introduced by shared memory threading increase the run time of the main work portion of the code, while the low-latency interconnect removes any benefit from the reduction in the number of messages in the communication phases. However, on a higher latency connection, such as Gigabit Ethernet, there may be much benefit from the hybrid model. The extra overheads from the threading portion are outweighed by the reduction in communication time for both point-to-point and collective communication. The smaller numbers of larger messages in a hybrid code mean that the communication profile is better suited to a higher latency interconnect than a pure MPI code with many smaller messages.

The hybrid model has also shown that it may be suited to codes with large amounts of collective communications. Results in this area look promising, with the hybrid code performing better at collectives than the MPI code on both interconnects tested. More work is needed in this area to see if this is a pattern exhibited by other codes.

7 Future Work

This work has examined two high performance interconnects: Gigabit Ethernet and high-end Infiniband and Infinipath connections. The next task is to examine the performance of the hybrid model on a mid-range standard Infiniband connection to fill in the gap in results between the low-end Gigabit Ethernet connection and the high-end Infiniband/Infinipath interconnects.

Following this it is intended to examine the performance of the hybrid model with larger scale scientific codes. The molecular dynamics simulation used in this work is a fairly simple simulation, modelling only short range interactions between particles. Examining the performance of the hybrid model with a more complicated code may reveal further performance issues not explored in this work. As much of the benefit from the hybrid model seems to come from reduced communications, a code that relies on more intra-node communication for long range force calculation may benefit further from the hybrid model.

References

[1] E. Strohmaier, J. Dongarra, H. Meuer, and H. Simon, "Recent trends in the marketplace of high performance computing," *Parallel Computing*, vol. 31, no. 3-4, pp. 261–273, 2005.

[2] A. Agarwal, "Performance tradeoffs in multithreaded processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 5, pp. 525–539, 1992.

[3] I. Nielsen and C. Janssen, "Multi-threading: a new dimension to massively parallel scientific computation," *Computer Physics Communications*, vol. 128, no. 1, pp. 238–244, 2000.

[4] V. Kazempour, A. Fedorova, and P. Alagheband, "Performance Implications of Cache Affinity on Multicore Processors," *Lecture Notes in Computer Science*, vol. 5168, pp. 151–161, 2008.

[5] S. Alam, P. Agarwal, S. Hampton, H. Ong, and J. Vetter, "Impact of multicores on large-scale molecular dynamics simulations," *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–7, April 2008.

[6] Message Passing Interface Forum, "MPI: A Message Passing Interface Standard Version 2.1." Available at http://www.mpi-forum.org/, June 2008.

[7] OpenMP Architecture Review Board, "OpenMP Application Programming Interface, Version 2.5." Available at http://www.openmp.org, Accessed March 2007.

[8] F. Cappello and D. Etiemble, "MPI versus MPI+ OpenMP on the IBM SP for the NAS Benchmarks," in *Supercomputing, ACM/IEEE 2000 Conference*, pp. 12–12, 2000.

[9] R. Rabenseifner, "Hybrid Parallel Programming on HPC Platforms," in *proceedings of the Fifth European Workshop on OpenMP, EWOMP*, vol. 3, pp. 22–26, 2003.

[10] L. Smith and M. Bull, "Development of mixed mode MPI / OpenMP applications," *Scientific Programming*, vol. 9, pp. 83–98(16), 2001.

[11] D. Henty, "Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modelling," in *Supercomputing, ACM/IEEE 2000 Conference*, pp. 10–10, 2000.

[12] E. Lusk and A. Chan, "Early Experiminents with the OpenMP/MPI Hybrid Programming Model," *Lecture Notes in Computer Science*, vol. 5004, p. 36, 2008.

[13] G. Jost, H. Jin, D. an Mey, and F. Hatay, "Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster," *NASA Ames Research Center, Fifth European Workshop on OpenMP (EWOMP03) in Aachen, Germany*, 2003.

[14] R. Car and M. Parrinello, "Unified Approach for Molecular Dynamics and Density-Functional Theory," *Physical Review Letters*, vol. 55, no. 22, pp. 2471–2474, 1985.

[15] M. Ashworth, I. Bush, M. Guest, A. Sunderland, S. Booth, J. Hein, L. Smith, K. Stratford, and A. Curioni, "HPCx: Towards Capability Computing," *Concurrency and Computation Practice and Experience*, vol. 17, no. 10, pp. 1329–1361, 2005.

[16] J. Hutter and A. Curioni, "Dual-level Parallelism for ab initio Molecular Dynamics: Reaching Teraflop Performance with the CPMD Code," *Parallel Computing*, vol. 31, no. 1, pp. 1–17, 2005.

[17] J. Dongarra, D. Gannon, G. Fox, and K. Kenned, "The Impact of Multicore on Computational Science Software," *CTWatch Quarterly*, vol. 3, pp. 3–10, 2007.