Performance Analysis of a Hybrid MPI/OpenMP Application on Multi-core Clusters

Martin J. Chorley[a], David W. Walker[a]

[a]*School of Computer Science and Informatics, Cardiff University, Cardiff, UK*

Abstract

The mixing of shared memory and message passing programming models within a single application has often been suggested as a method for improving scientific application performance on clusters of shared memory or multi-core systems. DL_POLY, a large scale Molecular Dynamics application programmed using message passing programming, has been modified to add a layer of shared memory threading and the performance analysed on two multi-core clusters. At lower processor numbers, the extra overheads from shared memory threading in the hybrid code outweigh performance benefits gained over the pure MPI code. On larger core counts the hybrid model performs better than pure MPI, with reduced communication time decreasing the overall runtime.

Keywords: multi-core, hybrid programming, message passing, shared memory

1. Introduction

The cluster architecture currently dominates the field of High Performance Computing (HPC). The Top 500 (http://www.top500.org) ranking of world supercomputers in November 2002 showed 18.6 percent of the list were classified as having a cluster architecture. By the November 2009 list, this percentage had grown to 83.4 percent. Multi-core processors are also more popular, with almost all of the Top 500 systems in November 2009 containing processors with a multi-core architecture.

These multi-core clusters represent a change in hardware architecture from previous generations of HPC systems, which were typically either symmetric multi-processors (SMP shared memory systems) or clusters of single core nodes. Multi-core clusters may be considered to be a blend of both these previous generations of HPC architectures. As the number of processing cores within a node increases the nodes begin to resemble SMP machines, where identical processors all share access to a large global memory. However, across the whole cluster the architecture retains the characteristics of a distributed memory machine, where

Email addresses: m.j.chorley@cs.cf.ac.uk (Martin J. Chorley), d.w.walker@cs.cf.ac.uk (David W. Walker)

the memory is distributed between the individual nodes of the system. It is important to make the distinction between individual multi-core nodes and SMP systems. Multi-core processors often involve some level of cache sharing, which can have performance implications for parallel codes [1].

With this change in hardware architecture there is a need to assess application performance and the programming models used within parallel applications to ensure maximum efficiency is being achieved from current HPC codes and systems.

1.1. Hybrid Parallel Programming

MPI [2] is the *de facto* standard for message passing parallel programming, offering a standard library interface that promotes portability of parallel code whilst allowing vendors to optimise communication code to suit particular hardware. OpenMP [3] is the *de facto* standard for shared memory parallel programming, offering a simple yet powerful method of specifying work sharing between threads. Compiler directives are used to allow the programmer to specify where parallelism should occur while leaving low level implementation to the compiler. The mixing of shared memory and message passing programming has been suggested many times as a method for programming applications on multi-core clusters. Combining MPI and OpenMP to create hybrid message passing and shared memory applications is a logical step when using multi-core clusters.

1.2. Previous Work

Although the combining of message passing and shared memory programming models is often suggested as a method for improving application performance on clusters of shared memory systems, a consensus to its effectiveness has not been reached.

Cappello and Etiemble have compared a hybrid MPI/OpenMP version of the NAS benchmarks with the pure MPI versions [4], 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 [5], 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 [6], 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 [7], 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 [8], finding that the hybrid model has benefits on slower connection fabrics. A well known example of a hybrid MPI and OpenMP code is the plane wave Car Parrinello code, CPMD [9]. 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 [10, 11].

Recently the hybrid model has also been discussed on clusters of multi-core nodes [12, 13]. The use of the hybrid model on multi-core clusters, and the performance of the model on such systems has been looked at in general terms with different test tools and performance models. Adhianto and Chapman have considered factors affecting performance of the hybrid model in [14] during the creation of a general performance model for such codes. Thakur and Gropp [15] discuss a test suite that enables the study of the cost of supporting thread safety in MPI implementations, and notice that a slower interconnect "masks some of the overhead of maintaining thread safety". Rabenseifner et al. [12] look at the problems associated with trying to match hybrid model parallelism to multi-core cluster architecture and examine the performance of different methods of hybrid implementations. Bull et al. describe a microbenchmark suite for analysing hybrid code performance and present results of the suite in [13]. In [16] Brunst and Mohr describe the profiling of hybrid codes with Vampir NG.

1.3. Contribution

The work in this paper examines the performance of a real world scientific molecular dynamics code, DL_POLY, under hybrid parallelisation on current production HPC multi-core clusters. Much of the previous work on the hybrid model has been focused on SMP systems or clusters of SMP systems; it is only recently that multi-core clusters have begun to be examined. While multi-core systems share characteristics with SMP systems, there are important differences that make the examination of the hybrid model on these systems a novel direction. Many studies choose to look at the performance of benchmark suites when considering programming model or hardware performance. These suites give a good overall picture of system performance, but do not tell us how specific large scale applications will perform on such systems. This work focuses on the performance of one large scale application: DL_POLY. It also examines two different multi-core clusters with differing characteristics and considers the effect the choice of communication interconnect has on the hybrid model performance.

1.4. Structure

The application used for this performance study is described in Section 2. The hardware and methodoly used for testing are described in Section 3, while performance results and analysis are presented in Section 4. Finally, conclusions are given in Section 5.

2. Hybrid Application

DL_POLY 3.0 is a general purpose serial and parallel molecular dynamics simulation package developed at Daresbury Laboratory [17]. This version of the application uses domain decomposition to parallelise the code and is suitable for large scale simulations on production HPC systems.

As with typical parallel molecular dynamics applications a simulation run is characterised by a repeating pattern of communication and forces calculations during each time step of the simulation. Processes carry out force and other calculations on their respective portions of the data domain, and then communicate the necessary data in order to carry out the next step of the simulation. Communication phases typically send and receive boundary data to and from neighbouring processes, migrate particles from one process to another and collectively sum contributions to system characteristics such as energy and stress potentials.

DL_POLY is a large scale application, written in Fortran 90, capable of a wide range of functionality. As such it contains many routines specifically for carrying out calculations for particular scenarios. The hybrid version of the application was created by modifying those functions and routines that were exercised by specific test cases. Test cases 10, 20 and 30 were used for performance analysis. Test case 10 simulates 343,000 SiC atoms with a Tersoff potential [18], Test case 20 simulates 256,000 Cu_3Au atoms with a Gupta potential [19]. Test case 30 simulates 250,000 Fe atoms with a Finnis-Sinclair potential [20]. These test cases are some of the larger of the provided test cases, which demonstrated acceptable scaling during initial testing.

2.1. Hybrid Version

The hybrid version of DL_POLY was created by adding OpenMP directives into the existing message passing source code. Analysis of the original code demonstrated that the test cases chosen for performance testing exercise the `tersoff_forces` (Test 10 only) and `two_body_forces` (Test 10, 20 and 30) routines primarily; these routines (and sub-routines called from within) were therefore taken as the focus of the hybrid parallelisation. These routines have a structure that includes a loop over all atoms in the system; this is the main work loop of the routine, and the part of the code taking the most runtime. It is this loop that is parallelised using OpenMP in order to create the hybrid code. A parallel region is started before the main work loop to allow each thread to allocate its own temporary arrays to be used in the forces calculation. `omp reduction` clauses are used to synchronise and sum contributions to the data values common to all threads such as energy and stress values. All MPI communication occurs outside of the OpenMP parallel regions as in a *master only* [21] style of hybrid code. The application can therefore be used without any specific support for multi-threading in the MPI library.

There are several differences in the operation of the pure MPI and the hybrid MPI and OpenMP code. Firstly, the hybrid code adds an extra layer of overheads to the molecular dynamics simulation, as with each time step a set of threads must be forked, synchronised and joined for each MPI process. These overheads are not present in the pure MPI code. Secondly, the communication profile of the hybrid code is changed from that of the MPI code. In the hybrid code there are in general fewer messages being sent between nodes, but the individual messages themselves are larger. Collective communication is also carried out between relatively fewer processes than in the pure MPI code running on the same number of cores. Thirdly, some sections of code have a smaller level of parallelism in the hybrid code as they lie outside the parallel region of shared memory threading. These differences are considered in the analysis of performance results (Section 4).

2.2. Instrumentation

The code has been instrumented in order to gain detailed information about the timing of certain parts of the application. In particular the routines responsible for carrying out MPI communication have been timed so that data may be collected on the communication profile of the code; the time spent carrying out point-to-point communication and collective communication has been recorded. Code both inside and outside the molecular dynamics simulation has been differentiated, allowing us to examine the overall application performance as well as the performance of the molecular dynamics simulation itself without the start up and IO overheads. The code has also been profiled using Intel Trace Collector and Analyzer to gather statistics on MPI communication.

3. Performance Testing

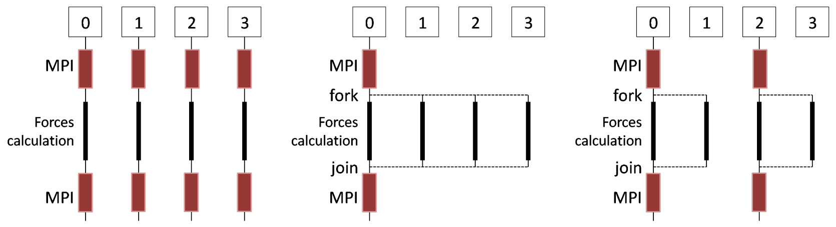
The code has been performance tested on two modern multi-core clusters: Merlin, a production HPC cluster, and Stella, an experimental benchmarking cluster.

3.0.1. Merlin

Merlin is the main HPC cluster at the Advanced Research Computing facility at Cardiff University (ARCCA). It is comprised of 256 compute nodes linked by a 20GB/s Infiniband interconnect. Each node contains two quad-core Intel Xeon E5472 Harpertown processors with a clock speed of 3.0GHz and 16.0GB RAM. The interconnect has a 1.8 microsecond latency and each node has one Infiniband connection. The system software is Red Hat Enterprise Linux 5, with version 11 of the Intel compilers used for compilation of code. Bull MPI 2-1.7 is the MPI library used.

3.0.2. Stella

Stella is a test cluster provided by Intel consisting of 16 nodes linked by a 10 Gigabit Ethernet communication network. Each node has two quad-core Intel Nehalem processors running at 2.93Ghz (giving a total of 128 cores for the whole system), and 24 Gigabytes of DDR3 memory. The communication network is linked with an Arista 7124S switch. The software stack is again based on Red Hat Enterprise Linux 5, with version 11 of the Intel compilers used for compilation and the Intel MPI library used for running the parallel code.



(a) MPI (b) Hybrid 1 (c) Hybrid 2

Figure 1: MPI and Hybrid versions

3.1. Methodolody

The code has been tested using three of the supplied DL_POLY test cases, which exercise the specific parts of the code modified in this work. As input/output performance is not of interest here, the DL_POLY code was modified to remove unnecessary input and output (such as the printing of statistics during simulations) and the test cases were modified to remove keywords relating to the creation of large output files. Each test was run three times on a range of core counts, and the fastest time of each run was used for comparison.

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 processes)$\times$(OpenMP threads) 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. 1 (showing each case running on a node with four cores) and described below:

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

4. Results

As with our previous work using a simpler molecular dynamics code [22] the performance results show an overall pattern linking the performance of the hybrid code, the problem size and the number of processor cores used. At low processor counts, the pure MPI code outperforms the hybrid code, due to the extra overheads introduced in the OpenMP parallelisation. At higher processor numbers the hybrid code performs better than the pure MPI code, as communication becomes more of a limiting factor to performance and the reduced communication times of the hybrid code result in a better overall performance.

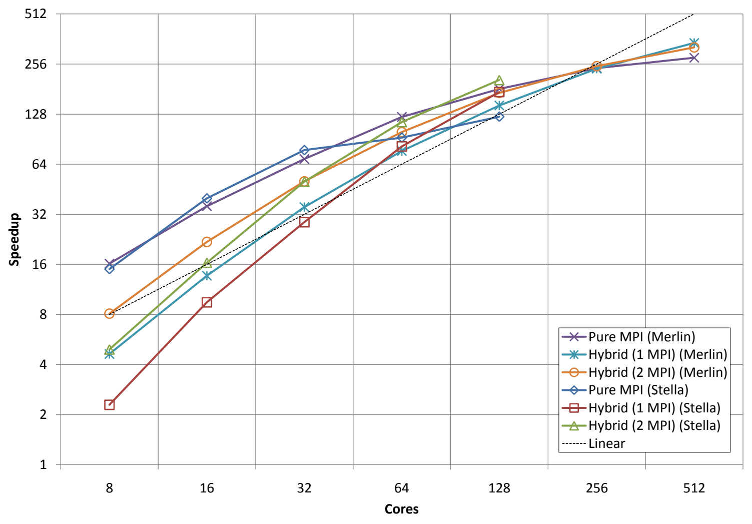


Figure 2: Speedup, Test 10

The previous work with a simpler molecular dynamics code did not show any benefit to the hybrid model on an Infiniband connection; benefits were only seen on slower interconnects. The results from the more complex DL_POLY application do show a benefit to using the hybrid code at high core counts, even on a fast low latency interconnect such as the Infiniband connection on Merlin. We also see that the hybrid model delivers performance benefits on the slower 10GigE connection on Stella, achieving better performance than the pure MPI code on lower processor core numbers than on Merlin.

4.1. Overall Timing

Figures 2 and 3 show the speedup for Test 10 and Test 20 on both clusters. For Test 10 the code scales very well up to 128 cores on Stella and 512 cores on Merlin, scaling better than linearly in some cases. From this plot it can be seen that the hybrid code scales better than the pure MPI at the upper limits of the core counts for both clusters. Test 20 exhibits much worse scaling than Test 10, but still shows the hybrid codes outperforming the pure MPI codes at higher core counts for both clusters.

Table 1 shows the total times for all three tests at all processor core counts on both Merlin and Stella. All three test cases exhibit the behaviour described above, with MPI performing best at lower processor numbers, and hybrid performing best at higher core counts. The point where the hybrid performance improves beyond that of pure MPI is not static.

In order to understand the performance difference between the hybrid and pure MPI codes it is necessary to examine the factors influencing the hybrid

	Merlin			Stella		
	Pure MPI			Pure MPI		
Cores	Test 10	Test 20	Test 30	Test 10	Test 20	Test 30
1	4117.78	244.861	420.727	3570.254	991.268	950.12
8	254.748	347.812	622.872	237.962	280.127	637.684
16	115.048	184.933	320.848	89.495	153.123	358.943
32	59.892	113.086	210.053	45.889	86.251	213.152
64	33.461	61.883	117.93	38.61	61.098	162.447
128	22.699	43.381	101.383	28.914	42.137	137.836
256	17.023	36.599	77.58			
512	14.742	29.822	130.05			
	Hybrid (1 MPI)			Hybrid (1 MPI)		
Cores	Test 10	Test 20	Test 30	Test 10	Test 20	Test 30
1	4117.78	244.861	420.727	3570.254	991.268	950.12
8	889.514	413.678	756.38	1558.429	341.197	901.38
16	301.109	293.243	581.246	377.229	237.64	577.69
32	116.428	154.874	297.767	124.379	124.857	280.447
64	53.58	82.141	150.113	43.615	61.447	159.069
128	28.533	51.052	90.144	20.536	39.309	110.959
256	17.146	32.339	54.108			
512	12.014	20.681	32.545			
	Hybrid (2 MPI)			Hybrid (2 MPI)		
Cores	Test 10	Test 20	Test 30	Test 10	Test 20	Test 30
1	4117.78	244.861	420.727	3570.254	991.268	950.12
8	509.274	420.914	845.346	723.777	356.974	820.162
16	188.618	229.697	424.977	217.888	186.295	417.888
32	81.588	123.048	229.781	71.049	94.837	228.779
64	41.264	71.296	144.279	31.127	55.938	162.962
128	23.98	46.088	92.674	17.385	36.181	117.978
256	16.575	31.25	61.295			
512	12.776	27.341	52.545			

Table 1: Overall Timing Results, Stella and Merlin

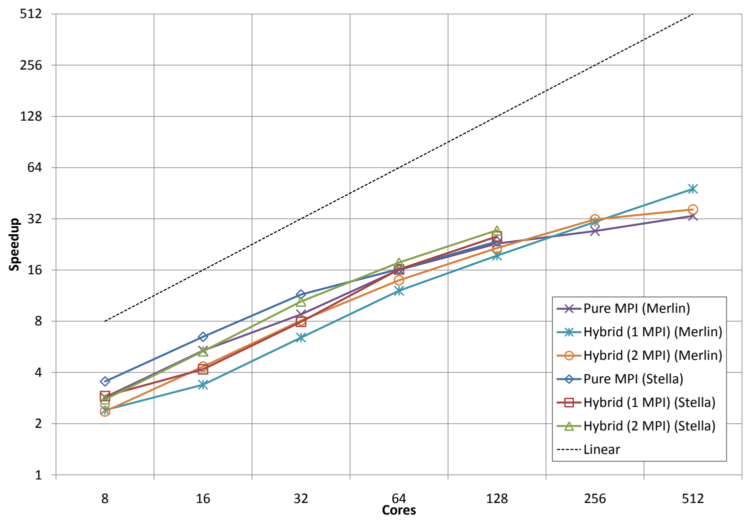


Figure 3: Speedup, Test 20

code performance:

1. **Communication Profile.** The changes to the communication profile of the code may have a large effect on performance, as the number of MPI processes is greatly reduced when running the hybrid code. Message numbers and sizes will therefore be different between the two codes, so the communication sections of code will perform in a different manner. This difference is examined in Section 4.2.
2. **OpenMP Related Overheads.** The extra overheads introduced by shared memory threading may be direct or indirect. Direct overheads are a result of the OpenMP implementation and include time taken to fork/join threads, carry out reduction operations etc. Indirect overheads are not caused by OpenMP itself. For instance, some parts of the code are run in serial on one node in the hybrid version as they are outside the parallel region, where they would be run in parallel in the pure MPI version. The use of OpenMP can also cause the compiler to refrain from carrying out loop transformations and optimisations that would improve performance [23]. These overheads are examined in Section 4.3.

4.2. Communication Profile

Simple statistics collection from the code illustrates the difference between the communication profiles of the hybrid and the pure MPI code. Figure 4 shows the average data transferred per process per time step and the total amount of data transferred between all processes per time step for Test 20. The general

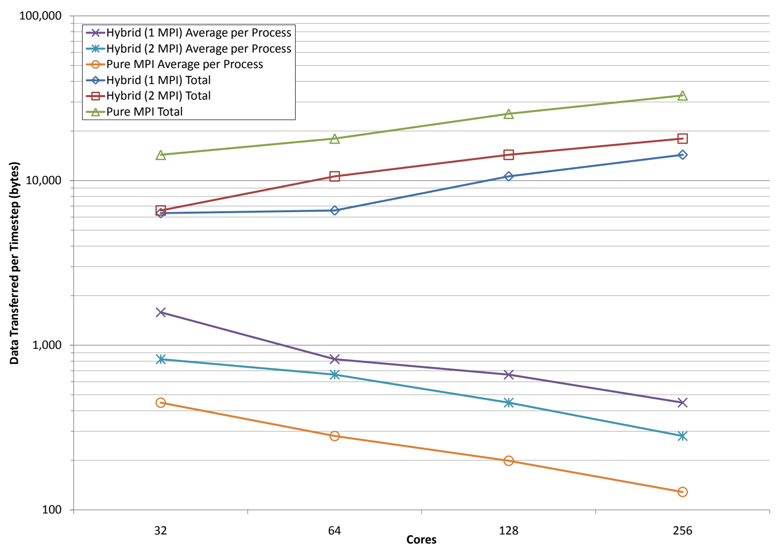


Figure 4: Communication Profile, Test 20

pattern revealed is that the average amount of data transferred per MPI process is higher in the hybrid code, while the total amount of data transferred is larger in the MPI code. The communication profile could be generalised by saying that the hybrid code has fewer large messages per timestep while the pure MPI code has more smaller messages per timestep. The ability of the interconnection network of a cluster to handle multiple small messages or fewer larger messages will therefore have an effect on the performance of the hybrid code relative to the pure MPI code.

Examining the time spent carrying out communication shows the empirical difference between the communication profile of the hybrid and pure MPI codes. The total communication time for Test 30 (Fig. 5) on Merlin illustrates that the total time spent in communication reduces for the hybrid codes as the number of cores increases, while it remains relatively constant for the pure MPI code. On the 10 GigE connection on Stella the total communication time has an overall upward trend for both hybrid and pure MPI codes, but remains consistently lower for the hybrid (1 MPI) code.

Looking at the time spent carrying out communication as a percentage of run time illustrates further the difference in communication pattern between the MPI and hybrid codes. Figure 6 shows these results for Test 10 on both clusters. It is clear that the communication as a percentage of total runtime is much lower in the hybrid code at higher core counts on both clusters.

We can also break down the communication to see the difference between collective communication, point-to-point communication and barrier synchronisation time. The time spent in `MPI_BARRIER` synchronisation is not a large part

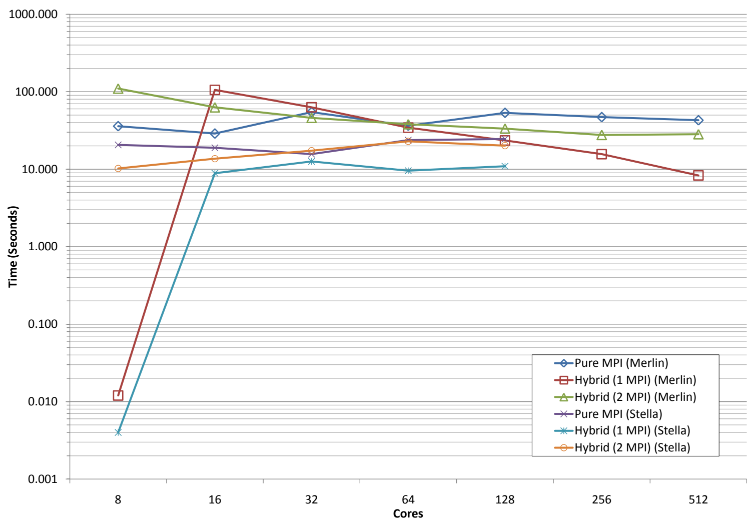


Figure 5: Communication Time, Test 30, Merlin and Stella

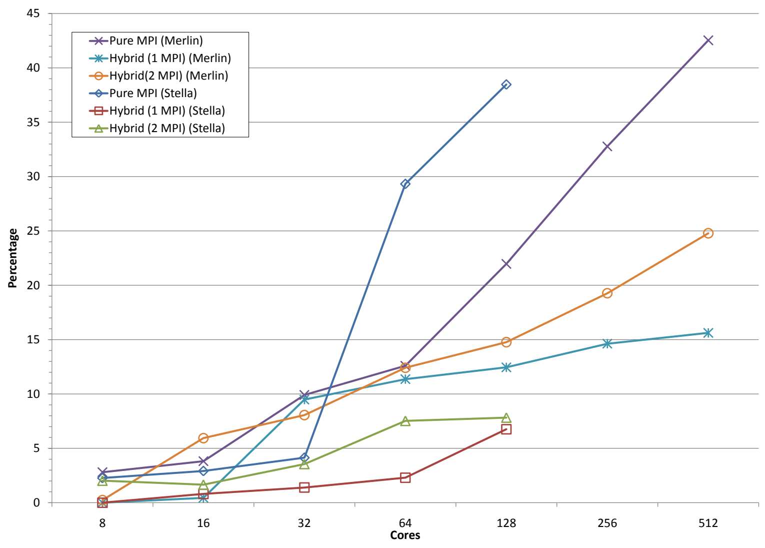


Figure 6: Communication Time as percentage of Total Time, Test 10

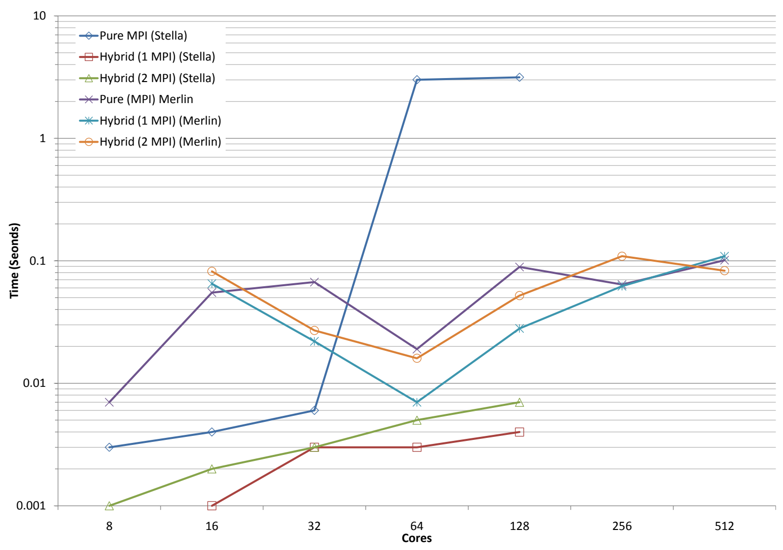


Figure 7: Synchronisation Time, Test 10

of the total runtime for either the pure MPI or hybrid codes, but examining the timings (Fig. 7) shows that the hybrid code performs much better on the slower 10 GigE connection than the pure MPI code, as may be expected due to the smaller number of MPI processes. This is especially true at 64 and 128 cores. Over the faster Infiniband connection on Merlin there is less difference between the codes, but in general the hybrid code performs better here too.

The time spent carrying out collective communication (Fig. 8 shows Test 20 results on Merlin and Stella) folows a similar pattern to that of the synchronisation time. The hybrid code spends far less time on collective communication than the pure MPI code above 64 cores on Merlin and 32 cores on Stella, while the pure MPI code performs better below those counts. Again, this is largely due to the reduced number of MPI processes in the hybrid code,

A different pattern is seen in the point-to-point communication time (Test 30 results shown in Fig. 9). Here we see that the hybrid code spends less time on point-to-point communication than the pure MPI code even at lower core numbers.

4.3. Threading Overheads

We can gain an understanding of the direct and indirect overhead introduced by the shared memory threading by looking at the difference in runtime of the main work loops in both the pure MPI and hybrid codes. Looking at the runtime of the main work loop in the `two_body_forces` routine, we can take the pure MPI time as a baseline, and examine the difference between that and the hybrid code runtime to get an understanding of the overheads that result

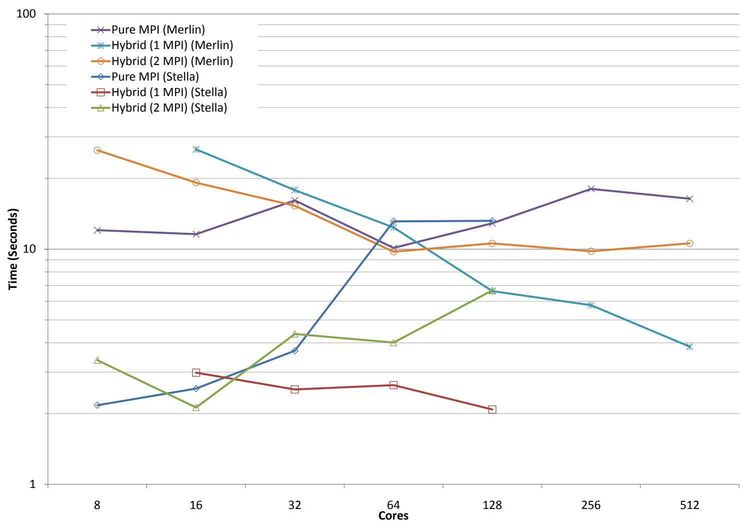


Figure 8: Collective Communication Time, Test 20, Merlin and Stella

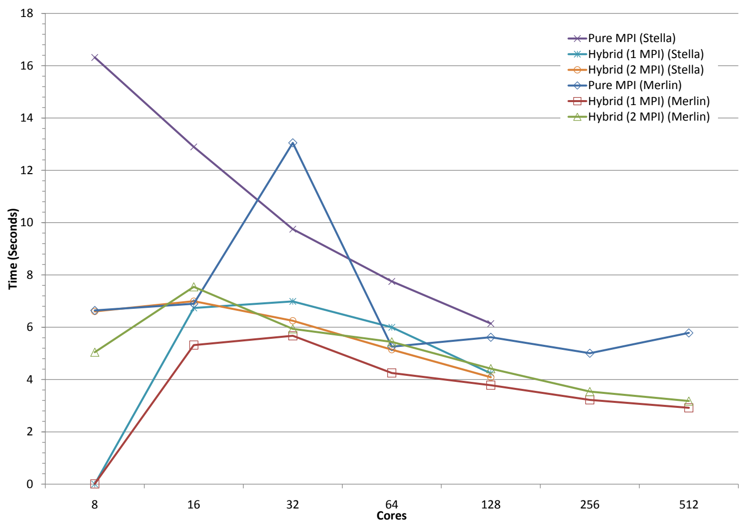


Figure 9: Point to Point Communication Time, Test 30, Merlin and Stella

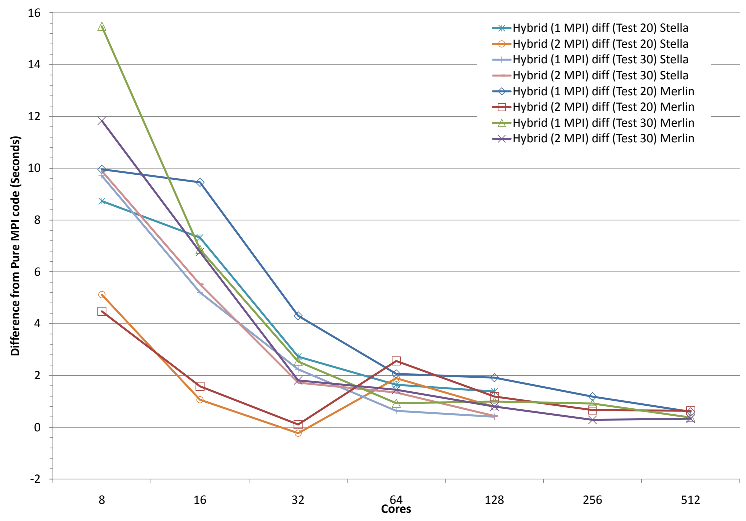


Figure 10: two_body_forces Loop Timing, Test 20 and 30

directly from the shared memory threading of the loop compared to the pure MPI parallelisation (Figure 10). It is apparent that while the overheads are quite large on small numbers of cores, they shrink considerably as the code is run on larger numbers of cores. These overheads are therefore less of an issue at the large core counts, where the better performance of the hybrid code is seen.

We can look at the performance of an individual routine that has been mostly parallelised in the hybrid version. The `metal_ld_compute` routine is called from within the `two_body_forces` routine, and contains two loops that have been parallelised with OpenMP and some global communication. Again, we take the pure MPI timing as a baseline and calculate the difference between that and the hybrid code runtime (Fig. 11). Here we see a different pattern to that observed previously - namely that the hybrid code outperforms the pure MPI at a higher core count on Merlin, but not Stella. On Stella the hybrid code outperforms the pure MPI in two instances (Hybrid (2 MPI) Test 20 at 32 cores, and Hybrid (1 MPI) Test 30 at 128 cores), but otherwise the pure MPI code performs better. On Merlin, the hybrid code performs better for Test 30 after 32 cores, and for Test 20 after 128 cores.

5. Conclusions and Future Work

We have modified the parallel molecular dynamics application DL_POLY 3.0 to create a hybrid message passing and shared memory version by adding OpenMP into the already existing MPI code. This code was tested on two multi-core clusters.

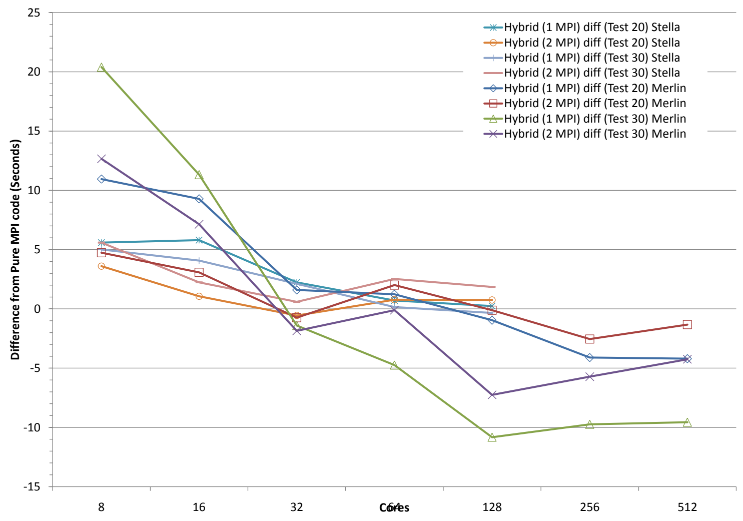


Figure 11: metal_ld_compute Timing, Test 20 & 30

Performance analysis of the hybrid DL_POLY code shows that at smaller core numbers on both systems the pure MPI code outperformed the hybrid message passing and shared memory code. The slower performance of the hybrid code at low core numbers is due to the extra overheads from the shared memory implementation, and the lack of any significant benefit from a reduced communication profile. For more cores on both systems, the hybrid code delivered better performance. In general the hybrid code spends less time carrying out communication than the pure MPI code, performing better at point to point communication at all core counts, and collective communication at higher core counts. This reduced communication is the main driver for performance improvements in the hybrid code. At low core counts the added overheads from OpenMP parallelisation reduce the hybrid code performance, but the effects of these overheads lessen as the number of cores increases.

The choice of system interconnect has an effect on the performance of the hybrid code when compared to the pure MPI code. Using a fast Infiniband interconnect the pure MPI code outperforms the hybrid up to a larger number of cores than when using a slower 10 GigE interconnect.

In order to understand the hybrid shared memory and message passing model further, investigation needs to be carried out to examine the effect the hybrid model has on other large scale applications in other computational science domains.

Acknowledgements

Thanks to ARCCA and Martyn Guest for access to cluster resources and their help and advice, and to William Smith and Ilian Todorov at Daresbury Laboratory for their assistance with the DL_POLY code.

References

[1] S. Alam, P. Agarwal, S. Hampton, J. Vetter, Impact of Multicores on Large-Scale Molecular Dynamics Simulations, 2008 IEEE International Symposium on Parallel and Distributed Processing (2008) 1–7.doi:10.1109/IPDPS.2008.4536181.

[2] Message Passing Interface Forum, MPI: A message-passing interface standard, International Journal of Supercomputer Applications 8 (3&4) (1994) 159–416.

[3] OpenMP Architecture Review Board, OpenMP Application Program Interface Version 2 (2005).

[4] F. Cappello, D. Etiemble, MPI versus MPI+ OpenMP on the IBM SP for the NAS Benchmarks, in: Proceedings of the ACM/IEEE 2000 Supercomputing Conference, 2000, p. 12.

[5] D. S. Henty, Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modelling, in: Supercomputing, ACM IEEE 2000 Conference, 2000, p. 10.

[6] L. Smith, J. Bull, Development of Mixed Mode MPI/OpenMP Applications, Scientific Programming 9 (2001) 83–98.

[7] E. Lusk, A. Chan, Early Experiments with the OpenMP/MPI Hybrid Programming Model, Lecture Notes in Computer Science 5004 (2008) 36.

[8] G. Jost, H. Jin, D. an Mey, F. Hatay, Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster, in: Fifth European Workshop on OpenMP (EWOMP03) in Aachen, Germany, Vol. 3, 2003.

[9] R. Car, M. Parrinello, Unified Approach for Molecular Dynamics and Density-Functional Theory, Physical Review Letters 55 (22) (1985) 2471–2474.

[10] M. Ashworth, I. Bush, M. Guest, A. Sunderland, S. Booth, J. Hein, L. Smith, K. Stratford, A. Curioni, HPCx: Towards Capability Computing, Concurrency and Computation: Practice and Experience 17 (2005) 1329–1361. doi:10.1002/cpe.895.

[11] J. Hutter, A. Curioni, Dual-level Parallelism for ab initio Molecular Dynamics: Reaching Teraflop Performance with the CPMD Code, Parallel Computing 31 (1) (2005) 1–17.

[12] R. Rabenseifner, G. Hager, G. Jost, Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes, Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009), Weimar, Germany (2009) 427–436.

[13] J. Bull, J. Enright, N. Ameer, A Microbenchmark Suite for Mixed-Mode OpenMP/MPI, Springer (2009) 118–131.

[14] L. Adhianto, B. Chapman, Performance modeling of communication and computation in hybrid MPI and OpenMP applications, Simulation Modelling Practice and Theory 15 (2007) 481–491.

[15] R. Thakur, W. Gropp, Test suite for evaluating performance of multithreaded MPI communication, Parallel Computing 35 (12) (2009) 608–617.

[16] H. Brunst, B. Mohr, Performance Analysis of Large-scale OpenMP and Hybrid MPI/OpenMP Applications with VampirNG, Lecture Notes in Computer Science 4315 (2008) 5.

[17] W. Smith, I. Todorov, The DL_POLY_3.0 User Manual, Daresbury Laboratory, 2009.

[18] J. Tersoff, Modeling solid-state chemistry: Interatomic potentials for multicomponent systems, Physical Review B 39 (8) (1989) 5566–5568.

[19] F. Cleri, V. Rosato, Tight-binding potentials for transition metals and alloys, Physical Review B 48 (1) (1993) 22–33.

[20] X. Dai, Y. Kong, J. Li, B. Liu, Extended Finnis–Sinclair potential for bcc and fcc metals and alloys, Journal of Physics: Condensed Matter 18 (2006) 4527–4542.

[21] R. Rabenseifner, Hybrid parallel programming: Performance problems and chances, in: Proceedings of the 45th Cray User Group Conference, Ohio, 2003, pp. 12–16.

[22] M. Chorley, D. Walker, M. Guest, Hybrid Message-Passing and Shared-Memory Programming in a Molecular Dynamics Application On Multicore Clusters, International Journal of High Performance Computing Applications 23 (2009) 196–211. doi:10.1177/1094342009106188.

[23] G. Hager, G. Jost, R. Rabenseifner, Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes, in: Proceedings of the Cray User Group, 2009, pp. 4–7.