Performance Engineering of Hybrid Message Passing + Shared Memory Programming on Multi-Core Clusters

Martin James Chorley

2012

Cardiff University
School of Computer Science & Informatics

A thesis submitted in partial fulfilment of the requirement for the degree of Doctor of Philosophy

Declaration

This work has not previously been accepted in substance for any other degree or award at this or any other university or place of learning and is not concurrently submitted in candidature for any degree or other award.

Signed ..... (candidate)

Date .....

Statement 1

This thesis is being submitted in partial fulfilment of the requirements for the degree of PhD.

Signed ..... (candidate)

Date .....

Statement 2

This thesis is the result of my own independent work/investigation, except where otherwise stated. Other sources are acknowledged by explicit references. The views expressed are my own.

Signed ..... (candidate)

Date .....

Statement 3

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... (candidate)

Date .....

For Lisa

Abstract

The hybrid message passing + shared memory programming model combines two parallel programming styles within the same application in an effort to improve the performance and efficiency of parallel codes on modern multi-core clusters. This thesis presents a performance study of this model as it applies to two Molecular Dynamics (MD) applications. Both a large scale production MD code and a smaller scale example MD code have been adapted from existing message passing versions by adding shared memory parallelism to create hybrid message passing + shared memory applications. The performance of these hybrid applications has been investigated on different multi-core clusters and compared with the original pure message passing codes. This performance analysis reveals that the hybrid message passing + shared memory model provides performance improvements under some conditions, while the pure message passing model provides better performance in others. Typically, when running on small numbers of cores the pure message passing model provides better performance than the hybrid message passing + shared memory model, as hybrid performance suffers due to increased overheads from the use of shared memory constructs. However, when running on large numbers of cores the hybrid model performs better as these shared memory overheads are minimised while the pure message passing code suffers from increased communication overhead. These results depend on the interconnect used. Hybrid message passing + shared memory molecular dynamics codes are shown to exhibit different communication profiles from their pure message passing versions and this is revealed to be a large factor in the performance difference between pure message passing and hybrid message passing + shared memory codes. An extension of this result shows that the choice of interconnection fabric used in a multi-core cluster has a large impact on the performance difference between the pure message passing and the hybrid code. The factors affecting the performance of the applications have been analytically examined in an effort to describe, generalise and predict the performance of both the pure message passing and hybrid message passing + shared memory codes.

Acknowledgements

Throughout this research I have been supported by many different people who deserve acknowledgement here. Firstly I must thank my supervisor David Walker for his invaluable assistance and support over the last five years. I must also thank Martyn Guest for technical advice and support; without his aid in gaining access to multi-core systems the performance analysis in this thesis would be much slimmer. The advice of the external examiners was also gratefully received and helped to improve the thesis immensely, as did the tireless work of Professor Alex Gray, who deserves many thanks for his help and assistance. Thanks are due to all the system admins and developers who have put up with a barrage of emails and questions from me over the years, and their frequent and timely responses were a huge help. The community of research students in COMSC is a great support network and thanks are due to all of them. Special thanks are due to Ian Cooper and Gualtiero Colombo who have been great office mates and work colleagues over the last few years, and Mark Hall who provided much support during his time here. I must also thank Matt, Chris, Will, Mark, Rich and Jon for keeping me entertained and distracted for the last year. I definitely need to acknowledge my colleagues and bosses, Roger Whitaker and Stuart Allen, who have led me onto new and exciting research projects and then put up with me when I needed to stop and come back to this thesis. I would like to thank everyone at EPCC in Edinburgh where I spent my time on the MSc in HPC during my PhD, being there taught me an enormous amount about HPC and computational science, and I am forever grateful for the opportunity to spend time in a truly great city. My friends have also provided much support while I have been working on this thesis. Robert and Carin, Chris J and Chris F have all made frequent visits to Cardiff and provided most welcome distractions from my work over the last few years. Finally of course I must thank my family. Mum and Ed, who have provided both emotional and financial support during my PhD deserve big thanks, as do my sister Hannah and nephews Ryan and Alex. I must also thank my father for the invaluable life lessons I have learnt from him in the last few years. Most of all, I thank Lisa. Her support and encouragement has been unwavering, and it is down to her that I have finally reached this point.

Thank you all.

Contents

Abstract **vii**

Acknowledgements **ix**

Contents **xi**

1 Introduction **1**

1.1 Introduction 2

1.2 Thesis Aims 4

1.3 Contributions 4

1.4 Thesis Structure 5

1.4.1 Small Scale MD Code 7

1.4.2 Real World MD Application 8

1.4.3 Performance Analysis 9

2 Background **11**

2.1 Parallel Programming 13

2.1.1 Data Parallelism 13

2.1.2 Message Passing and Shared Memory Programming 15

2.1.3 MPI 17

2.1.4 OpenMP 27

2.1.5 MPI vs. OpenMP 30

2.2 Multi-core Clusters 30

2.2.1 Multi-core Processors 32

2.2.2 Multi-core Nodes vs. SMP Systems 34

2.2.3 HPC Interconnects 36
2.3 Hybrid Programming 38
2.3.1 Hybrid Model Performance 39
2.3.2 Hybrid Code Profiling 41
2.3.3 Performance Issues with the Hybrid Model 42
2.3.4 Classifying Hybrid Codes 44
2.3.5 Other MP + SM Combinations 45
2.3.6 Previous Work Comparison 46
2.4 Performance Modelling 46
2.4.1 Modelling MPI Routines 48
2.4.2 Modelling Hybrid Code 49
2.5 Molecular Dynamics 50

3 Performance Testing **53**
3.1 Hardware 54
3.1.1 Merlin 54
3.1.2 CSEEM64T 54
3.1.3 Stella 55
3.1.4 Hardware Performance 55
3.2 Testing Methodology 59
3.2.1 Performance Timing 60
3.2.2 Job Schedulers 62
3.2.3 MPI Processes and OpenMP Threads 65
3.3 Nodes/Cores/Processes/Threads 65

4 Small Scale MD Code **69**
4.1 MD Introduction 70
4.1.1 Hybrid Version 71
4.2 Performance Testing 73
4.2.1 Methodology 73
4.3 Hybrid Code Performance 74

4.3.1 Memory Bandwidth and Cache Sharing 75
4.3.2 Overall Timing 78
4.3.3 Communication Profile 88
4.3.4 Shared Memory Overheads 98
4.3.5 Routine Breakdowns 101

5 Real-World MD Application 107
5.1 DL_Poly Introduction 108
5.1.1 Hybrid Version 109
5.2 Performance Testing 112
5.2.1 Methodology 112
5.3 Hybrid Code Performance 113
5.3.1 Overall Timing 113
5.3.2 Communication Profile 121
5.3.3 Threading Overheads 132

6 Hybrid Model Performance 141
6.1 Introduction 142
6.2 Communication Profile 144
6.2.1 Point-to-Point Communication 145
6.2.2 Collective Communication 156
6.2.3 Total Communication 161
6.2.4 Communication Timing Performance Models 166
6.3 Computation 167
6.3.1 Direct Overheads 167
6.3.2 Indirect Overheads 173
6.4 Overall Performance 174
6.5 Applicability to General Hybrid Code Performance 179
6.6 Future Hybrid Code Performance 182
6.7 Performance Model Evaluation 186

7 Conclusion **189**

7.1 Introduction 190

7.2 Shared Memory Threading 191

7.3 Communication Profile 192

7.4 Performance Models 193

7.5 Hybrid Code Performance 194

7.6 Future Work 195

7.6.1 Hybrid Implementations 195

A Practical Notes on Running Hybrid Code **197**

List of Figures **199**

List of Tables **207**

Listings **211**

Bibliography **213**

Chapter 1

Introduction

Overview

This thesis is concerned with examining and analysing the hybrid message passing + shared memory parallel programming model as it applies to a particular class of parallel applications and their performance on multi-core clusters. The performance of the hybrid model has been examined with reference to the performance of the pure message passing programming model, allowing comparisons to be made between the two. Analysis of the performance of the two models reveals the main differences between the paradigms and allows understanding of the model performance.

This chapter presents an overview of the thesis topic and a motivation for the work included. It presents the main aims of the thesis, describes the work carried out and the contributions arising from that work. It also presents the structure of the thesis.

1.1 Introduction

This thesis is focused in the field of High Performance Computing (HPC). HPC is a fast changing area, where technologies and architectures are constantly advancing; a field "characterized by a rapid change of vendors, architectures, technologies and the usage of systems" [109, 108]. Much programming effort is focused on improving and maximising parallel efficiency of codes on current hardware architectures. The result of this effort may be that parallel code and applications that are written for one generation of hardware may not perform efficiently on the next generation of parallel architectures. "[S]oftware that encapsulates all this time, energy and thought, routinely outlasts ... the hardware it was originally designed to run on" [42], and a "rapid growth in computational capability ... is driving computational scientists to develop new, more complex algorithms to make better use of the systems" [6]. Understanding the best programming models and parallelisation strategies to use in order to achieve maximum efficiency of new hardware is an important task [116, 9]. As hardware changes, application codes and programming models must necessarily change also; with reference to modern multi-core architectures: "We are on the verge of a transformation in software design at least as potent as the change engendered a decade ago by message passing architectures, when the community had to rethink and rewrite many of its algorithms, libraries and applications" [25].

This thesis specifically examines the behaviour and performance of the hybrid message passing (MP) + shared memory (SM) programming model, (sometimes also more distinctively known as mixed-mode programming [23]) which combines both message passing and shared memory programming styles within a single parallel application in order to take advantage of the hierarchical and hybrid nature of the underlying parallel hardware. Two hybrid Molecular Dynamics (MD) applications are analysed within this work. Both are adapted from pre-existing 'pure' message passing codes that have been parallelised using the Message Passing Interface (MPI) library. OpenMP parallelisation is added to these applications to provide shared memory parallelism on top of the existing message passing parallelisation in order to create a hybrid

code. This hybrid message passing + shared memory code is performance tested and compared to the original pure MPI code, to draw conclusions about the performance of the hybrid model. Performance tests are carried out on a number of different multi-core HPC clusters, each of which utilises a different HPC interconnect between nodes.

Much of the previous work on the hybrid message passing + shared memory model focuses on Symmetric Multi-Processing (SMP) systems (shared memory systems) or clusters of SMPs. While multi-core systems share many characteristics with SMP systems, there are significant differences as discussed in Section 2.2 that may affect code performance. As such the study of the hybrid model on multi-core systems is a novel direction of research. The work presented in this thesis contributes further knowledge to the discussion of the hybrid programming model and its suitability for use on multi-core clusters.

As a significant portion of the performance differences between hybrid and pure MPI codes are due to the difference in communication patterns between the two types of code, (see, for example, [26, 62, 98]) this thesis also examines the effect that the choice of interconnect 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, a more standard 1 Gigabit Ethernet connection and a newer 10 Gigabit Ethernet connection. The properties of the interconnect used are found to have a significant bearing on the performance of the hybrid applications, and this is discussed in detail in the performance results.

The thesis provides a quantitative analysis of the differences in performance between a pure message passing and a hybrid message passing + shared memory code. This analysis gives insights into the performance of the hybrid model not only as it applies to the application codes examined in this thesis, but also into how general hybrid codes may perform on current and future generations of HPC architectures.

In the future the number of cores per processor is likely to increase, and these many-core nodes may be combined to form ever-larger clusters. Performance engineering of parallel codes and understanding the performance of the hybrid model is likely to become increasingly important

in order to understand the performance characteristics of future petascale/exascale clusters and will help with understanding how to achieve good performance and scalability when using such systems.

1.2 Thesis Aims

The main aim of this thesis is to offer an understanding of the hybrid message passing + shared memory model, particularly as it applies to Molecular Dynamics codes. It aims to provide a detailed investigation of the performance of such codes on modern multi-core clusters, examining not just the runtime of the applications, but also the factors affecting performance, and to provide an analytic description of these factors. In doing so it aims to increase the general knowledge of the hybrid message passing + shared memory model.

1.3 Contributions

The main contributions offered by this thesis are:

1. A hybrid Molecular Dynamics code has been created, tested and analysed. The analysis of the performance results of this code allow us to draw conclusions on the applicability of the hybrid model to simple parallel applications of this type. Details are revealed as to which parts of the code are significantly affected by the hybridisation, and where the performance differences occur as a result of this.

2. A larger MD application (DL_Poly 3.0) has been adapted to use hybrid parallelisation. This has again been performance tested and analysed, allowing further conclusions to be made on the hybrid model and allowing its applicability to large scale Molecular Dynamics codes to be assessed. This shows how the results from the small scale MD code apply to a real-world application, with larger scope and more complicated communication

and memory access patterns. Again, the results reveal significant differences between the performance of the hybrid message passing + shared memory and the pure message passing codes.

3. The differences between pure message passing and hybrid message passing + shared memory codes have been analytically examined. Models have been created attempting to describe the performance differences that may be seen between the two types of code, breaking these differences into their constituent factors, allowing the behaviour of particular parts of the application to be examined as well as the general performance. These models may aid with the creation of future hybrid codes and with the understanding of current code performance on future generations of parallel architecture.

1.4 Thesis Structure

The thesis is structured as follows:

- Chapter 2 presents the background knowledge behind the work in this thesis and discusses related and relevant work. It begins in Section 2.1 by working from first principles of parallel programming, introducing the ideas of shared memory programming and message passing programming, then introducing the two standards for using both these models. In Section 2.2 multi-core clusters are introduced, describing multi-core processors, the differences between multi-core clusters and previous generations of hardware, and giving information on the HPC interconnects used within these systems. Section 2.3 presents the idea of combining message passing and shared memory programming, analysing previous work on the hybrid model and how the previous work compares and relates to the work contained in this thesis. This is followed with a discussion on performance modelling in Section 2.4. Finally, molecular dynamics codes are introduced in Section 2.5

- Chapter 3 describes the multi-core cluster hardware used for performance testing of applications within the thesis, as well as the software used on these clusters, and then presents the testing methodology used for performance analysis.

- Chapter 4 discusses the first contribution of the thesis: the creation, testing and performance analysis of the first hybrid MD code. It presents the results and draws conclusions based on these. The first MD code used in the thesis is a very simple application written in the C programming language that simulates a three-dimensional fluid using a shifted Lennard-Jones potential to model the short-range interactions between particles. The chapter starts by discussing the code in further detail in Section 4.1, before describing the creation of the hybrid version of the code from the existing message passing version. The overall performance of the code is then described and analysed in Section 4.3, followed by an in-depth look at the differences in both the communication and computation of the two styles of code. The creation of a hybrid version of the code in this chapter acts as a proof of concept, illustrating that while performance gains can be made with the hybrid model, they are highly dependent on both the interconnect used and the number of nodes of a system that are used. Pure message passing performance is much better than hybrid message passing + shared memory code performance on low numbers of nodes, and with fast low-latency interconnects.

- Chapter 5 describes the contribution arising from creating and testing a hybrid version of the real world application DL_Poly, extending the conclusions from the previous chapter to see how they apply to a larger scale code with a more complicated communication pattern. It follows a similar pattern to the previous chapter, beginning by introducing the code in Section 5.1, describing the general structure and the creation of the hybrid version. It then presents overall results of a performance comparison of the pure message passing and the hybrid message passing + shared memory versions of the code in Section 5.3. This overall discussion is again extended to look in detail at the performance differences between the two types of code. This second application is written in Fortran and is much larger in scale, containing more complicated communication patterns and increased

amounts of collective communication. The hybrid version of this code shows that hybridisation of existing large scale applications can also deliver performance gains, this time obtaining better performance than the pure message passing code at large processor numbers over a fast Infiniband interconnect. However, the performance of the pure message passing code is still better than the hybrid message passing + shared memory code on low numbers of processors.

- Chapter 6 gathers the main performance results from the analysis of the two hybrid codes and examines the factors affecting performance of the hybrid message passing + shared memory and pure message passing codes. It introduces and examines performance models which describe the behaviour of these codes, enabling an analytical understanding of the code performance. These results are generalised to the differences between the pure message passing model and hybrid message passing + shared memory models allowing prediction of performance of other similar parallel applications on current and future hardware.

- Chapter 7 presents the conclusions of the work, drawing together the findings from the previous chapters, and in Section 7.6 it highlights future directions for this avenue of research.

1.4.1 Small Scale MD Code

The work presented in Chapter 4 investigates the performance of a hybrid molecular dynamics application and compares the performance of the application to the same code parallelised using pure MPI. Much of the work in this chapter has been published as [32]:

- M.J. Chorley, D.W. Walker, M.F. Guest (2009), Hybrid Message-Passing and Shared-Memory Programming in a Molecular Dynamics Application On Multicore Clusters, 196-211. In International Journal of High Performance Computing Applications 23 (3). **doi:10.1177/1094342009106188**

In this chapter the performance of both versions of the code is considered on two high-end multi-core systems, and the effect that the choice of interconnection fabric has on the performance of a pure MPI code compared with a hybrid MPI and OpenMP code is discussed, examining both high-end HPC Infiniband and Infinipath interconnects and a lower performance 1 Gigabit Ethernet connection.

The pure MPI code is found to perform better than the hybrid code on higher end Infiniband and Infinipath connections and at lower processor numbers on the Gigabit Ethernet connection. However, at higher processor numbers on the standard 1 Gigabit Ethernet connection the hybrid model has the better performance.

1.4.2 Real World MD Application

The work presented in Chapter 5 examines the performance of a large scale hybrid molecular dynamics application, again comparing the performance with a pure MPI version of the code.

This chapter has been published in an edited form as [31]:

- M.J.Chorley and D.W.Walker (2010), Performance Analysis of a Hybrid MPI/OpenMP Application on Multi-core Clusters. In Journal of Computational Science, 1 (3). **doi: 10.1016/j.jocs.2010.05.001**.

The performance of the hybrid DL_Poly application is considered on two multi-core systems: one utilising an Infiniband connection, the other utilising a 10 Gigabit Ethernet connection.

With this large scale MD code the pure MPI version is again found to provide the better performance at low processor numbers. However, at higher processor numbers the hybrid message passing + shared memory code performs better on both the 10 Gigabit Ethernet and the lower latency higher bandwidth Infiniband connection.

1.4.3 Performance Analysis

The work presented in Chapter 6 aims to analytically describe the performance differences observed between the hybrid and pure MPI codes, generalising results so that suggestions may be made about the performance of the model on future generations of hardware.

Summary

This chapter has introduced the thesis topic, and the motivation for the work carried out in the dissertation. It has described the aims and contributions of the thesis, as well as the thesis structure. The following chapter will examine the background to the work contained in the thesis.

Chapter 2

Background

Overview

This chapter discusses the background topics and previous work in HPC relevant to the research presented in this thesis.

It begins in Section 2.1 by discussing some basic theory of programming an application in parallel, before then introducing the two dominant styles of parallel programming in HPC: Message Passing (MP) programming and Shared Memory (SM) programming. The *de facto* standards for these programming styles, MPI [84] and OpenMP [92], are then introduced and compared, with the advantages and disadvantages of both explored.

Section 2.2 then discusses issues surrounding multi-core clusters and HPC hardware. Reference is made to typical HPC interconnection networks used in clusters, with particular focus on the interconnects found in the multi-core clusters used in this work: 1 and 10 Gigabit Ethernet, Infiniband and Infinipath. Multi-core processor technology is examined, and the differences between multi-core and Symmetric Multi Processing (SMP) hardware are shown.

Combining message passing and shared memory programming models to create hybrid message passing + shared memory code is then examined along with previous work on the subject in Section 2.3. Previous work on the use of the model on clusters of both SMP hardware and multi-core nodes is examined with reference to the relevance of the work to this thesis.

Issues pertaining to the performance modelling of parallel code are discussed in Section 2.4,

with particular reference to the modelling of message passing communication.

Finally Molecular Dynamics codes are introduced in Section 2.5, with the typical structure of such codes presented.

2.1 Parallel Programming

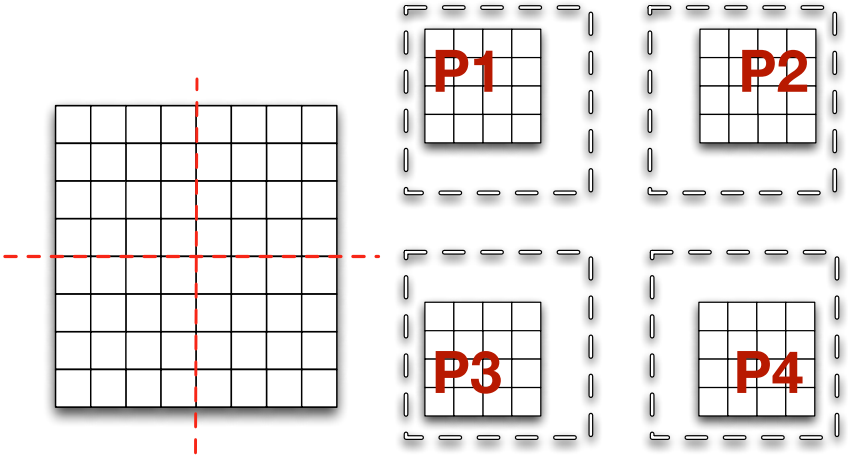
In order to understand the operation of the applications used in this thesis it is necessary to introduce some basic concepts of parallel programming. Parallel programming is the term used to describe the process of writing an application that may be run on more than one processor. The aim is that by increasing the number of processors working on completing a specific task the time taken to complete this task will be reduced, as portions of the application can be executed in parallel with one another.

When implementing an application in parallel, the application must be broken down so that separate portions may be executed at the same time. This may be done via task parallelism in which the separate tasks undertaken by the application are executed concurrently, or it may be done via data parallelism in which the data to be operated on is broken into separate sections and each section of data is operated on at the same time. The data parallelism style of programming is "the most common strategy for scientific programs on parallel machines" [41]. Scientific applications frequently operate on large domains of data, and the separation of this data into distinct parts is an efficient way of allowing the application to run in parallel. The applications examined in this thesis all use the data parallelism strategy, so this is introduced next.

2.1.1 Data Parallelism

The data parallelism style of parallel programming is concerned with dividing the data domain of an application into smaller sub-domains, each of which may be operated on at the same time by a separate processor, as illustrated in Figure 2.1, in which a data domain, (here a set of cells to be processed, 2.1a) is divided into sections and allocated to four separate processors, P1, P2, P3 and P4 (2.1b).

Often a processor will need to read data from the neighbouring sub-domains in order to correctly calculate results within its own sub-domain. A data-parallel decomposition strategy will therefore often involve some form of communication between individual processors so that



(a) Data Domain. (b) Decomposed and Allocated.

Figure 2.1: Domain Decomposition - a domain of cells is divided into sub-domains, each of which is assigned to an individual processor.

they may have access to all the data required to perform their task. The data from neighbouring processors will need to be kept up-to-date, so multiple communications between processors may be needed as the application runs. In addition to this swapping of data between individual processors during a run of an application, there may also be times when all processors need to communicate as a group in order to calculate some global statistic, or synchronise some value. In this case there are many strategies to communicate data globally. For example, a value may be reduced to one process which then broadcasts the new value to all other processes, or all processes may simultaneously communicate with each other. Both 'point-to-point' and 'collective communication' are discussed further in Section 2.1.3.

The communication between processors in the parallel application may be either explicit via message passing, or implicit via shared memory, as described in the next section. This results in differences in implementation details but the overall parallel strategy remains the same. In a message passing implementation each processor will maintain its own copy of the data that surrounds its own subdomain but which resides on neighbouring processes. This data is usually termed 'halo data'. This halo data will need to be updated through explicit message passing as the simulation progresses. In a shared memory implementation, each processor will access the halo data it requires through direct access to the memory required, so there is no explicit

communication.

A parallel application using the data parallelism strategy will usually exhibit a regular sequence of behaviour, where each processor performs calculations on some data, communicates updated data to its neighbours and receives updated data itself, then performs more calculations on its data. This pattern of a calculation phase followed by a communication phase is commonly seen in parallel applications, including those used in this research.

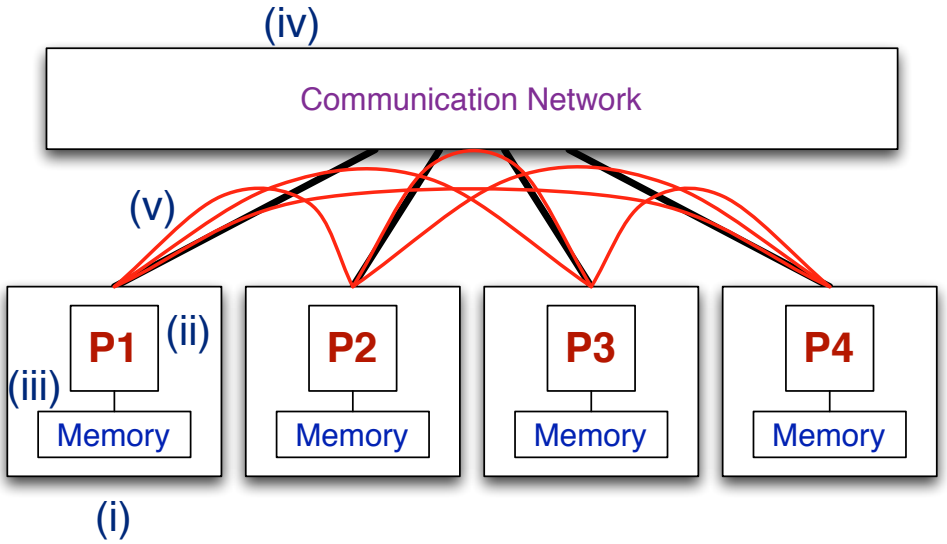
2.1.2 Message Passing and Shared Memory Programming

> "For the past decade, those developers who have focused on workstation and SMP solutions have threaded their programs, while those interested in supercomputing solutions have switched from vector to message passed programming" [74].

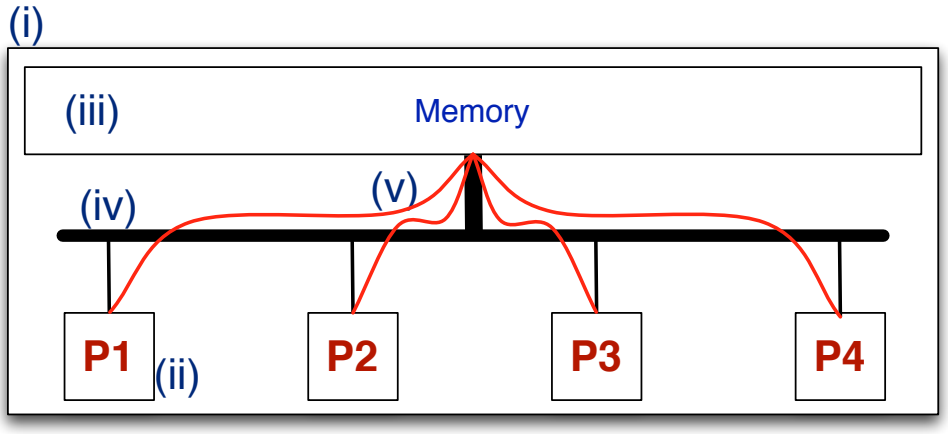
HPC has in recent years been dominated by two styles of implementing a parallel application: Message Passing (MP) and Shared Memory (SM) programming, each of which is closely related to a different class of parallel hardware.

1. Message Passing programming is associated with Distributed Memory (DM) machines, where independent nodes are linked together via a communication network (a cluster architecture). In this scenario separate processes are run in parallel on each node or processor and use explicit communication through messages in order to coordinate their work.

2. Shared Memory programming is closely associated with Shared Memory (SM) machines, where several processors within the same logical system share access to a memory space. This scenario typically involves using threading within an application, and communication between threads is implicit. Threads primarily access those portions of the shared memory relevant to themselves, but also have access to the shared memory 'belonging' to the other threads, allowing synchronisation and coordination of the parallel task.



(a) Distributed Memory - individual processors within nodes communicate through explicit messages over a communication network.



(b) Shared Memory - individual processors access the same shared memory space, allowing implicit communication.

Figure 2.2: Distributed Memory and Shared Memory.

The two styles of programming and the hardware with which they are closely associated are illustrated in Figure 2.2. Firstly, Figure 2.2a shows an example of distributed memory hardware, in which individual nodes (i), each containing a processor (ii) and local memory (iii) are connected together via a communication network (iv). In order to communicate, the processes running on each node send explicit messages (v) to each other through the communication network. Figure 2.2b shows an example of shared memory hardware (i), in which individual processors (ii) are connected to a large shared memory (iii) via a shared bus connection (iv).

Communication is carried out implicitly, as a thread running on one processor will have access to the data it requires in the shared memory space (v).

The *de facto* standards for programming in Message Passing and Shared Memory styles in HPC are MPI [84] and OpenMP [92], as introduced in the following sections. Both have become dominant in their areas, although other alternatives are available. Global Arrays [89, 88] can be used to provide shared memory constructs across distributed memory clusters, providing for concepts such as single sided communication (where only one process is explicitly involved in communication) and for large arrays of data to be shared between remote processes. Unified Parallel C (UPC) [114] offers a similar ability to carry out shared memory threading over distributed memory hardware. Shared memory programming may also be accomplished using threading libraries such as Posix threads (pThreads)[85], or Intel Threaded Building Blocks (TBB)[94]. Some take issue with the use of libraries for threading however: [16] suggests that the approach taken by pThreads in implementing threads in a library is not ideal, as issues concerning under-specification of behaviour lead to correctness depending on the implementation specifics. A further option for running shared memory code on distributed memory clusters may be automatic translation from OpenMP code to MPI [13].

2.1.3 MPI

> "Message Passing Interface (MPI) is a widely accepted standard for writing message passing programs." [68]

The Message Passing Interface (MPI) standard [84] defines a set of library routines and data types that can be used to implement the Message Passing style of parallel programming within an application. Use of the standard is widespread throughout HPC and many parallel codes use it to enable their parallel operation. The standard specifies the routines that should be available within an MPI implementation, their behaviour as part of a parallel application, and the data types that may be used between MPI processes, but does not go into detail about the

implementation. Thus it does not matter how an MPI library implements the communication, merely that it behaves as the standard requires.

MPI usage is fairly straightforward and follows a similar outline pattern in most parallel applications. Basic initialisation of a set of MPI processes and the associated MPI environment is done through a call to the `MPI_Init` routine within the application code; this routine must be called before any other MPI routines are used. This initialisation method sets up the default MPI communicator `MPI_COMM_WORLD`, which contains the context for the set of processes running in the parallel application. Each process within the communicator is assigned a unique identifying 'rank' number that can be used to identify processes within MPI routines. The initial call to `MPI_Init` is matched by an `MPI_Finalize` call at the end of the application code, which closes down the MPI environment, and after which no MPI routines may be used. Between these two calls, application code may call other methods from the MPI library to carry out message passing communication between processes.

There are two main classes of communication defined by the MPI standard which are relevant to this work: Point-to-Point and Collective communication.

Point-to-Point communication involves one process sending data to one other receiving process. This may be achieved in the simplest case through the use of `MPI_Send` and `MPI_Recv` calls. The sending process will initiate an `MPI_Send` call, specifying parameters such as the data to be sent and the rank of the destination process. The receiving process will call the `MPI_Recv` method to receive this data, specifying the maximum amount of data it can receive, and the rank of the sending process it is expecting to receive data from. This is the simplest type of point-to-point communication, although other routines are available to provide for synchronous/asynchronous and blocking/non-blocking communication.

Collective communication typically involves all processes within a parallel application communicating simultaneously. This may be done to broadcast a value to all processes (using `MPI_Bcast` for example) or to reduce a value (using `MPI_Allreduce` for instance) across all running processes. Global communication may also be used to synchronise processes using

a call to `MPI_Barrier`. It is worth noting that most collective communication is expensive in terms of communication time, as all processes will need to be synchronised in order to carry out the communication, and imbalance between the running time of the processes will cause a hold up to all other processes. For example, as [76] says: "The complete data exchange collective, MPI_Alltoall is one of the most intensive communication patterns used" [76]. The expense (in terms of performance) of collective communication leads to a desire to reduce its usage within parallel message passing applications.

It is worth noting the differences between point-to-point and collective communication as they apply to the requirements of HPC interconnects: "collective communication requirements are strongly differentiated from point-to-point requirements. Collective communication ... tended to involve very small messages that are primarily latency bound. As the numbers of cores increases, the importance of these fine-grained, smaller-than-cache-line sized collective synchronisation constructs will likely increase." [9]. So, although collective communication in general involves small messages that do not require much bandwidth to transmit, they are bound by the latency of the connection - that is the ability for the interconnect to setup a connection from one process to another. As the number of MPI processes involved in a collective communication increases, this can have a significant effect on performance. This is an important factor even as interconnects improve, as "latency is likely to improve much more slowly than bandwidth" [9]. This indicates that as the number of processing cores per node increases, the performance of collective communication is likely to worsen, if connection latency does not also improve. For point-to-point communication however, latency is less of an issue: "The sizes of most point-to-point messages are typically large enough that they remain strongly bandwidth bound, even for on-chip interconnects" [9]. This means that for most point-to-point messages the latency of the interconnect is not as important, rather it is the bandwidth of the connection between processes which has the larger effect on performance.

While MPI specifies many other types of communication such as Single Sided Communication and Parallel Input/Output, these are not discussed here as they are not relevant to the work presented within this thesis.

An important issue when using MPI in a hybrid code is that of thread safety, and the behaviour of MPI library calls when called from within a multi-threaded environment. The MPI standard defines an initialisation method (`MPI_Init_thread`) for specifying the level of thread support required, allowing application programmers to explicitly request that an implementation behave in a thread safe manner. There is no guarantee however that an implementation will actually provide this level of support. Much work has already been done examining the issues of multi-threading and MPI. In [113] a test suite is described for evaluating the performance of multi-threaded MPI communication, looking at the specific case of MPI communication in hybrid message passing + shared memory codes, where MPI calls may be made from different threads. It notes that while there are overheads associated with maintaining thread safety, a slow interconnect "such as GigabitEthernet masks some of these overheads" [113], highlighting that slower data transfer may hide the increased overheads of ensuring thread safety. Although this indicates that a slower interconnect may actually be a benefit in some cases when using a hybrid message passing + shared memory model, it does suggest that when using a faster interconnect the overheads of ensuring thread safety will be exposed in the code performance, which could be a significant disadvantage of the model. Both [50] and [11] examine how the MPI standard relates to thread safety issues, again finding that requiring thread safety adds extra overheads to MPI operations. However, both papers ([50] and [11]) present discussions on limiting these overheads and reducing their impact within message passing code, suggesting that it is possible to do this. It is worth noting that the application codes examined in this thesis do not require MPI calls to be made from within different threads, so theoretically they do not explicitly require a thread safe MPI implementation to operate correctly. However in practice they do actually need the MPI implementation to support multithreaded environments to enable correct operation, as some implementations will restrict the spawning of shared memory threads if explicit multi-threading support is not requested by the programmer.

MPI is designed primarily for distributed memory systems where each node contains a single processor. On multi-core systems, "many scientific applications today are written in MPI using a one-process-per-core model that partitions memory among the cores" [39]. This is the most

natural progression when taking a pure MPI application onto a multi-core system, requiring no changes to the application, and treating the multi-core node exactly the same as a collection of single core nodes.

Although MPI is designed for implementing a Message Passing style of programming with processes communicating through some type of networking interconnect, on systems where shared memory nodes are available many MPI implementations will actually use communication through the shared memory for intra-node communication. When MPI is able to use direct shared memory access within a compute node to carry out communications, performance can be dramatically improved [18] as the communication will not need to be transferred over the network interconnect, removing the overheads of both establishing a connection and transmitting the data.

There are many MPI implementations available: among the most popular are MPICH [49] and OpenMPI [47]. The MPICH implementation of the MPI standard is the basis for many vendor specific MPI implementations [17, 64], which may include further fine tuning and performance enhancements specific to vendor hardware. As implementation details vary from implementation to implementation, performance differences may be observed when using the same application with different MPI libraries [38]. Communication time is in fact dependent on many software and hardware parameters [78], which can make the tuning and understanding of performance issues a difficult task.

Advantages

> "explicit parallelism often provides a better performance and a number of optimised collective communication routines are available for optimal efficiency. Data placement problems are rarely observed and synchronisation occurs implicitly with subroutine calls and hence is minimised naturally" [104].

MPI has many advantages that have made it the *de facto* standard for message passing programming on HPC clusters. The specification defines interfaces for both C and Fortran,

two of the most widely used languages in HPC. The specification itself provides a large number of message calls to suit particular needs, covering synchronous and asynchronous communication, blocking and non-blocking sends and receives, single-sided communication, parallel IO, separate or joined send and receive calls and synchronisation constructs such as broadcast, reduce and all_to_all. A large number of parameters for each message routine allow fine grained control over communication between processes.

As stated above from [104], MPI implementations are optimised for typical operations, and the style of message passing programming naturally leads to a more efficient style of programming with regard to synchronisation.

The advantages of the MPI specification itself have led to further advantages that stem from the dominance of MPI within the HPC ecosystem. Because of its wide use in the scientific computing community it is supported well on HPC clusters by hardware vendors, compilers and systems support staff. As a widely used specification there is a large knowledge base in the community to rely on when coding, testing and running MPI codes. The performance problems are mainly well understood, and for many tasks there is already a best practice to utilise MPI to solve a problem. MPI is now well supported by code profiling tools and debuggers, aiding in the coding process. The separation of communication routines from computation allows both to be optimised independently of one another [98].

Disadvantages

One of the immediate drawbacks of MPI is that it presents the novice programmer with a steep learning curve in order to parallelise a code. Much of the parallelisation work must be done by the programmer, who is responsible for the whole process of work sharing, from the division and sharing of data across the processes to the synchronisation and gathering of results. “MPI has evolved as the dominant library, but enormous, assembly-language style effort is required to develop MPI programs” [74]. The amount of control given to the application programmer by its low-level approach is also a disadvantage, as it makes the library more complicated and

prone to errors.

When moving from an existing sequential application to a parallel application it may be hard to relate the resultant parallel application to the sequential version: "[a]n arbitrary MPI program may not have much resemblance to a sequential program from which it was derived" [74]. Significant effort may be required in order to re-purpose a parallel application to use MPI, as the library "typically requires a thorough (re-) design of a parallel application" [21].

Explicit communication adds overheads to the running of the application, and as already discussed, collective communication can be very time consuming indeed. There is also an issue that "[d]ecomposition, development and debugging of applications can be time consuming and significant code changes are often required." [104].

Data Decomposition and Message Passing

In order to understand how the applications used in this thesis communicate data, and how that relates to the performance of both the pure message passing and hybrid shared memory + message passing models, it is necessary to explain the typical methods of decomposition and communication of data within a message passing application using a data parallel model.

Given a 2 dimensional domain of data, with width w and height h, and a 2 dimensional mesh of processors with width x and height y it is clear that when this is decomposed over the processors being used to run the application, each processor will end up with a subdomain of approximately $\frac{w}{x} \times \frac{h}{y}$, provided that the domain may be decomposed so that $\frac{w}{x}$ and $\frac{h}{y}$ make sense within the logic of the application. If the domain sizes are multiples of the processor mesh sizes, this is usually not an issue.

Similarly for a 3 dimensional decomposition, if the data domain has width w, height h and depth d, and it is decomposed onto a 3 dimensional mesh of processors of width x, height y and depth z, then each processor will have a subdomain of data of dimension approximately $\frac{w}{x} \times \frac{h}{y} \times \frac{d}{z}$.

When carrying out a data decomposition within a parallel application written in a message

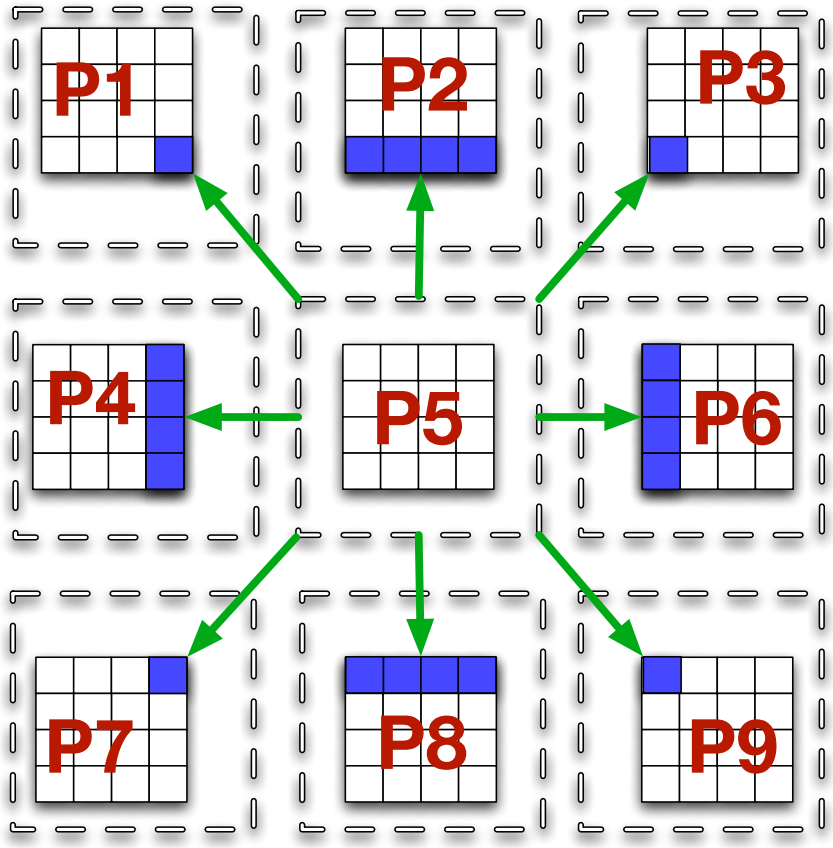


Figure 2.3: Data Required from Neighbour Processes - P5 requires knowledge of the data belonging to the subdomains on the neighbouring processes in order to complete the work for its own subdomain.

passing style, it is important to ensure that each process has access to the data it needs during the application run. This accessing of data belonging to other processors is shown in Figure 2.3, simplified to the two dimensional case. Here, in order to carry out its work, processor P5 needs the neighbouring data in the cells (highlighted in blue) belonging to the other 8 processors P1, P2, P3, P4, P6, P7, P8 and P9. This data will also need to be kept loosely coherent as the application progresses, so that it is current at the points in the algorithm where it is needed. Each processor will keep a local copy of the data from its neighbours (halo data), and this will be updated periodically throughout the application run.

The updating of required data follows a simple pattern. Sets of messages are exchanged, with all processes communicating in the same direction at the same time, enabling all data to be updated with a minimum of communication. This process is illustrated in Figure 2.4. Here, processes communicate data in four separate phases in order to ensure that each process has an

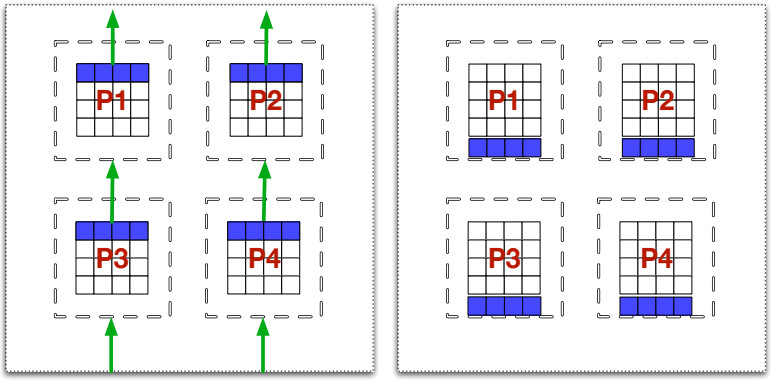
up to date copy of the halo data it requires. Firstly, each process sends data from its top most boundary up to the receiving process above (Figure 2.4a) resulting in each process having a copy of the boundary data from the process below (Figure 2.4b). The second phase involves similar communication, this time in the down direction (Figure 2.4c), resulting in each process having a halo copy of the data from both the process above and the process below (Figure 2.4d). Data is then sent from each process to the process to its right. This time however, as well as the boundary data from the process' right most edge, data from the rightmost edge of the upper and lower halo data is also sent (Figure 2.4e). This enables each process to have not just the data for the left halo copy, but also for the upper and lower left corners (Figure 2.4f). This process is then carried out again, in the left direction (Figure 2.4g), finally ending with all processes having a copy of all the halo data they require, from the processes above and below, to the left and the right, and from the upper and lower left and right corners (Figure 2.4h).

This describes the simplified two dimensional case, but the same process occurs for a three dimensional decomposition. A three dimensional decomposition will have an extra set of messages for communicating data in the extra dimension.

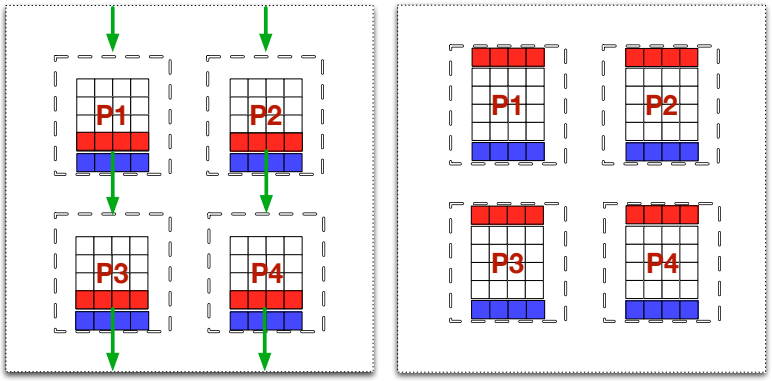
Communication between processes when carrying out updates of halo data in three dimensions can be described in terms of the size of their subdomains. If a process has a subdomain of width sd_w, height of sd_h and depth of sd_d, and a halo depth of h_d, and it follows the procedure above for communicating and updating halo data, then the following holds:

1. A process can first send data to the processors to the left and right of itself, sending a halo of size $(sd_h \times sd_d) \times h_d$, while also receiving the same size of halo from the neighbouring processors.

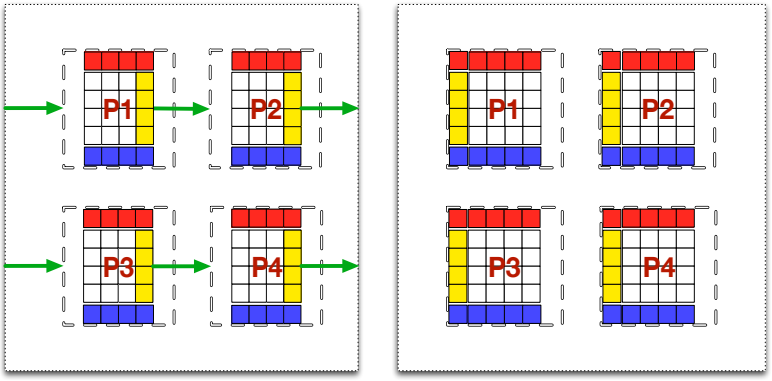
2. It can then send data to the up/down neighbours, comprising its own halo of size $sd_w \times sd_h$, plus a portion of the left/right halo data it received from the left/right neighbours, giving a total size of $(sd_w \times sd_h + (2 \times h_d \times sd_h)) \times h_d$. As before, it also receives this amount of data from its up/down neighbours.



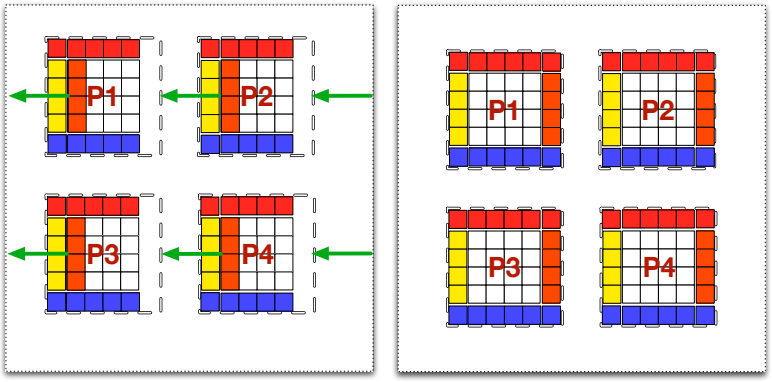
(a) Up Message (b) Up Message Received.



(c) Down Message. (d) Down Message Received.



(e) Right Message. (f) Right Message Received.



(g) Left Message. (h) Left Message Received.

Figure 2.4: Data Parallelism - Halo Data Communication.

3. Finally it may send data to the north/south neighbour processors. This message comprises a halo of its own data of size $sd_w \times sd_d$ plus additional data from the two sets of halos it has already received, giving a total message size of $(sd_w + 2 \times h_d) \times (sd_d + 2 \times h_d) \times h_d$.

These three steps, comprising six distinct messages, will ensure that each processor has a copy of the correct halo data it needs to carry out calculations on its own subdomain.

2.1.4 OpenMP

> "OpenMP is probably the most commonly used communication standard for shared-memory based parallel computing" [21]. "OpenMP is becoming the standard programming model for shared memory parallel architectures" [45].

The OpenMP Application Programming Interface (API) [92] defines a set of library routines and compiler directives that facilitate parallelisation on shared memory systems using threading. A newer standard than MPI, it specifies some library routines for general tasks (such as discovering the number of threads in a parallel application) but relies primarily on compiler directives to parallelise a code. These directives are translated at compile time into parallel code by a compatible compiler, which will produce the threaded application code. It is now a widely accepted standard for "annotating programs for parallel executions" [66]. The bulk of directives (in OpenMP 2.0) are related to work sharing constructs and synchronisation.

In order to use OpenMP the programmer specifies which sections of the code should be carried out in parallel through the declaration of parallel regions, using the `omp parallel` and `omp end parallel` directives. Within these regions, the programmer can then specify the distribution of work via constructs such as the sharing of iterations in a loop (using an `omp for` directive), or (if using OpenMP 3.0) the specification of tasks that may be carried out concurrently. Data access may be controlled also, allowing variables to be declared as private to a thread or shared between all threads. Other constructs allow for the synchronisation of threads at barriers (`omp barrier`), and for other useful operations such as the copying in of data

values to a parallel region, or the reduction of values across a group of threads. OpenMP also allows nested parallelism (i.e. threads spawned from within other threads, which may match underlying hierarchical architecture features in multi-core systems), but the performance may be reduced using this style [37].

As with MPI, the use of OpenMP introduces overheads to a parallel code. In contrast to MPI, these overheads are not generally explicitly incurred through calls to external libraries, rather they are inherent within the operation of the shared memory threading. Many operations will be carried out as part of the parallel operation of the code which will consume run time. Threads must be forked and joined at the start and end of each parallel region. Synchronisation may also need to be carried out while an application is running, as may reduction of global variables. Some shared memory related constructs such as locking of variables for atomic updates can be very expensive in terms of runtime. Various systems and toolkits exist to characterise these overheads caused by OpenMP, such as CLOMP [19] or the EPCC micro-benchmarks [22].

While OpenMP is designed primarily for shared memory hardware, distributed shared memory versions do exist, but these do not provide the same level of performance as pure MPI on distributed memory hardware [19].

Advantages

A clear advantage of OpenMP is its high level nature which makes it much easier to perform simple parallelisation of a code [75]. The low level implementation is done by the compiler; this allows the programmer to specify *what* to parallelise, without having to specify exactly *how* it is parallelised. The API defines a large range of directives and routines to use in a parallel code, allowing a vast range of codes to use OpenMP for parallelisation. The API defines the exact behaviour of routines and directives in both C and Fortran, appealing to the wide C and Fortran code base already in use in HPC. The ease of use is pointed out in [21]: “OpenMP ... can be easily adapted to existing sequential software”.

Simple parallelisation of code using OpenMP is much more straightforward than using MPI,

with far less programming effort required. OpenMP is also overall much easier to use than 'hand-threading' using pThreads [75]. Code parallelised using OpenMP can be easily made sequential by simply turning off the OpenMP compilation option.

Many tools exist for debugging OpenMP, and the nature of the shared memory threading makes it easier to debug and tune than a distributed application [75] such as one using MPI, or a shared memory application using pThreads or similar mechanisms. [74] has found that: "OpenMP performance profiling can be done more easily than for performance profiling of a program making manual calls to a threads library", and that for OpenMP: "current tools can very accurately locate threading correctness problems" [74]. These examples point to the advantage OpenMP applications have over distributed MPI based applications when debugging and maintaining application code.

OpenMP also has some speed advantages; on some architectures, OpenMP barriers are faster than the equivalent MPI barriers, and broadcasting data can also be much faster [60].

Disadvantages

Although simple parallelisation is very straightforward with OpenMP, it can be much harder to get very good performance when a more complicated strategy is required. It may also be hard to debug problems such as false sharing caused by an incorrect parallelisation strategy. The shared memory model is only scalable as far as the shared memory hardware, and this disadvantage is carried into OpenMP. The knowledge base for OpenMP among the HPC community, while growing, is still smaller than that of MPI. Additionally there may be performance problems caused by OpenMP itself. Synchronisation of OpenMP threads may cause issues relating to cache that may slow down an application [98], and furthermore OpenMP may have performance problems on cache-coherent Non Uniform Memory Access (ccNUMA) hardware [20]. The use of OpenMP during compilation may stop the compiler carrying out certain loop optimisations which may affect code performance [52].

2.1.5 MPI vs. OpenMP

Understanding how MPI and OpenMP compare in terms of performance is an issue frequently discussed in the literature. Much comparison of MPI and OpenMP has been done (e.g [27, 15, 8]), with no real consensus reachable as to the better model to use. In any given situation, best performance is dependent on many factors, most importantly the hardware and algorithms under consideration.

2.2 Multi-core Clusters

> "Commodity off the shelf (COTS) clusters, driven in the mid-1990s by Beowulf clusters, have grown in popularity and performance as microprocessor-based clusters have been enhanced by architectural ideas that were pioneered by earlier custom systems" [74].

> Clusters are "perhaps more widely used than any other type of parallel computer because of their low cost, flexibility, and accessibility" [43].

As [43] and [74] highlight, a significant trend within the HPC community over recent years has been away a move away from large Massively Parallel Processing (MPP) machines based on proprietary chip technology and software, and a move towards clusters of standard PCs or workstations using off-the-shelf components and open source software. This cluster architecture has come to dominate the field: the June 2010 Top500 ranking[1] of the world's supercomputers shows that 424 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 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. The introduction of multi-core processors has enabled

[1] http://www.top500.org

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 within these systems.

These clusters are normally built from 'standard' hardware. 'Nodes' containing one or more processors, along with local memory and (optionally) some local storage are linked through one or more networks. These nodes are typically built from commodity components, usually many-socket motherboards primarily designed for the rack-mounted server market, coupled with commodity processors, memory and hard drives. Many clusters include dedicated high-speed networks such as Infiniband for communication between nodes and network file storage access, while also containing slower networks such as Gigabit Ethernet for tasks such as cluster management.

Not only is the cluster now the dominant HPC architecture, but "almost all current high performance computing systems now contain nodes which consist of shared memory multiprocessors" [23]. The vast majority of these shared memory multi-processor nodes will be multi-core processors. These multi-core clusters can have significant effects on the performance of codes: "application runtimes on large High-Performance Computing (HPC) systems with dual- or quad-core chips are slower when compared, core for core, with single-core chip performance" [103]. To compensate for the effects of multi-core processors on code performance may require making architectural changes to application code: "users should examine their codes and consider restructuring them to increase locality, increase intra-node communications, assign MPI ranks to promote spatial locality, use compiler optimizations and make other multicore aware changes" [103].

Recent advances, such as 64-bit architectures, multi-core and multi-threading processors [2] and software multi-threading [87], can all be used to benefit the scientific computing community. These technological advances are leading to the dividing lines between HPC architectures becoming blurred. As discussed, previous HPC architectures 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 (SMP's). Systems in which the nodes themselves are identical SMP systems introduce another level of complication, while the introduction of multi-core processors has yet further increased this complexity. As the number of processing cores within processors 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 multi-core 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.

2.2.1 Multi-core Processors

> "Multicore architectures integrate multiple processing units into one chip to overcome the physical constraints of unicore architectures, and their exponentially growing power consumption" [110].

As it has become increasingly hard to pack more transistors into a single processor and increase the raw power of the chip, manufacturers have turned to putting multiple processor cores on one chip to increase performance. This has had a large effect on HPC: "Multi-core processors

are beginning to revolutionize the landscape of high-performance computing" [3]. The design of multi-core processors is hoped to "sustain performance growth while depending less on raw circuit speed" [3].

Much work has been done examining the effects of multi-core processors on the performance of scientific applications within HPC. Multi-core processors have many performance effects on codes, increasing performance of some, and decreasing the performance of others [102]. Much as with the discussion over which is best, OpenMP or MPI, the effect of multi-core processors on code performance is highly dependent on the class of application and algorithms being used.

There are significant architectural differences between single core and multi-core processors that can affect code performance. Memory access can be a problem due to the multiple levels of cache and memory access latency [110], as found in many studies: "cache and memory contention may prevent the multi-core system from achieving best performance, because two cores on the same chip share the same L2 cache and memory controller" [29]. In [103] it is agreed that "memory bandwidth contention is the single most important cause of multicore performance degradation" [103].

Attempts can be made to overcome the performance problems caused by multi-core processors, focusing on areas such as the intra and inter-node communication in an application, and the placement of processes or data within a node or across a cluster. Solutions such as "reducing network overhead by more logical process placement, updating or optimizing multi-node I/O, and working with more distributed and shared-memory programming models" [102] may help performance. Similarly, "overlapping working sets in threads running on cores sharing a cache can improve runtime" [93]. The importance of the communication on overall performance is highlighted: "intra-node communication is as important as optimizing inter-node communication in a multi-core cluster" [29]. These all serve to show that to get increased performance using multi-core processors is not as simple a task as simply deploying an existing parallel application on to a multi-core machine, but that significant engineering effort may be required to change the application structure and operation to gain best performance on a multi-

core cluster.

Despite the performance drawbacks and increased effort in engineering applications for multi-core systems, multi-processor systems may still exhibit larger performance improvements than single processor systems [70]. Awareness of cache affinity can have a significant effect on performance when using multi-processor multi-core systems, and multi-processor multi-core systems can increase application performance above the increase from multi-processor systems alone [70].

Some aspects of poor performance of applications on multi-core clusters are due to the processor designs themselves; multi-core processors "do not take advantage of the proximity between cores to improve synchronisation and communication between concurrent threads. Thread synchronization and communication instead use memory/cache interactions" [46]. This suggests that performance could be further improved if individual cores within a chip were able to directly communicate with one another, rather than requiring communication through the cache or memory.

Overall system design also has a part to play when multi-core processors are used. A cc-NUMA architecture may reveal performance problems: "it is hard to obtain high performance on the latter [cc-NUMA] architecture, particularly when large numbers of threads are involved" [30].

2.2.2 Multi-core Nodes vs. SMP Systems

> "[P]rogrammers will not be able to consider these cores independently (i.e. multi-core is *not* "the new SMP") because they share on-chip resources in ways that separate processors do not" [42].

The distinction between SMP and multi-core systems is an important one. In an SMP system, a number of distinct physical processors access shared memory through a shared bus, with each processor having its own caches and connection to the bus. Multi-core processors however 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 [4]. Cache sharing may introduce performance problems, as "access by a core to a memory block limits concurrent access to the same block by other cores" [3]. The increase in the numbers of cores causes issues with "data-locality, shared cache, bus contention and memory bandwidth limits ... due to increases in resource sharing" [4]. These problems with multi-core processors (as discussed in the previous section) can also occur with SMP systems, so while cache contention is not an issue in an SMP machine (as the processor cores do not share cache), there may still be issues related to shared bus access and memory bandwidth limits.

SMP systems typically suffer from a lack of scalability. There is an upper limit to the number of cores you can attach to a shared bus before the bandwidth constraints become a limiting factor. "In SMP's, contention due to the shared bus located between the processor's L2 cache and the shared main memory subsystem adds additional delay to the memory latency" [77]. There are solutions to this problem such as Non-Uniform Memory Access (NUMA) systems or cache-coherent Non-Uniform Memory Access (cc-NUMA) systems, where a global memory is partitioned between sets of processors, but these systems are still limited in scalability. SMP systems will normally contain a number of cores measured in the tens. One term used to describe such SMP nodes (with somewhere in the region of 16-128 cores) when used in a cluster system is a 'fat node'. These SMP nodes will typically have very large memories too, measured in the tens of gigabytes.

Nodes of a multi-core cluster will normally contain many fewer cores than SMP nodes. Current typical nodes may contain one or two dual or quad core processors (although hex-core is now a distinct possibility), so a node may have anywhere from 2 to 8 cores, and will usually have less memory than a large SMP node. These nodes are considered 'thin nodes', compared to the 'fat node' SMPs.

This distinction between multi-core nodes and traditional SMP systems obviously becomes harder to define as SMP systems start to contain multi-core processors, and multi-core

processors become many-core, with physical chips containing many tens of processing cores.

2.2.3 HPC Interconnects

> "An important part of modern supercomputing platforms is the network interconnect" [72].

A major contributory factor to the performance of an HPC cluster is the performance of the interconnect used to connect nodes together. There are many types of interconnect in use today, ranging from low end 1 Gigabit Ethernet (GigE) through 10 Gigabit Ethernet (10GigE) and Infinipath connections to high end Infiniband interconnects, and performance of the different network interconnects is key to their adoption in modern HPC systems: "Modern interconnects such as Infiniband, Quadrics and Myrinet have become popular due to their low latency and increased performance over traditional Ethernet" [72]. Work has already been done comparing the performance of message passing communication such as MPI over these interconnects [79].

Gigabit Ethernet

Gigabit Ethernet (GigE) is a networking standard (IEEE standard 802.3z), commonly used as a network interconnect in a Local Area Network (LAN). However, as it "offers one gigabit per second raw bandwidth" [91] it is also suitable for use within HPC clusters. In [91] the standard is examined and performance problems of TCP/IP over GigE, and of the MPICH library over TCP/IP are discussed. While GigE offers a fairly good level of bandwidth for HPC interconnects, the latency may not be as good as in other HPC interconnects, such as Infiniband, Myrinet etc; the so-called "ethernot" interconnects. "Understanding the performance gap between ethernet and ethernot interconnects is a major issue for application developers" [12]. Indeed, "Ethernet-based network architectures such as Gigabit Ethernet (GigE) have delivered significantly worse performance than other system-area networks [e.g, Infiniband (IBA), Myrinet]" [12]. Possibly more suitable as an HPC interconnect is the 10 Gigabit Ethernet

(10GigE) standard, which, as the name suggests, offers 10 gigabits per second raw bandwidth, and improved latency times. "[I]n most experimental scenarios, 10GigE provides comparable performance with IBA and Myrinet" [12]. While GigE is therefore suitable for use in HPC clusters the performance is not as good as the performance of other HPC interconnects. The high latency in particular is of concern for codes where large amounts of collective communication are involved, as collective communication is primarily latency bound. However, 10GigE offers improvements that make it more of an option in HPC clusters.

Infiniband

Infiniband is a proprietary interconnect standard, designed specifically as a high-speed I/O interconnect. Recently, it has become "a popular interconnect for high-performance computing to connect commodity machines in large clusters" [72]. A significant reason for its popularity in such systems is that it offers a large bandwidth and low latency times [79, 73]. Many MPI implementations over Infiniband are capable of exploiting direct memory access [73, 80] for inter-node communication, as well as intra-node communication. The MPI library can therefore directly access the memory of remote nodes, facilitating faster data transfer between the nodes of a system.

As with Gigabit Ethernet, some work has been done examining the performance of MPI over Infiniband. In [79] it is found that the high bandwidth offers an advantage for communicating large messages, while also noting that as the number of nodes used increases, the memory consumption for MPI also increases. On large numbers of cores this increased memory usage could be a limiting factor for the size of application that can be run on an HPC cluster. Reducing the number of MPI processes needed to run on as many cores could therefore reduce the memory usage of the MPI library and so increase the memory available to the user application.

2.3 Hybrid Programming

> "Combining shared-memory and distributed-memory programming models is an old idea... One wants to exploit the strengths of both models: the efficiency, memory savings and ease of programming of the shared-memory model and the scalability of the distributed-memory model" [81].

> "[I]t seems natural to employ a hybrid programming model which uses OpenMP for parallelization inside the node and MPI for message passing between nodes" [52].

> "Hybrid method of parallelization (using MPI for inter-node communication and OpenMP for intra-node communication) seems a natural fit for the way most clusters are built today" [100].

The hybrid message passing + shared memory programming model is often discussed as a method for achieving better application performance on clusters of shared memory systems (for example, see [26, 97, 98, 104]). 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 a consensus as to its effectiveness has not yet been reached. When considered as a whole much of the literature examining the hybrid message passing + shared memory model is contradictory. As with many situations, the applicability and performance of applications using the hybrid message passing + shared memory model is highly dependent upon a number of different factors. Among these factors are the class of application being parallelised, the algorithms used within the application and the type of hardware the application is run on.

Much work on the hybrid model has been done on previous generations of hardware, in particular clusters of SMP systems. However, more recently, some work has been done on clusters of multi-core nodes. This previous work is discussed here.

2.3.1 Hybrid Model Performance

> "There seems to be a general lore that pure MPI can often outperform hybrid, but counterexamples do exist and results tend to vary" [99].

A hybrid MPI/OpenMP version of the Nasa Advanced Supercomputing (NAS) benchmarks is compared and contrasted with the pure MPI versions in [26], and it is concluded that performance depends on several parameters such as memory access patterns and hardware performance, and that "even optimized hybrid codes may provide insignificant performance improvements compared to the original MPI version" [26]. It is also noted that "[f]or all benchmarks, the MPI computation times are less than the MPI+OpenMP ones." [26], suggesting that the addition of OpenMP to the code has actually slowed the computational parts of the code. The specific case of a hybrid message passing + shared memory Discrete Element Modelling (DEM) code is examined in [55], which agrees with [26] in finding that that OpenMP overheads result in the pure MPI code outperforming the hybrid code. However [55] also concludes that that the fine-grain parallelism required by the hybrid model results in poorer performance than in a pure OpenMP code, suggesting that the hybrid message passing + shared memory code is therefore worse than either a pure message passing or pure shared memory code. A direct contradiction is found in [104], where the conclusion of examining a hybrid message passing + shared memory code is that in certain situations the hybrid model can offer better performance than pure MPI codes, but that it is not ideal for all applications. An example using SMP nodes is found in [54], which concludes that performance improvements can be made with the hybrid model, and that memory overhead is drastically reduced in hybrid code compared to pure MPI code. A further example with SMP nodes in [24] shows performance improvements with kernel algorithms but with a significant amount of work needed, and that pure MPI offers better performance with real applications. It concludes that problems with memory bandwidth, cache access and threading overheads result in lower performance. Again an example hybrid application does not perform as well as either pure OpenMP or pure MPI versions in [101], while a Simulated Annealing optimisation example in [36] finds better performance with MPI

+ OpenMP hybrid code than with pure MPI. The hybrid approach has even been used across Grid systems in [119]. Another example is found in [40], again focused on SMP clusters, but suggesting that performance can be improved with the hybrid model. In [67] the hybrid model is found to deliver performance benefits, with the performance gains a result of "inherently lower latency of shared memory threads across processors within a node" [67], while in [107] the load balancing of hybrid CFD codes on SMP clusters is examined. In [105] a hybrid Quantum Monte Carlo code is developed, but only tested on one SMP node, and results are inconclusive. In [53] a positive conclusion is reached on the experience and performance of "developing hybrid MPI and OpenMP parallel paradigms for real applications", and in [68] it is found that a hybrid code performs better than pure MPI over Gigabit Ethernet, but that overall scalability of the code decreases. Again, in [68] one of the NAS parallel benchmarks is examined, and it is found that the hybrid model has benefits on slower connection fabrics. The plane wave Car Parrinello code, CPMD [28] 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, 63]. In [86] the hybrid and pure MPI approaches are found to deliver similar performance, but on large numbers of SMP nodes the hybrid approach outperforms MPI. However, in [44] it is found that the hybrid paradigm is 'inferior compared to a pure MPI parallelization" [44]. This performance is suggested to be because "MPI libraries tend to be highly optimized for message passing communication and provide poor support for thread management" [44], however as discussed in the section on MPI this may not actually be the case. A micro-benchmark suite for analysing hybrid code performance and results of the suite are presented in [23], showing that understanding the performance of the hybrid message passing + shared memory model is of importance to HPC.

Some of the literature is flawed, testing hybrid codes on only one SMP node and so not testing the impact of reduced communication requirements, which is shown to be an important part of the hybrid model performance in much of the other literature (and this thesis). Even ignoring

these studies though there is no consensus in the literature as to the effectiveness of the hybrid message passing + shared memory model. The contrasting set of results and conclusions found throughout the literature show that as with the problem of MPI vs. OpenMP, the performance of the hybrid message passing + shared memory model depends heavily on the algorithm or application being parallelised and the hardware used for testing. There is no simple way to know if a hybrid version of an application will perform better or worse than a pure message passing one without a deep understanding of the performance of the hardware it is to be run on and the structure of the application itself.

2.3.2 Hybrid Code Profiling

It is claimed in [21] that "existing tools typically concentrate on either MPI or OpenMP or exist for dedicated platforms only. It is therefore difficult to get [an] overall picture of a hybrid large-scale application". However, a "framework to automatically analyze and optimize performance of hybrid MPI/OpenMP applications through integration of an optimizing compiler with a performance analysis tool" is proposed in [56]. The interactions between MPI processes and OpenMP threads is examined in [81] and a tool that may be used to examine the operation of a hybrid application is presented. A further tool for a similar purpose including automatic performance analysis is presented in [117]. In [21] the profiling of hybrid codes with Vampir NG is examined. This suggests that tools do exist for profiling hybrid codes, but that perhaps they have yet to reach mainstream use and acceptance in HPC.

As previously mentioned, hybrid codes require MPI implementations to support messages sent from multiple threads [11]. Some prior work has focused on this issue, examining whether hybrid codes function as expected, and that the spawning and joining of threads from within MPI processes operates as desired [81]. A tool is proposed for checking the correct operation of MPI in hybrid applications in [57]. In [113] a test suite that enables the study of the cost of supporting thread safety in MPI implementations is described, and it is noticed that a slower interconnect "masks some of the overhead of maintaining thread safety" [113].

2.3.3 Performance Issues with the Hybrid Model

> "[T]he hybrid model can make ... parallelization more difficult and the performance gains could not compensate for the effort" [99, 82].

There are many issues which can affect the performance of the hybrid model, both positively and negatively, and so can explain the results found in the prior work. The amount of communication required between processes may be reduced in a hybrid code, resulting in a better performance: "With pure MPI, additional intra-node communication is necessary between the MPI processes. With hybrid MPI+OpenMP programming, this intra-node communication is substituted by direct accesses to the application data structures in the shared memory" [98]. However, the hybrid model can also add overheads to a code, resulting in a slower performance: "The OpenMP parallelization inside of the MPI processes may cause an additional overhead. The creation of parallel regions and the synchronisation at their end induces additional work, mainly, if a fine-grained OpenMP parallelization is used" [98]. So, while communication overhead is reduced in a hybrid code (so improving performance) additional shared memory overheads are introduced (so making performance worse).

It may be possible to improve the hybrid model performance by overlapping communication and computation using shared memory threading, however: "Overlapping of communication and computation is a chance for an optimal usage of the hardware, but causes serious programming effort in the application itself to separate numerical code that needs halo data and that cannot be overlapped with the communication therefore, causes overhead due to the additional parallelization level (OpenMP) and communicating and non-communication threads must be load balanced" [97]. Additional programming and code maintenance effort are therefore required in order to maximise the performance of the hybrid message passing + shared memory model, which is obviously not desirable.

[61] describes three key areas of performance difference in hybrid codes: (i) less memory is used with domain decomposition in a hybrid code as fewer domains equals less surface area (or

halos) of the domains need to be stored in memory. (ii) Fewer MPI processes equals less MPI memory usage (more of a factor as core numbers increases). (iii) Collective communication uses fewer processes. As collective communication is latency bound, performance is improved with fewer messages. The work in [61] looks at 3D volume rendering, but finds similar effects on performance as this thesis. However, it only looks at very large numbers of cores (> 1728).

The memory requirements of hybrid codes are something that has been examined many times: "in domain decomposition scenarios, the more MPI domains a problem is divided into, the larger the aggregated surface and thus the larger the amount of memory required for halos. Other data like buffers internal to the MPI subsystem, but also lookup tables, global constants, and everything that is usually duplicated for efficiency reasons, adds to memory consumption" [52]. "Shared memory hybrid parallel programming with MPI and OpenMP avoids partitioning of memory, and, for some applications, provides access to a large enough amount of memory to simulate increasingly large problems" [39]. These papers suggest that the increase in the number of cores per node creates a potential problem with the memory usage of MPI, but that a hybrid model utilising shared memory within a node could alleviate this problem, maximising the problem size that can be examined within a parallel application.

An extra performance effect may occur due to the compiler optimisations and their interaction with OpenMP: "Just by switching on OpenMP, some compilers refrain from some loop optimizations, causing a significant performance hit" [52]. Therefore, optimisations which the compiler can make in the pure MPI code can sometimes not be made in a hybrid MPI + OpenMP code.

A further problem with hybrid codes comes from a software engineering angle, that of code maintainability: "a program that combines multiple programming paradigms is not easy to develop, maintain and optimize" [21].

Some studies suggest that a simple hybrid code, applying OpenMP at the loop level within an existing MPI code cannot deliver improved performance: "conversion of a pure-MPI code to an optimized hybrid MPI/OpenMP code may not be possible by inserting OpenMP `#pragmas`

alone. One may even have to restructure the code to effectively leverage the inter-processor bandwidth via shared memory" [100]. It has also been suggested that this method "can be significantly improved by applying several manual optimizations (loop permutation, loop exchange, use of temporary variables)" [26].

2.3.4 Classifying Hybrid Codes

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 [97, 98], 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. "most of the work on the hybrid OpenMP-MPI programming paradigms addresses fine-grain parallelization" [44]. 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 multi-core nodes. This is the approach used for both the applications in this thesis. 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. While it *may* be possible to get better results using this style, the amount of work effort required is large, and an initial aim of the work in this thesis is to examine if the hybrid model can deliver performance improvements with simple programming effort. This is desirable in many cases: "The struggle is delivering performance while raising the level of abstraction. Going too low may achieve performance, but at the cost of exacerbating the software productivity problem" [9]. "From an existing MPI code, the simplest approach is the incremental one: it consists in OpenMP parallelization of the loop nests in the computation part of the MPI code" [26].

A further taxonomy of parallel programming models on hybrid platforms is given in [52]:

- Pure MPI

- Hybrid Master-Only
- Hybrid Overlapping Communication and Computation
- Pure OpenMP

This taxonomy makes no distinction between a fine-grained OpenMP parallelisation and a coarse grained OpenMP parallelisation. The distinction is only made between hybrid codes where all communication is carried out through the master thread, and hybrid codes where all threads may carry out communication. The code used in this thesis uses a Hybrid Master-Only style, all communication is carried out through the master thread.

In [44] it is claimed that "hybrid models are not efficiently supported by existing MPI implementations, resulting in an imbalanced message passing that is performed solely by the master thread", however this is recognised as an explicit and correct style of hybrid programming in the taxonomy above.

2.3.5 Other MP + SM Combinations

Although hybrid codes are most often created by combining MPI and OpenMP, other approaches have been attempted. A hybrid benchmark code created using MPI and Unified Parallel C (UPC) has been tested in [39], finding that performance improvements are possible using this approach. An example using MPI and pThreads can be found in [83], while in [62] a library is proposed combining MPI and pThreads to reduce intra-node communication. Performance gains are observed when "many intra-node communications are performed" [62]. These approaches are not widespread in the field and are not examined in this thesis, but it seems again that with these other languages/libraries performance is again very dependent upon the specific circumstances in which the model is used.

2.3.6 Previous Work Comparison

As discussed above, much of the previous work focuses on clusters of SMP systems. The work presented in this thesis focuses on clusters of multi-core nodes, which, as mentioned in Section 2.2.2 contain significant architectural differences. Few examples of related work use modern multi-core clusters, and unsurprisingly those that do again disagree on the performance of the model, depending on the application used.

Many of the conclusions in the previous work are based on benchmark suites such as the NAS parallel benchmarks, or the EPCC micro-benchmark suite, which may not reveal the performance of real-world applications. Those results that do come from real application performance do not agree on the performance of the hybrid model, suggesting that performance differs based on the class of application used.

Many studies concentrate on a small number of nodes or processor cores, or a large number of nodes and processor cores, without examining the full range. This thesis finds that results differ as the number of cores is increased.

Although some prior work notes that a slower interconnect may benefit the hybrid model, none fully compare the effect of the interconnect on the model performance across a full range of node and core counts. This is addressed in both performance comparisons carried out in this thesis.

2.4 Performance Modelling

> "Efforts to base performance evaluation and benchmarking on realistic applications exist, today. However, they are often overlooked, as we are still used to considering metrics that are based on kernel benchmarks or even raw machine performance" [7].

> "Performance modelling of parallel applications can aid us in the understanding

> and creation of parallel codes, and allow us to predict future performance of these codes on newer generations of hardware" [51].

Many studies have already been carried out attempting to model the performance of message passing communication. A model of communication timings of MPI over distributed memory systems is given in [78]. This model, while very detailed, requires a large amount of effort gathering parameters, and is limited to codes with regular access patterns. Several well accepted (simpler) models exist for modelling communication, such as the Hockney model [58], the LogP model [34] and its extensions, the LogGP model [5], and the PLogP model [71]. A detailed study of these models is carried out in [95], assessing how they apply to MPI collective communications, stating that "performance of collective communications are critical to high performance computing" [95]; as such it is clearly important to be able to analytically understand their performance. The study finds that "all the models can provide useful insights into various aspects of the collective algorithms and their performance" [95], but also finds that simple point-to-point models do not take into account network congestion.

The Hockney model [58] provides that the time taken (t_{comm}) to transfer a message of size m bytes from one process to another across an interconnect is:

$$t_{comm} = \alpha + m\beta \tag{2.1}$$

where α is the latency of the connection, and β the asymptotic time taken to transmit a single byte. This simple model can be extended to further communication operations such as collective communication. However, many current MPI implementations may automatically change the algorithm used to carry out collective communication depending on parameters such as the number of processes communicating, the speed of the network and the size of the message [112, 115]. It may therefore not always be clear which algorithm is being used under the method call for a collective communication within a code, and so which particular model applies best to describe performance.

[5] and [112] both suggest a simple model for computation when considering work involved in communications. If γ is the time taken to compute with one byte, computation time is simply:

$$t_{comp} = \gamma\, m \tag{2.2}$$

2.4.1 Modelling MPI Routines

When carrying out point-to-point communication, the codes examined in this thesis use either `MPI_Sendrecv` (which is a simple combination of `MPI_Send` and `MPI_Recv` within the same message call), or a combination of non-blocking communication (using `MPI_Isend`, `MPI_Recv` and `MPI_Wait` or `MPI_Send`, `MPI_Irecv` and `MPI_Wait`). These communications may be modelled effectively given the model in equation 2.1 [112], although as already mentioned, this does not take into account network congestion when multiple processes are communicating simultaneously.

There are a number of collective communications used by the codes in this thesis. Although not all are described here, the majority of the runtime attributable to collective communications in both codes is due to one of the following operations.

Firstly, some use is made of `MPI_Bcast`. This routine broadcasts data from one process to all others (a 'one to many' operation), and many algorithms exist for carrying out this operation. The simplest linear algorithm for a broadcast over p processors may be modelled as:

$$t_{bcast} = (p-1) \times (\alpha + m\beta) \tag{2.3}$$

A perhaps more widely used algorithm is the binomial tree algorithm, which can be modelled as:

$$t_{bcast} = (\log\ p) \times (\alpha + m\beta) \tag{2.4}$$

Most of the collective communication in the applications used in this thesis is done through calls to `MPI_Allreduce`, a 'many to many' or 'all to all' operation. This routine essentially collects and sums values of data from all processes, ending with all processes containing the final sum of the data. This may be modelled for small messages simply as a reduce operation followed by a broadcast, with an approximate time of:

$$t_{allreduce} = 2(\log\ p) \times (2\alpha + 2m\beta + \gamma\ m) \tag{2.5}$$

Finally, use is also made of `MPI_Barrier`. Conceptually this may be described as a collective communication such as a broadcast with a message size of 0 bytes. It can therefore be modelled linearly as:

$$t_{barrier} = (p-1) \times\ \alpha \tag{2.6}$$

Or, using a binomial tree algorithm, as:

$$t_{barrier} = (\log\ p) \times\ \alpha \tag{2.7}$$

2.4.2 Modelling Hybrid Code

Some attempts are now being made to model the performance of the hybrid model, such as that in [1] which discusses the creation of a general performance model for hybrid codes considering the differences between computation and communication for pure MPI and hybrid OpenMP and MPI codes. This model approaches the performance differences from the position of attempting to quantify the negative effects of the hybrid model, and partly functions by capturing parameters from running code in order to quantify numerical parameters of the model. One of the main aims of the model is to understand the 'best' number of MPI processes and OpenMP threads to run in order to get best performance from a particular hybrid code. As such, it does

not attempt to describe overall runtime, rather looking at parameters such as efficiency as a function of the number of MPI processes and OpenMP threads.

2.5 Molecular Dynamics

> "The ability to follow the motion of many particles in their own and applied fields has long been crucial to both theoretical and experimental studies" [59].

Molecular Dynamics (MD) applications cover the class of application following the simulation of multiple particles, the forces acting between and upon them, and the resultant effect of those forces upon individual particles and the system as a whole. "MD simulations enable the study of complex, dynamic processes that occur in biological systems" [4]. Such simulations involve numerical approximations of physical systems: "At the application level the science has to be captured in mathematical models, which in turn are expressed algorithmically" [42]. These algorithmic expressions generally follow a similar pattern within parallel Molecular Dynamics applications:

1. A system of particles is initialised.
2. This system (or domain) is divided between separate processes.
3. Then, at each timestep:
   (a) Loop over the local particles in a processor.
   (b) Calculate and sum the forces acting upon each particle (a sum of forces due to chemical bonds and interactions with other particles as well as any external fields) [4].
   (c) Calculate the motion of each particle (using Newton's equations of motion - usually calculated with a Velocity Verlet algorithm).
   (d) Update the position of each particle within the space of the simulation.

(e) Perform update communication with neighbours about boundary/halo data and migrate particles to other processes if they have moved far enough in the space to have changed sub-domains.

(f) Perform global communication to update system properties such as the energies of the system.

4. Finally, at the end of the simulation calculate system properties and statistics.

This structure is typical of an MD application: 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.

Many parallel MD applications exist, and [4] contains a performance analysis of some of these applications on multi-core clusters, finding some key conclusions relevant to this work: “On stand-alone dual and quad-core systems, the applications showed sensitivity to the implementation and usage of the MPI communication library” [4]. Also, “On the application front, particularly at large scale, we anticipate that alternate algorithms, programming models and implementation will be investigated to reduce the load balancing problems” [4].

Despite being very different in terms of functionality, the MD applications in this thesis are constructed in a similar way, both following the general structure given above.

Summary

This chapter has introduced the background topics relevant to the work presented in the thesis. It has introduced the ideas behind parallel, shared memory and message passing programming, and the two main methods of implementing these styles, MPI and OpenMP. It discussed combining the two styles to create a hybrid application and examined previous work done on the hybrid model, finding that a consensus has not been reached on the performance of the hybrid message passing + shared memory model and that results vary depending on the applications, algorithms, implementation and hardware used. It has introduced ideas surrounding performance modelling of parallel code, in particular modelling the runtime of communication phases of code. Finally it has introduced the class of applications studied in this thesis, Molecular Dynamics codes, presenting the typical structure of such codes. The following chapter will introduce the hardware used for performance testing, and the testing methodology used.

Chapter 3

Performance Testing

Overview

This chapter describes the hardware used for carrying out the performance testing of the applications in the thesis and the testing methodology used for the performance analysis. It describes the hardware and software of each of the multi-core clusters used in the performance analysis. It also presents the results of some basic performance tests looking at the performance of both MPI and OpenMP on one of the clusters. The chapter then discusses the methodology for testing and collecting results, before describing the methods in which combinations of MPI processes and OpenMP threads were used to test the hybrid model for comparison with the performance of the pure MPI code.

3.1 Hardware

Several different clusters have been used for performance testing of the applications examined in this thesis. *Merlin* is a production cluster in use at the Advanced Research Computing at Cardiff (ARCCA)[1] facility at Cardiff University. *CSEEM64T* is a benchmarking system in use at Daresbury Laboratory. *Stella* is a benchmarking cluster in use at Intel. All three are clusters of multi-core nodes with differing individual characteristics.

3.1.1 Merlin

Merlin is the main HPC cluster at the Advanced Research Computing facility at Cardiff University (ARCCA). 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 and each node has one Infinband link. The cluster also has a Gigabit Ethernet network available, however this is not a dedicated communication network and is also used for node and job management. Results from this network are therefore not a reliable measure of performance and have not been considered in this thesis. The compute nodes of the cluster 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.

3.1.2 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

[1] http://arcca.cf.ac.uk

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.

3.1.3 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.

3.1.4 Hardware Performance

In order to understand the performance of the Molecular Dynamics applications (Chapters 4 and 5) as they relate to the hardware performance, it is necessary to examine the MPI and OpenMP performance. This section uses two benchmarking tools to examine the performance of the Merlin cluster, in order to provide a basic understanding of the cluster performance, allowing conclusions about application performance to be drawn.

MPI performance

Profiling of the Infiniband interconnect on Merlin using the Intel MPI Benchmark suite, running one MPI process per core on a range of different node counts reveals the relationship between the number of processes used, the message size and the achievable bandwidth on this cluster, as shown in Figures 3.1 and 3.2. Figure 3.1 shows the bandwidth available when using the `MPI_Sendrecv` routine to transmit a message of a given size between a number of processes. The processes are arranged in a chain, with each process sending to the process to it's left and

receiving from the process to the right. This represents a similar scenario to the point-to-point communication in the first MD code, which also uses the `MPI_Sendrecv` routine for much of its point-to-point communication. Figure 3.2 shows a similar chain of processes sending data to the left and receiving from the right, this time using separate calls to `MPI_Send`, `MPI_IRecv` and `MPI_Wait`, as in the DL_Poly code. It is clear that in both cases the bandwidth achieved varies as the size of message and number of processes changes. The bandwidth rises as the message size rises to around 64k, then begins to fall as the message size increases above this level. In both cases a lower number of processes delivers a higher available bandwidth. These performance figures indicate that the communication network on the cluster suffers from increased congestion and lower performance as the number of processes increases. They also highlight that there is an optimum message size (around 64k) to be used to get the best performance from the interconnect, and that sending messages that are smaller or larger can have a significant negative effect on performance. It is also worth noticing that a higher bandwidth is achievable using the `MPI_Sendrecv` routine than when using separate calls to `MPI_Send`, `MPI_IRecv` and `MPI_Wait`. An expected outcome of this would be better communication performance in a code which primarily uses the `MPI_Sendrecv` routine than in a code that uses the latter routines.

The overheads associated with message passing communication are observable by examining the time taken to transmit a message of size 0 bytes, as in Figure 3.3. Here, the time taken to transmit the message is measured in microseconds, illustrating that the latency and message exchange overheads are very small on this cluster, using this interconnect. However, it is also clear that as the number of processes increases the overhead associated with messages increases. As the number of processes used rises from 8 to 512 the message overhead approximately doubles. Although the overheads are measured in microseconds they can still have an effect on code performance when carrying out multiple message exchanges per iteration, over many iterations. Just one collective message per iteration would lead to over a second of overhead when running on 8 cores for 500 iterations; this would increase to two and a half seconds when running on 512 cores. As a proportion of the total runtime, this could be a far more significant

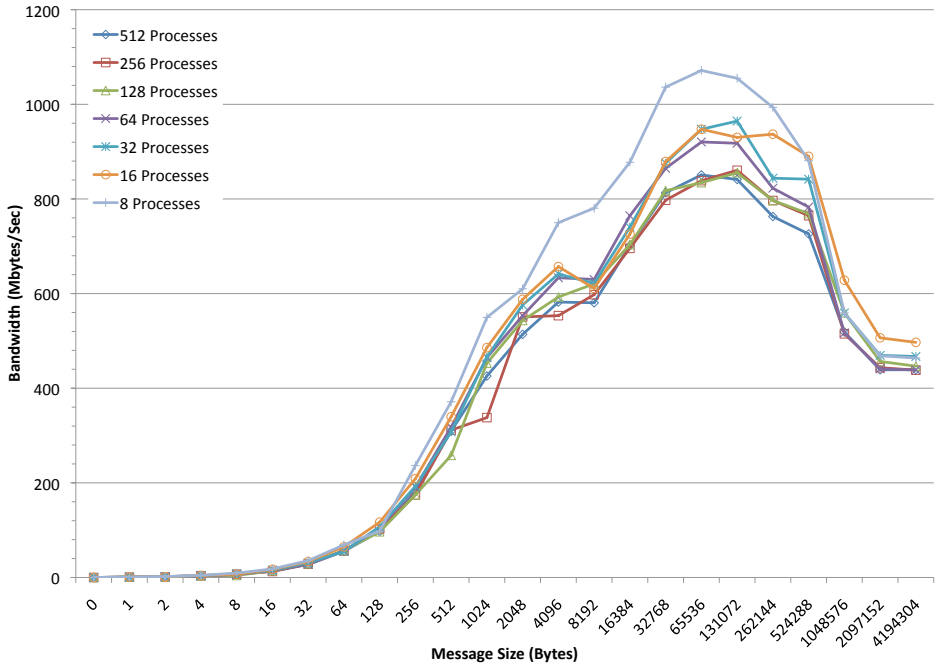


Figure 3.1: Merlin - achievable bandwidth using MPI_Sendrecv on a chain of 8 - 512 processes, each process sending data to the neighbour to the left and receiving data from the neighbour to the right.

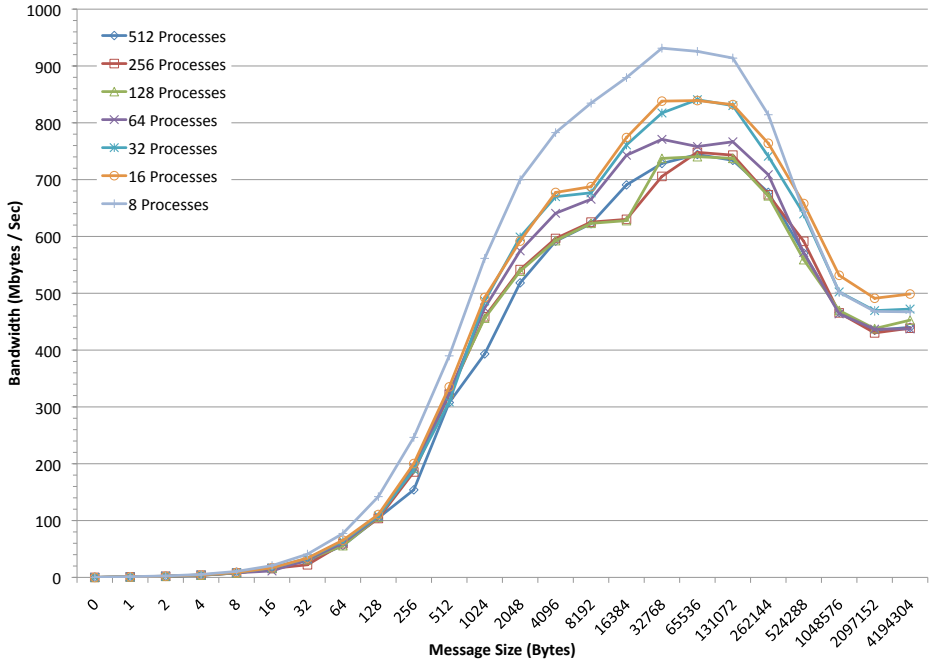


Figure 3.2: Merlin - achievable bandwidth on a chain of 8-512 Processes, each process using MPI_Send, MPI_IRecv and MPI_Wait to send data to its neighbour to the left and receive data from its neighbour to the right.

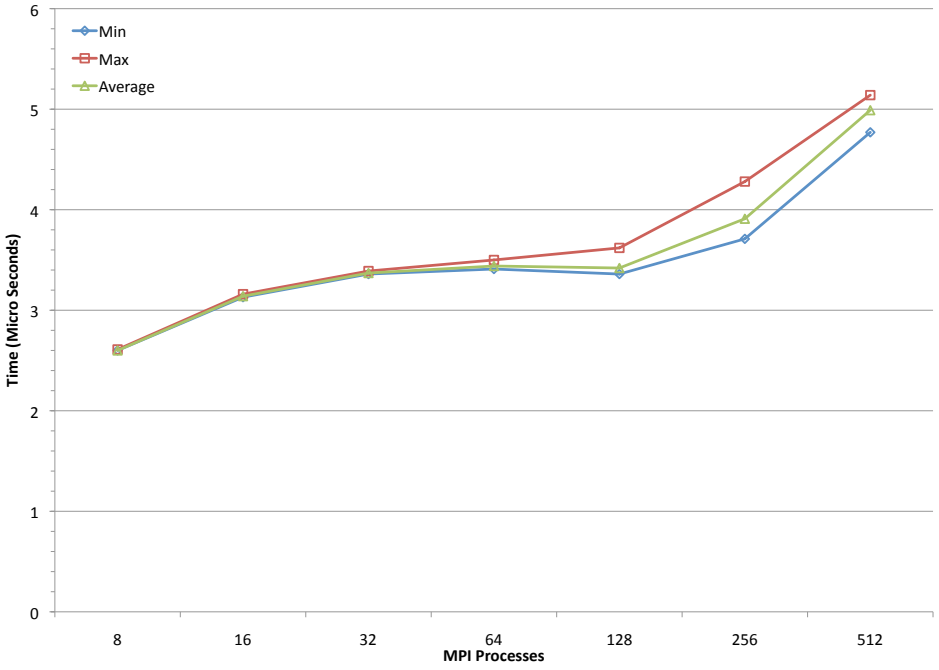


Figure 3.3: Merlin - message overhead when sending a message of 0 bytes size on 8-512 Processes. As the number of processes increases the overheads increase.

overhead when running on large numbers of cores than when running on small processor counts.

OpenMP Performance

OpenMP has certain overheads associated with its use within application codes. Profiling of the OpenMP performance on Merlin with the EPCC Micro-benchmarks [22] allows these overheads to be examined. Table 3.1 shows the average time and overhead associated with the `OMP PARALLEL FOR`, `OMP BARRIER` and `OMP REDUCTION` clauses, while the plot in Figure 3.4 shows how the time and overhead associated with the `PRIVATE` clause changes as the data size associated with the clause changes. Again, although the figures presented in both Table 3.1 and Figure 3.4 are presented in microseconds, when many OpenMP operations are used per iteration they can build up to have a noticeable effect on performance over many iterations. Table 3.1 shows that the `OMP PARALLEL FOR` and `OMP REDUCTION` clauses are almost twice as expensive in terms of runtime than the `OMP BARRIER` clause. Figure 3.4

Clause	Time (μs)	+/-	Overhead (μs)	+/-
Parallel For	2.0122932	0.0133486	1.9212088	0.0144792
Barrier	1.1184576	0.0153798	1.0273728	0.0165108
Reduction	2.141216	0.1811344	2.0468374	0.1818834

Table 3.1: OpenMP Clause Timing & Overheads, with errors (+/-) as provided by benchmark software.

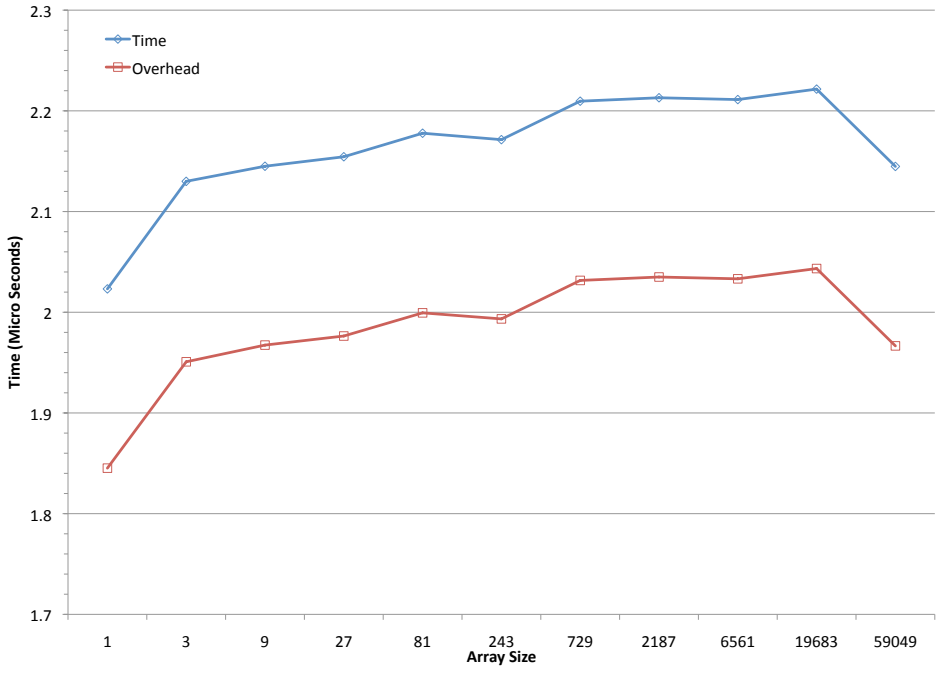


Figure 3.4: OpenMP PRIVATE clause timing & overhead as array size changes. Both overhead and total time increase as the array size increases, until the largest size tested, where both times decrease.

illustrates that in general as the size of an array increases it takes more time to use that array in an `OMP PRIVATE` clause.

3.2 Testing Methodology

Details of the specific testing methodology used for performance analysis of both applications is presented in Chapters 4 and 5. Some testing details are generic to both performance tests however and so are presented here.

When compiling code on each system, the same compiler was used to compile both pure

MPI and hybrid MPI + OpenMP codes, with identical compiler flags used, with the obvious exception that OpenMP compilation was turned on for compiling the hybrid code. All other compiler options remained the same. Similarly, all performance tests using the same hardware and interconnect were done using the same MPI library for both pure MPI and hybrid MPI + OpenMP codes.

3.2.1 Performance Timing

[48] discusses the problems involved in obtaining reproducible measurements of parallel code: "The Unix operating system, together with network hardware and software ... introduces sporadic intrusions into the test environment that must be taken into account" [48]. In line with this observation, when testing each class of code for each application a number of runs were performed, typically 3. Results are reported as an average of all 3 runs, along with the minimum observed time, as measuring time should be done by taking the "minimum time of a number of tests" [48]. For each combination of MPI processes and OpenMP threads three runs of code were performed, with each class of code being tested as a separate job in the job scheduling system.

Errors

Errors are presented with all results throughout the thesis. Unless otherwise specified these errors are given as one standard error to the mean (1 SEM). This value is used rather than simply using the standard deviation as it also takes into account the sample size. Results are reported graphically as in Figure 3.5, where a line plot presents the average observed timing with error bars and a separate marker presents the minimum observed timing. For some plots (particularly for overall timing plots) the errors observed are particularly small, such that the error bars on the plot are too small to be visible.

Where results report some combination of results (for instance when subtracting one time from

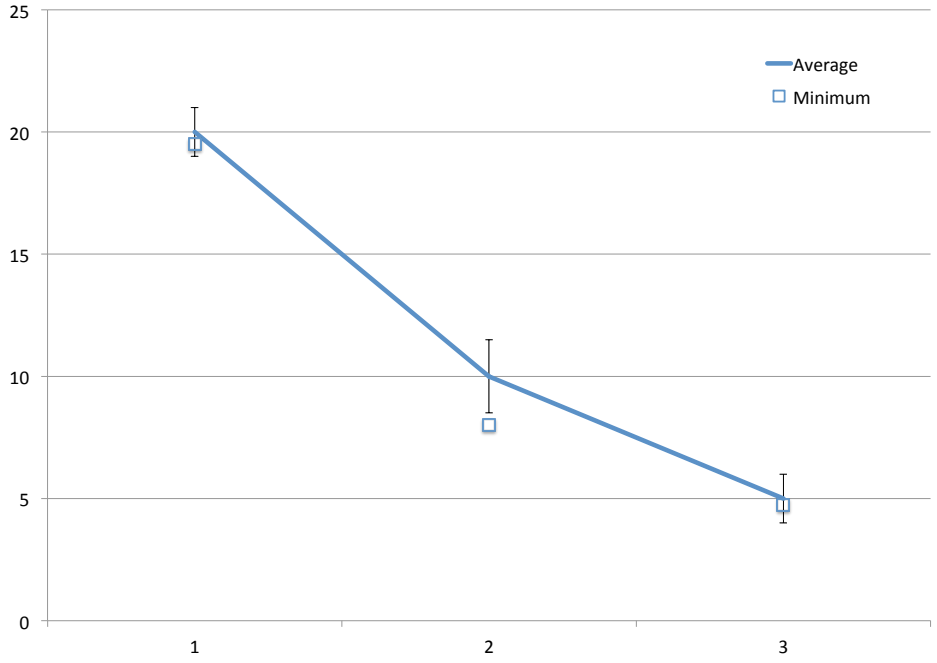


Figure 3.5: Example graphical results presentation. Average recorded times shown as a line plotted with error bars of 1 standard error to the mean (1 SEM). The minimum recorded time is also presented as a separate marker.

another to examine the difference between the two), errors are calculated by combining the error estimates for both measurements.

For results where only one measurement was taken (for example when measuring memory bandwidth in Section 4.3.1) 5% error bars are given.

As runs of application code in the thesis are only repeated a small number of times, some caution must be taken regarding the reported errors. Such errors, calculated from a small sample size may not be statistically valid. However, they do add an extra level of understanding to the performance of the code. In particular, while relatively small error bars do not necessarily indicate that errors are actually very small, large errors might indicate some variability in the code performance requiring further investigation.

Timing methodology

Timing was carried out by instrumenting sections of the code using the `MPI_Wtime` function from the MPI library. Times were collected from the master MPI process and represent elapsed wall time for that section of the application code. For the MD code presented in Chapter 4 the calls to the timing function were inserted directly into the code, while in the DL_Poly code in Chapter 5 the calls were encapsulated within a library function that created, started and stopped timers. This extra encapsulation is not expected to have had any significant effect on the timing results.

Where overall time is reported, it is the total elapsed wall time reported by the master MPI process (the process with rank 0) from when the MD calculations are begun to when they complete, so excluding any initial input or final output operations. Timers for measuring overall timing are started before the main iteration loop within the code and stopped once the loop is complete. The times for individual function calls or classes of communication (point-to-point, collective or synchronisation) represent the cumulative elapsed wall time spent in those functions over all iterations, again within the master process (rank 0). For reporting of function calls and communication timing (calls to the MPI communication library), the function call within the code is 'wrapped' with calls to `MPI_Wtime` to start and end a timer specific to that function. Reported times are therefore the total time spent within that function over all iterations of the simulation.

On all systems used, (64-bit Linux systems) the resolution of the `MPI_Wtime` function was 10^{-6} seconds.

3.2.2 Job Schedulers

On both CSEEM64T and Merlin, access to the compute nodes is controlled through the use of a job scheduling system. Jobs are submitted as script files which specify which resources they require and the job scheduling system then matches the jobs to available nodes. This guarantees

application code exclusive access to the backend compute nodes of the cluster and ensures no runtime disruption due to other users codes.

When running performance tests on fewer cores than are available in one node (when carrying out tests on memory bandwidth or cache access for instance), the whole node was reserved using the job scheduler to ensure exclusive access.

On Stella no job scheduler was available to guarantee exclusive access, however the cluster was accessed by only a single user at a time, so exclusive access to cluster resources was available by default.

An example job script for the running of code is given in Listing 3.1. In this job DL_Poly is being tested on 8 nodes of Merlin, using the hybrid code, running with 1 MPI process and 8 OpenMP threads per node. Lines 1-5 contain commands for the PBS job scheduler. Line 2 specifies a job running on 8 nodes, using 8 processor cores per node, with 1 MPI process per node. Line 3 controls the placement of MPI processes, line 4 specifies how much wall time the job needs to run, and line 5 gives the job a name. Lines 7-21 set up the environment variables needed by the code and the job script, then lines 23-44 actually run the tests and collect the output.

Each class of code was submitted as a separate job to the job scheduler. This means that each combination of MPI processes and OpenMP threads was run as a distinct job, and will therefore have been run at different times and potentially on different nodes to another combination using the same number of processing cores. All systems used in this thesis are homogenous dedicated HPC clusters so performance should be reproducible across all nodes of a cluster at different times with minimal variance.

```
#!/bin/bash
#PBS -l select=8:ncpus=8:mpiprocs=1
#PBS -l place=scatter
#PBS -l walltime=00:60:00
#PBS -N dlpoly(hybrid)

cd $HOME/dl_poly/hybrid/execute
#openmp threads per process
export OMP_NUM_THREADS=8

#number of nodes and mpi procs per node
NNODES=`cat ${PBS_NODEFILE} | sort -u | wc -l`
NMPI=`uniq -c ${PBS_NODEFILE} | awk '{print $1}' | uniq`

#location of dl_poly executable to use
CODE=$HOME/dl_poly/hybrid/execute/DLPOLY.Y
#working directory (on lustre filesystem)
WDPATH=/scratch/$USER/${NNODES}.${NMPI}.${OMP_NUM_THREADS}

#output directory
OUTDIR=$HOME/dl_poly/hybrid/execute/output

for TESTCASE in 10 20 30
do
  for RUN in 1 2 3
  do
        #make working directory valid
        mkdir ${WDPATH}
        cd ${WDPATH}
        #copy executable into working directory
        cp -f ${CODE} .

        #set path to retrieve datafiles from
        DATAPATH=$HOME/dl_poly/testdata/normal/TEST${TESTCASE}
        cp -f ${DATAPATH}/CONFIG .
        cp -f ${DATAPATH}/CONTROL .
        cp -f ${DATAPATH}/FIELD .

        mpirun -np 8 ${WDPATH}/DLPOLY.Y

        mv OUTPUT ${OUTDIR}/${TESTCASE}.dlpoly.${NNODES}.${NMPI
           }.${OMP_NUM_THREADS}.${RUN}.OUTPUT
        rm -rf ${WDPATH}
  done
done
```

Listing 3.1: Hybrid Job Script, DL_Poly, Merlin

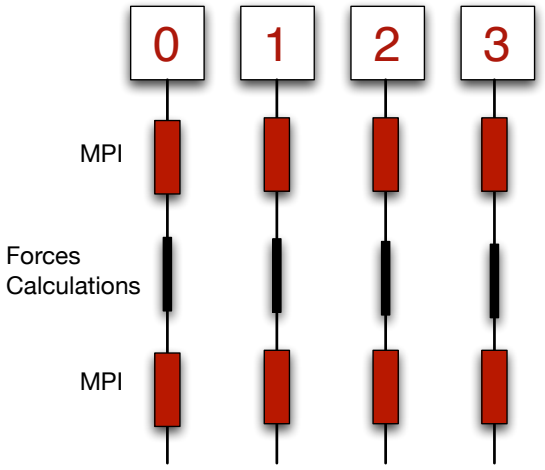
3.2.3 MPI Processes and OpenMP Threads

When running the performance tests of the applications 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)×(OpenMP threads) cores used per node. Each set of test and processor core counts was tested with three combinations of MPI processes and OpenMP threads, as illustrated in Figure 3.6 and described below:

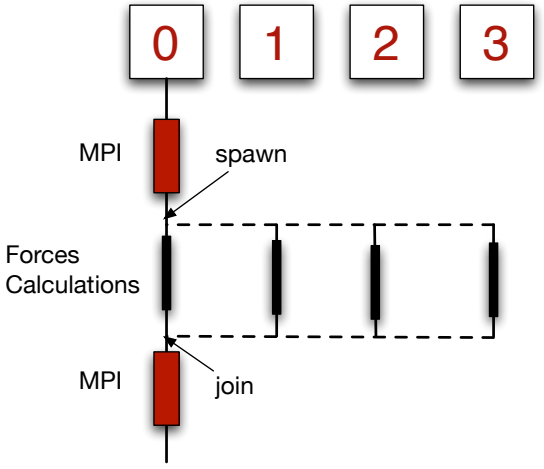
1. Pure MPI - One MPI process is started for each core in a node, no OpenMP threads spawned: Figure 3.6a.
2. Hybrid (1 MPI) - One MPI process started on each node, all other cores filled with OpenMP threads: Figure 3.6b.
3. Hybrid (2 MPI) - Two MPI processes started on each node (one on each processor in the case of dual-processor systems), all other cores filled with OpenMP threads: Figure 3.6c.

3.3 Nodes/Cores/Processes/Threads

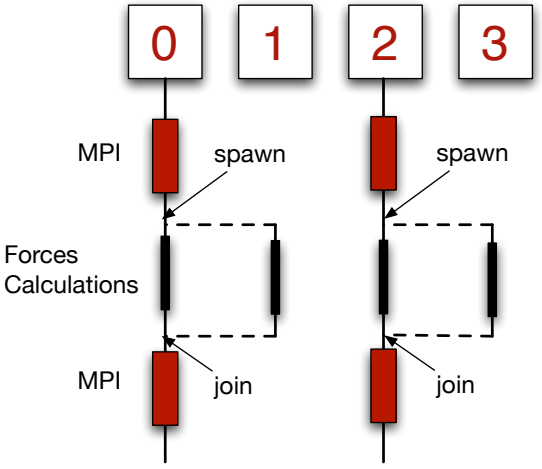
The tables in this section illustrate the different numbers of nodes, processor cores, MPI processes and OpenMP threads that were used for testing on each cluster. The combinations of nodes, number of cores, total MPI processes and OpenMP threads per MPI process (TPP) for each core count for Merlin is shown in Table 3.2, for CSEEM64T in Table 3.3, and for Stella in Table 3.4.



(a) Pure MPI - 1 MPI process per core, no OpenMP threads.



(b) Hybrid (1 MPI) - 1 MPI process per node, all other cores filled with OpenMP threads.



(c) Hybrid (2 MPI) - 2 MPI processes per node, all other cores filled with OpenMP threads.

Figure 3.6: MPI Processes and Shared Memory Thread combinations for Pure MPI, Hybrid (1 MPI) case and Hybrid (2 MPI) case

		Pure MPI	Hybrid (1 MPI) 8 TPP	Hybrid (2 MPI) 4 TPP
Nodes	Cores	MPI	MPI	MPI
8	64	64	8	16
16	128	128	16	32
24	192	192	24	48
32	256	256	32	64
40	320	320	40	80
48	384	384	48	96
56	448	448	56	112
64	512	512	64	128

Table 3.2: Nodes, total number of cores, total MPI processes and OpenMP threads per process (TPP), Merlin

		Pure MPI	Hybrid (1 MPI) 4 TPP	Hybrid (2 MPI) 2 TPP
Nodes	Cores	MPI	MPI	MPI
4	16	16	4	8
8	32	32	8	16
12	48	48	12	24
16	64	64	16	32
20	80	80	20	40
24	96	96	28	56

Table 3.3: Nodes, total number of cores, total MPI processes and OpenMP threads per process (TPP), CSEEM64T

		Pure MPI	Hybrid (1 MPI) 8 TPP	Hybrid (2 MPI) 4 TPP
Nodes	Cores	MPI	MPI	MPI
1	8	8	1	2
2	16	16	2	4
4	32	32	4	8
8	64	64	8	16
16	128	128	16	32

Table 3.4: Nodes, total number of cores, total MPI processes and OpenMP threads per process (TPP), Stella

Summary

This chapter has presented details of the three HPC clusters used for performance analysis, and the methodology used when testing both hybrid message passing + shared memory (MPI + OpenMP) and pure message passing (MPI) code. Performance results were shown describing the general performance of MPI as it relates to the number of processors used and the amount of data being communicated, and describing the overheads of OpenMP as they relate to problem size. The next chapter describes the creation and performance analysis of the first hybrid application studied in this thesis, the small scale MD code.

Chapter 4

Small Scale MD Code

Overview

This chapter describes the creation and testing of a hybrid message passing and shared memory Molecular Dynamics code written in C. It describes the general structure of the code and how the hybridisation was applied, then presents the methodology used for testing. The results of performance testing the code are then shown and discussed. It examines the factors affecting the performance of the hybrid code when compared to the pure MPI code, and justifies the performance differences observed.

4.1 MD Introduction

The first Molecular Dynamics (MD) application used in this thesis is a simple 'example' application. Written in C, it covers a single type of molecular dynamics simulation, simulating 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 comprised of a number of cells. Each cell in the three-dimensional space contains 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 individual particles may migrate from one cell to another as the simulation progresses. No load balancing is carried out during the simulation, but with this code 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 decomposition of the cells over the processors to be used is calculated at runtime, based on input given by the user as to the number of processors in each direction in 3D 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.
- The `sum` routine contains collective communications for summing macroscopic quantities (the virial and the kinetic and potential energies of the system).
- The `hloop` routine checks to see if the code is still equilibrating.

Method / Library Call	Percentage of Total
Application:User_Code	
main.c:main	100.00%
force.c:force	71.73%
cold-start.c:cold_start	8.83%
MPI:MPI_Recv	8.54%
movea.c:movea	7.18%
movout.c:movout	5.39%
moveb.c:moveb	4.16%
hloop.c:hloop	2.16%
scalet.c:scalet	2.16%
edgbuf.c:edgbuf	1.09%
MPI:MPI_Sendrecv	1.02%
sum-energies.c:sum_energies	0.52%
MPI:MPI_Allreduce	0.52%
edgadd.c:edgadd	0.44%
initialise-particles.c:initialise_particles	0.28%
pseudorand.c:pseudorand	0.09%
MPI:MPI_Barrier	0.09%
pack-buffer.c:pack_buffer	0.06%

Table 4.1: Profile of MD application, 4 Nodes, Merlin.

- The `movea` and `moveb` routines which each perform part of the Velocity Verlet algorithm.

4.1.1 Hybrid Version

The hybrid version of the MD application code was created by adding OpenMP parallelisation on top of the original MPI parallel code. Table 4.1 shows a flat profile (collected using Intel Trace Analyzer and Collector) of the percentage of the total runtime spent in each method of the pure MPI code running on 4 nodes of Merlin, with 8 MPI processes per node and using the Infiniband connection. Methods or calls to the external MPI library taking up less than 0.05% of the total time are not shown.

From the table it is clear to see that the force method makes up the majority of the runtime, accounting for almost 72% of the total runtime when running on 4 nodes. It is this method that

```
1   #pragma omp parallel default(none) shared(v,p,fx,fy,fz,rx,ry
       ,rz,head,list,map,natm,sigsq,rcutsq,rcut,vrcut,dvrc12,
       dvrcut,mx,my,mz,cellix,celliy,celliz,sfx,sfy,sfz)
2   {
3
4      #pragma omp for private(i)
5      for(i=0;i<natm;++i)
6      {
7         // initialise forces array
8      }
9
10     #pragma omp for private(xc,yc,zc,rxi,ryi,rzi,fxi,fyi,fzi,
          rxij,ryij,rzij,rijsq,rij,sr2,sr6,vij,wij,fij,fxij,fyij
          ,fzij,i,icell,j,jcell) reduction(+:v,p)
11     for(icell=1; icell <= mx*my*mz; icell++)
12     {
13        // forces calculations
14     }
15  }
```

Listing 4.1: OpenMP additions to forces main work Loop

is the focus of the OpenMP parallelisation.

The force method contains a for loop over all the particles within the subcells belonging to the process. To parallelise this section of code with OpenMP a parallel region is started using a `omp parallel` directive just before the main `forces` loop of the application, with the main particle arrays being shared between threads, along with other global variables (Listing 4.1, line 1). The iterations over the sub-cells in the main loop are divided between the threads using an `omp for` directive, private copies of many variables and arrays are created and a `reduction` clause is used to manage the global sums performed to evaluate the virial and the potential and kinetic energies (line 10). 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 parallelisation has been applied to the loops over all particles in both the `movea` and `moveb` routines, where the velocity verlet calculations are carried out. In all other routines the hybrid code will be running with less overall parallelisation

than the MPI code, so a decrease in performance of the routines without any form of OpenMP parallelisation is seen when compared to the MPI code.

This parallelisation uses a master only style of hybrid implementation[98]; there are no calls to the MPI library contained within any of the parallel regions, so no requirements are placed on the MPI implementation as to the level of threading support required.

4.2 Performance Testing

Performance analysis of both the Hybrid and pure MPI versions of the simple MD application was carried out on the Merlin and CSEEM64T clusters. Both are clusters of multi-core nodes with a high-speed low-latency 'typical' HPC interconnect: an Infiniband interconnect in Merlin and an Infinipath connection in CSEEM64T. CSEEM46T also has a dedicated Gigabit Ethernet network. For details of the hardware, see Sections 3.1.1 and 3.1.2.

4.2.1 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 time steps. A range of core counts was used for testing, from 64 to 512 cores on Merlin, and from 16 to 96 cores on CSEEM64T, as illustrated in the tables in Section 3.3. Each combination of test size and number of cores was run three times. Results are presented as described in Section 3.2, with average times recorded presented with error bars of 1 standard error to the mean and minimum times recorded also reported.

The assignment of processor cores to the three dimensions for the data decomposition was chosen to be as equal as possible, as seen in Table 4.2, where the number of processor cores in each of the X, Y and Z dimensions are given with the number of MPI processes used at each core count.

Number of Cores	64	128	192	256	320	384	448	512
Pure MPI								
MPI Processes	64	128	192	256	320	384	448	512
X	4	8	8	8	8	8	8	8
Y	4	4	6	8	8	8	8	8
Z	4	4	4	4	5	6	7	8
Hybrid (1 MPI)								
MPI Processes	8	16	24	32	40	48	56	64
X	2	4	4	4	5	6	7	4
Y	2	2	3	4	4	4	4	4
Z	2	2	2	2	2	2	2	4
Hybrid (2 MPI)								
MPI Processes	16	32	48	64	80	96	112	128
X	4	4	6	4	5	6	7	8
Y	2	4	4	4	4	4	4	4
Z	2	2	2	4	4	4	4	4

Table 4.2: Distribution of processor cores across dimensions for three-dimensional data decomposition

If using less than the full number of cores on a node, when carrying out tests on memory bandwidth for instance (see Section 4.3.1), the full node was reserved via the job queueing system to ensure exclusive access while the performance testing was carried out.

Sections of code have been instrumented using the default MPI timer function `MPI_Wtime` to calculate runtimes of individual methods within the code. This study is primarily concerned with raw performance, so I/O has been minimised within the code and removed wherever possible. Times reported are elapsed wall time as measured using `MPI_Wtime` in the master MPI process (the MPI process with rank 0).

4.3 Hybrid Code Performance

This section details and discusses the performance results of the hybrid MD code as compared to the pure MPI code on both Merlin and CSEEM64T. General performance issues surrounding

System Processor ID	Physical ID	Core ID
0	0	0
1	1	0
2	0	1
3	1	1
4	0	2
5	1	2
6	0	3
7	1	3

Table 4.3: Example processor ID, physical ID and core ID numbering

multi-core processors are discussed first, followed by the overall timing results of the code. A detailed breakdown of the results is then given, looking at the reasons behind the differences in performance for the hybrid and pure MPI codes.

4.3.1 Memory Bandwidth and Cache Sharing

The use of multicore processors in modern HPC systems has created issues [42] 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). Experiments have been done with the hybrid and pure MPI code to assess the effects of these on the code performance.

Memory bandwidth and cache sharing issues are easily exposed in a code by comparing performance of the application with fully populated and under populated nodes or processors. In order to carry out these tests it is necessary to control which processes are assigned to which cores on which physical processor. A combination of standard linux command line tools allow this to be done. Examining `/proc/cpuinfo` on each node of a system describes the layout of the processing cores of the node in terms of processor ID as recognised by the system, physical ID and core ID. So, for example, a node of Merlin may report core/processor IDs as in Table 4.3. In this example, to ensure four processes are running on the same processor it is necessary to ensure they are assigned to system processors 0, 2, 4 and 6 or 1, 3, 5 and 7. An example avoiding

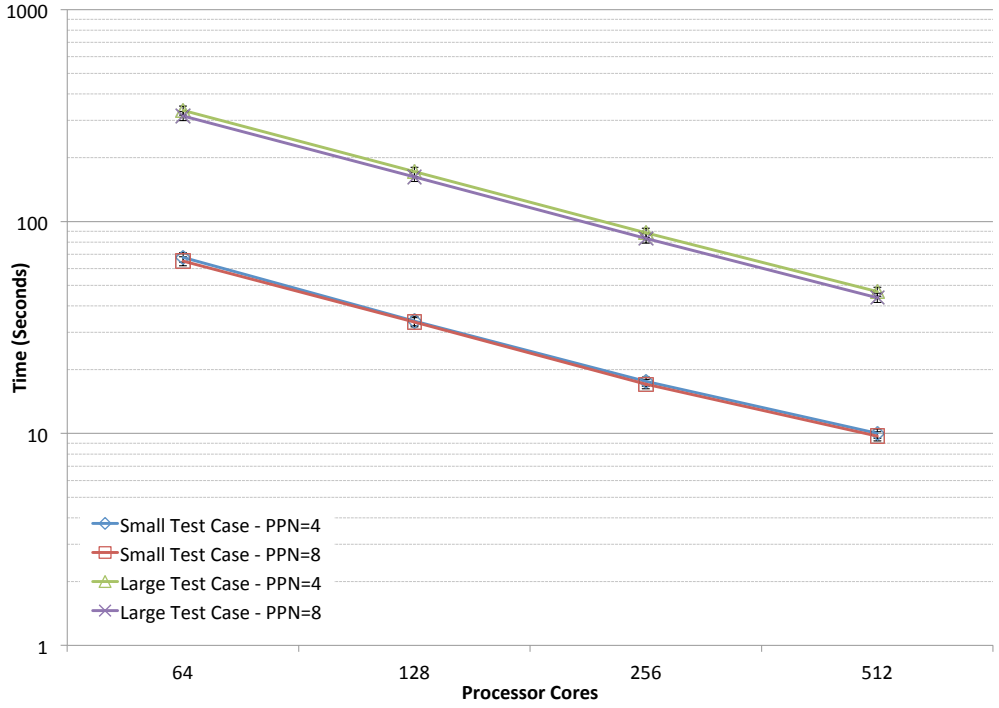


Figure 4.1: Merlin memory bandwidth tests. Error bars of 5% shown. Very little performance difference is observed between the codes running on a fully populated node (ppn=8) and an under populated node (ppn=4). Overlapping error bars indicate no statistical significance in difference between timings on under populated and fully populated nodes.

cache sharing would require processes to be assigned to system processors 0, 1, 4 and 5 (since cores 0 and 1 share L2 cache, as do cores 2 and 3). This process assignment can be done using the command line tool `taskset`, which can be used to set the CPU affinity of a process. A combination of knowledge of how the system views the processors in terms of numbers and use of the `taskset` tool allows fine control of process placement to test for performance issues related to memory bandwidth and cache sharing.

Combined effects of Memory Bandwidth and Cache Sharing

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), with processes placed so that cache is not shared, any performance differences due to either the

Small Simulation						
	Pure MPI			Hybrid (1 MPI)		
	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						
	Pure MPI			Hybrid (1 MPI)		
	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 4.4: Timing effects of cache sharing. Times in seconds. MPI code affected more by cache sharing than hybrid code, but effects on overall performance are small.

reduced memory bandwidth or the cache sharing on a fully populated node can be seen. The results of this test are shown in Figure 4.1, which shows the total elapsed wall time for both the small and large test sizes running on under populated and fully populated nodes. They clearly show that there is little performance difference between a fully populated and under populated node, demonstrating that memory bandwidth and cache sharing is not a large issue with this MD code, so is not a significant factor when examining performance results.

Cache Sharing

The effect on performance due solely to cache sharing can be examined separately by running the code on an underpopulated single node of the system using either one or both processors. Each node in the Merlin cluster contains two quad-core processors and each physical processor has two pairs of cores on a chip sharing an L2 cache. Therefore, 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. As both processors share the connection to main memory, the memory bandwidth contention when running the same number

of threads/processes on a node should be the same whether the processes are concentrated on one processor or spread over both. The timing results for the two codes, and difference between the exclusive and shared cache timings are presented in Table 4.4.

The differences between shared and exclusive cache use are very small, never larger than 4.5%, which occurs in the pure MPI code. Overall the pure 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 in both the small and large test case. This is to be expected; 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 therefore only sharing cache between cores during the execution of the OpenMP parallelised routines, at most other points only one MPI process is running which will have exclusive access to the cache. This effect of the hybrid parallelisation means that it is reasonable to expect that the Hybrid code will be affected less by cache sharing over the entire run of the application. Examining the portion of the total difference that can be attributed to the `forces` routine (where both hybrid and pure MPI codes are running in parallel) 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 a significant difference between either code.

The results of the cache sharing analysis shows that cache sharing has a slight negative impact on the performance of the code, but that the difference is very small between shared and exclusive cache use. The cache sharing is also not a significant cause of the performance difference between the pure MPI and hybrid codes, as both are affected.

4.3.2 Overall Timing

This section presents the results of the overall code timing for the simple MD code on both clusters. All results represent the minimum total elapsed wall time (as recorded using the `MPI_Wtime` function) reported by the master process (the MPI process with rank 0).

Throughout the overall performance results observed for this code two simple patterns can plainly be observed.

The first pattern is that the main work portion of the code (the `forces` routine) is always slower in the hybrid code than the pure MPI code. This is due to 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 energy 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. Additional overheads are caused by the need to create private copies of individual arrays within each OpenMP thread and populate these arrays with the correct data. This is evidenced in Listing 4.1, showing that the hybrid version of the code creates a number of private copies of arrays within the forces loop, and also showing the reduction clause added to synchronise energy values between threads. These additions can be expected to add overheads as seen in the EPCC benchmark results in Section 3.1.4, particularly the results seen in Figure 3.4 which demonstrates that the private clause overheads increase as the size of array to be copied increases and the results in Table 3.1 showing the overheads added by each reduction clause.

The second pattern is that 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 interconnects well. However, when using the lower bandwidth 1 Gigabit Ethernet interconnect and running the code on large numbers of nodes, the opposite is seen. When using large numbers of cores over such an interconnect, the hybrid code performs better than the pure MPI code. A larger number of MPI processes results in more network traffic and congestion when Gigabit Ethernet is used, reducing the communication efficiency, and therefore reducing the communication performance. The hybrid code uses fewer processes, so suffers less from this problem, and thus performs better.

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. This is not seen to be a significant factor in the performance differences in this application on these clusters. The MPI implementations used for performance testing on the two systems include communication devices that allow 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 perform (nearly) as well as the shared memory accesses in the hybrid code. This is a common feature of recent MPI implementations, which results in few differences between shared memory and message passing code when running on one node, removing one of the areas where performance may be improved between hybrid and pure MPI codes.

The two main factors that do have an effect on the timing results are the changes in communication profile between hybrid and pure MPI codes, discussed in Section 4.3.3 and the extra overheads in the hybrid code as a result of the shared memory parallelisation, discussed in Section 4.3.4.

Merlin Overall Timing

On the Merlin cluster, (two quad-core processors per node), it is clear that when using the Infiniband connection the MPI code is consistently faster than either of the hybrid versions. The hybrid (1 MPI) version is slower than the hybrid (2 MPI) approach, while both are slower than the pure MPI code. Figures 4.2 and 4.3 show the timing of the application for both the small and large tests on Merlin using the Infiniband connection.

The performance gaps between the three code variants remains consistent as the number of cores increases, demonstrating that the scalability of the three codes is very similar. The performance of the hybrid (1 MPI) and hybrid (2 MPI) codes is also very similar throughout all core counts. Using the smaller test case the performance difference between the hybrid (1 MPI) and hybrid (2 MPI) codes is smaller than when using the large test case.

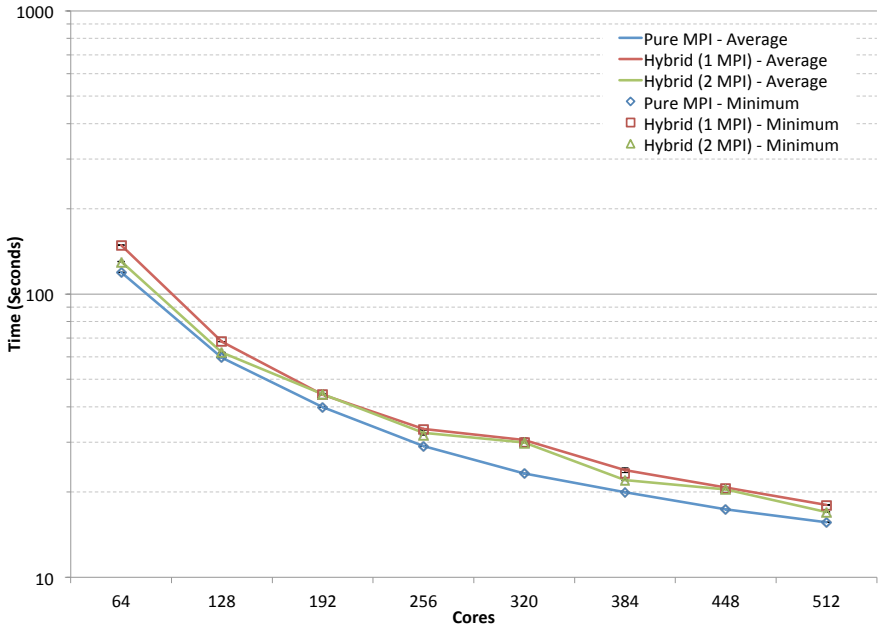


Figure 4.2: Average performance timing for the small test case on Merlin using the Infiniband interconnect, shown with error bars of 1 SEM and minimum observed timing. MPI is consistently faster than both hybrid (1 MPI) and hybrid (2 MPI).

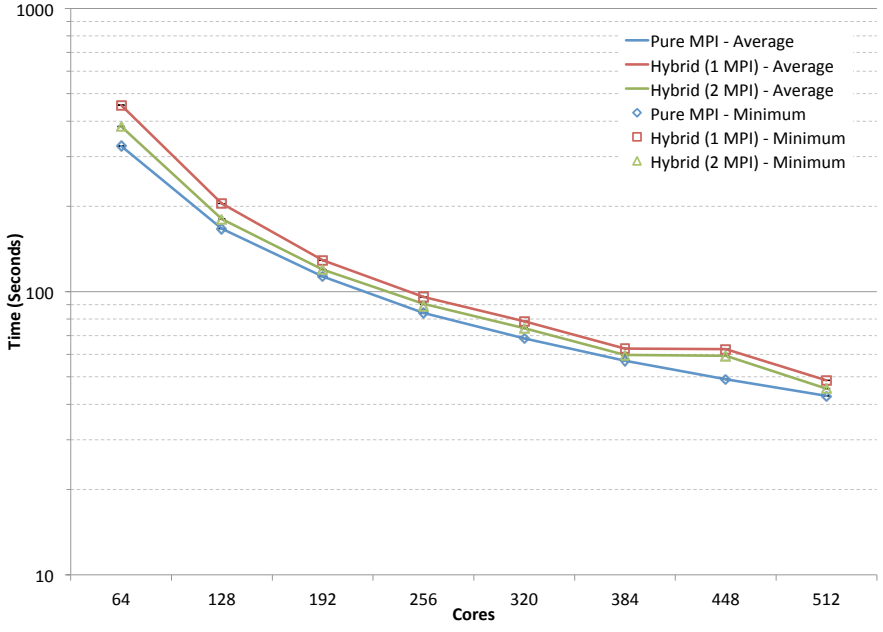


Figure 4.3: Average performance timing for the large test case on Merlin using the Infiniband interconnect, shown with error bars of 1 SEM and minimum observed timing. MPI is consistently faster than both hybrid (1 MPI) and hybrid (2 MPI).

In the small test case both hybrid codes experience performance ‘blips’ at 320 cores, with hybrid (2 MPI) showing another ‘blip’ at 448 cores. In the large test case both codes show a similar performance anomaly at 448 cores. A possible cause for these anomalies may be the processor distribution used to assign processor cores across the three dimensions of the data decomposition. As can be seen in Table 4.2, these performance ‘blips’ occur in situations where one of the dimensions is an odd number, which could lead to some imbalance in communication between MPI processes. However, the pure MPI case also has an odd processor core distribution in one dimension at these core counts and it does not exhibit such performance anomalies. Further experimentation with the breakdown of processor cores across the three dimensions is therefore necessary to firmly conclude that this is the cause.

For all results on Merlin the standard errors to the mean are relatively small for all core counts, being difficult to see behind the plot lines and minimum time markers. Although this does not indicate (due to the small sample size from which errors are calculated) that the actual errors are also this small, it does show that a large amount of variance was not seen between the three runs performed on this system.

Overall there is one clear conclusion from this data - the pure MPI model outperforms the hybrid message passing + shared memory model at all core counts.

CSEEM64T Overall Timing

On the CSEEM64T cluster using the Infinipath connection a similar performance pattern to that seen on the Merlin cluster is observed: the pure MPI code outperforms both the hybrid (1 MPI) and hybrid (2 MPI) approaches at all problem sizes, as seen in Figures 4.4 and 4.5. Unlike on Merlin however, there are large differences between the hybrid (1 MPI) and hybrid (2 MPI) results when using the Infinipath connection. The hybrid (2 MPI) 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 MPI) timings. The performance of the hybrid code is effectively reversed from the situation using the Infiniband connection on Merlin, where the hybrid (1 MPI) code was consistently

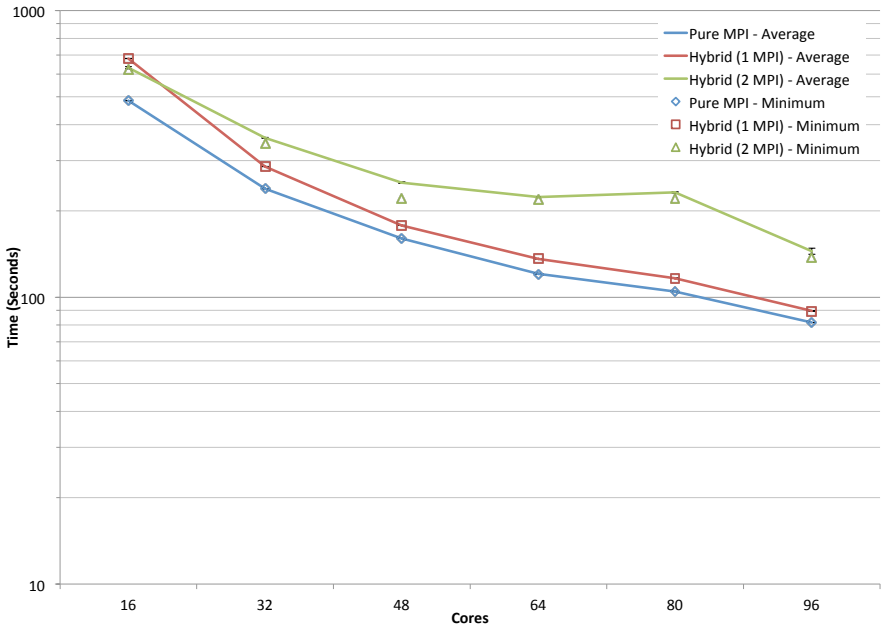


Figure 4.4: Average performance timing for the small test case on CSEEM64T using the Infinipath interconnect, shown with error bars of 1 SEM and minimum observed timing. Hybrid (1 MPI) and hybrid (2 MPI) out performed by pure MPI at all core counts.

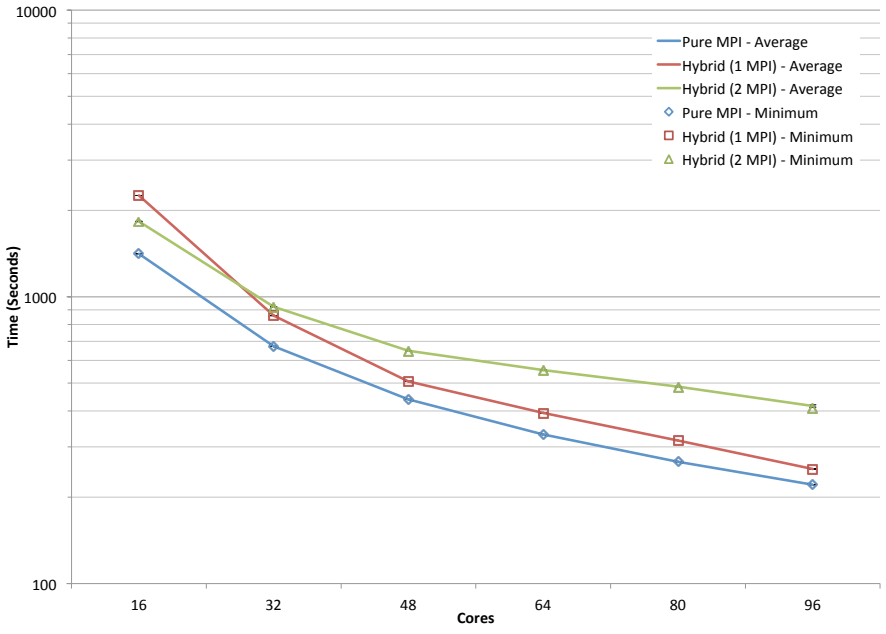


Figure 4.5: Average performance timing for the large test case on CSEEM64T using the Infinipath interconnect, shown with error bars of 1 SEM and minimum observed timing. Hybrid (1 MPI) and hybrid (2 MPI) out performed by pure MPI at all core counts.

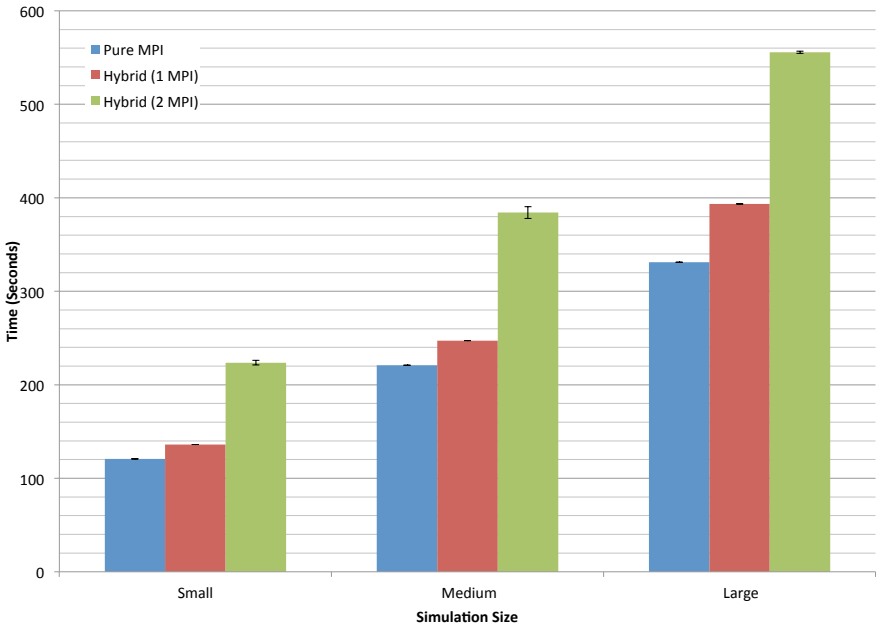


Figure 4.6: Average 64 core timing for all tests on CSEEM64T using the Infinipath connection. Pure MPI provides best performance, while hybrid (1 MPI) performs better than hybrid (2 MPI).

slower than the hybrid (2 MPI) code. Using the Infinipath connection on CSEEM64T, the hybrid (2 MPI) is slower than the hybrid (1 MPI) on all but the lowest core counts. Figure 4.6 shows the code running at all simulation sizes on 64 cores, showing the performance gap between MPI, hybrid (1 MPI) and the hybrid (2 MPI) codes clearly.

Using the Gigabit Ethernet connection the results are very different, with the hybrid codes generally performing better than the pure MPI code at all problem sizes when running above a certain number of cores. Below 48 cores the pure MPI code performs better than both hybrid (1 MPI) and hybrid (2 MPI) codes. However, both hybrid (1 MPI) and hybrid (2 MPI) are faster than pure MPI above 48 cores for the small (Figure 4.7) and large (Figure 4.9) test cases, while hybrid (2 MPI) is faster than pure MPI above 48 cores for the medium test case, and hybrid (1 MPI) is faster than pure MPI above 64 cores (Figure 4.8).

It is worth noticing that the performance gap between pure MPI and hybrid codes when running on large numbers of cores is quite small, even on this slower interconnect, where the hybrid code is expected to have a significant advantage over the pure MPI code due to reduced

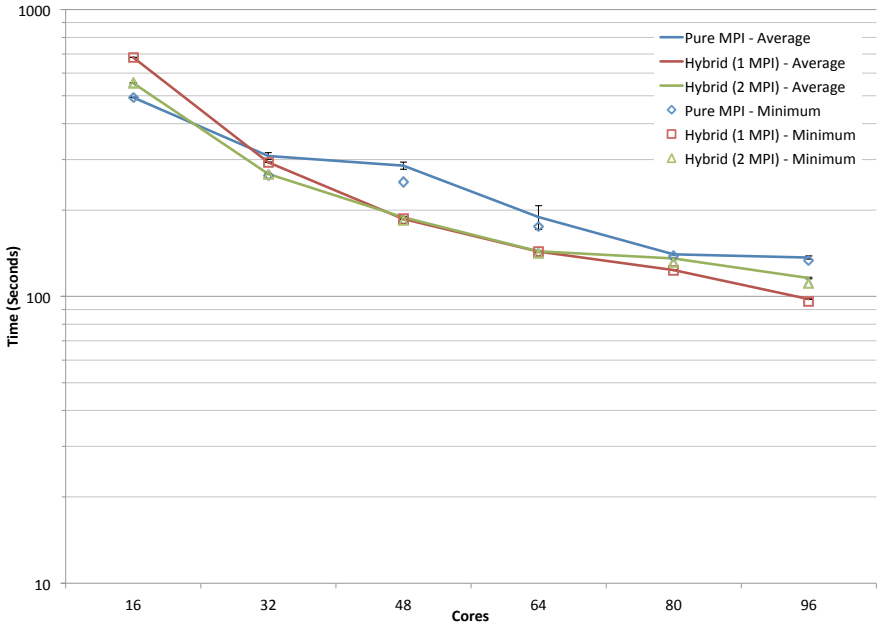


Figure 4.7: Overall average timing, small test on CSEEM64T using GigE connection, presented with error bars of 1 SEM and minimum observed timing. Below 48 cores pure MPI provides best performance, above 48 cores both hybrid (1 MPI) and hybrid (2 MPI) provide better performance than pure MPI.

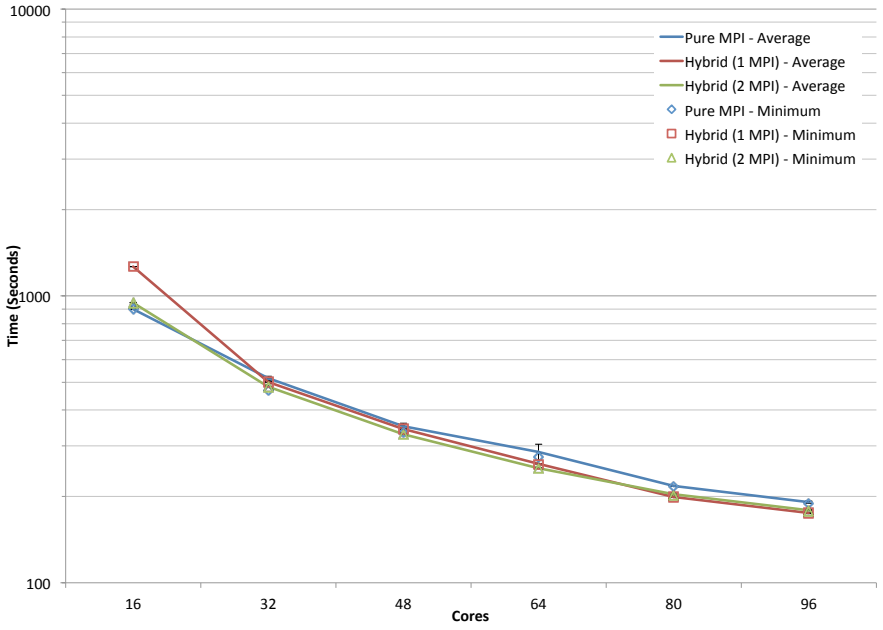


Figure 4.8: Overall average timing, medium test on CSEEM64T using GigE connection, presented with error bars of 1 SEM and minimum observed timing. Below 48 cores pure MPI provides best performance, above 48 cores both hybrid (1 MPI) and hybrid (2 MPI) provide better performance than pure MPI.

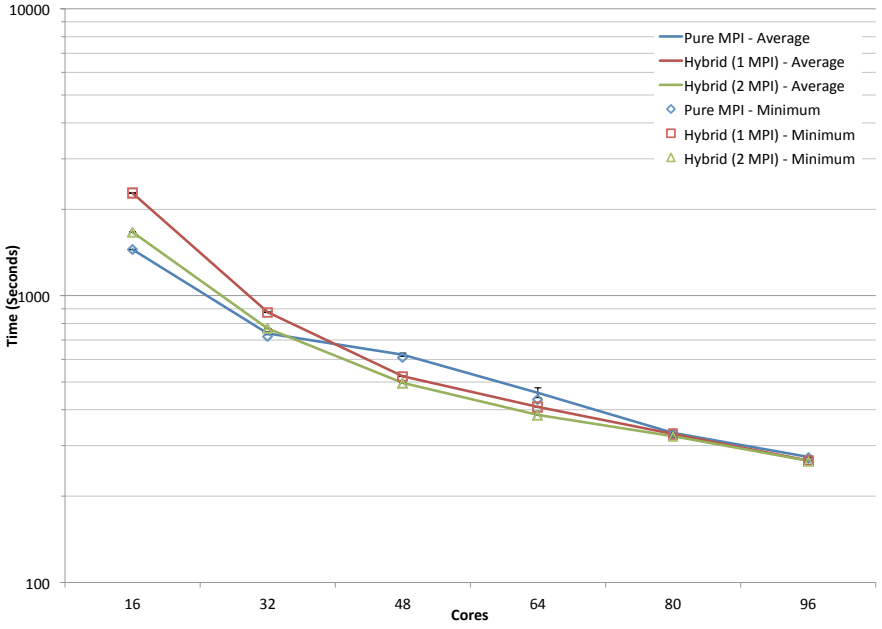


Figure 4.9: Overall average timing, large test on CSEEM64T using GigE connection, presented with error bars of 1 SEM and minimum observed timing. Below 48 cores pure MPI provides best performance, above 48 cores both hybrid (1 MPI) and hybrid (2 MPI) provide better performance than pure MPI.

communication costs. This demonstrates that communication is not a large enough factor in this MD code to cause significant differences between the performance of the two models, explaining why there is no performance benefit from the hybrid model on the faster Infiniband and Infinipath interconnects.

Figure. 4.10 shows the overall timing results at 64 cores for the pure MPI, hybrid (1 MPI) and hybrid (2 MPI) codes for all three simulation sizes on the Gigabit Ethernet connection. Comparing this to Figure 4.6 illustrates the difference the interconnect has on the performance of the codes. The hybrid model is clearly faster than the pure MPI when running over a Gigabit Ethernet connection. Since the only difference between the two sets of performance results is the interconnect used, this change in performance must be down to the communication performance of the code.

For both interconnects at all core sizes the standard errors to the mean are relatively small. Again, this does not prove that actual errors are small, but does show a lack of variance observed

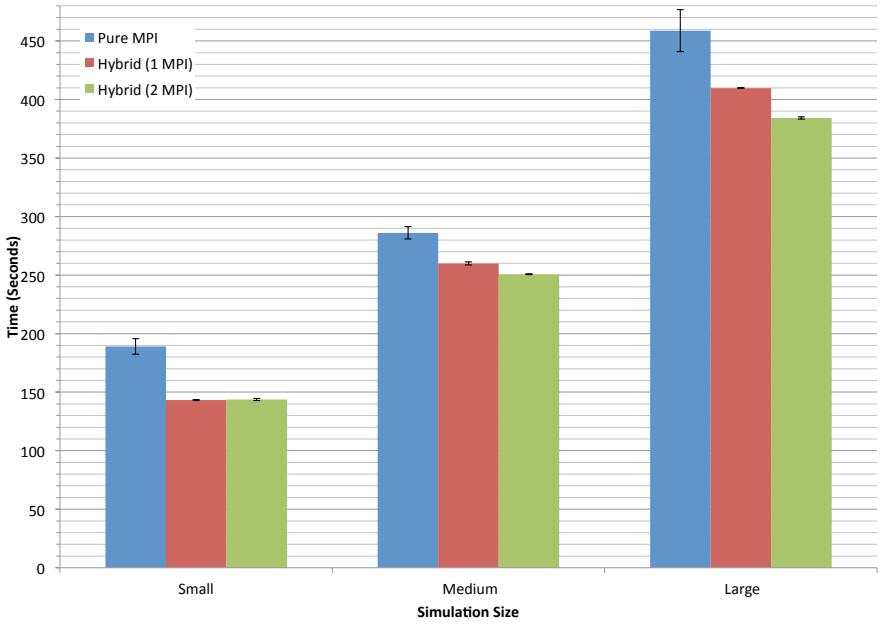


Figure 4.10: Average 64 core timing for all tests on CSEEM64T using the GigE connection, presented with error bars of 1 SEM. Hybrid (2 MPI) performs better than hybrid (1 MPI), both perform better than pure MPI.

in the three runs performed on both interconnects on this system. It is however worth noticing that for the pure MPI code at some core counts the standard errors observed are larger on the Gigabit Ethernet connection than on the Infinipath connection. This is particularly noticeable at 64 cores for all three test cases for instance, and can easily be seen by comparing the standard errors presented in Figure 4.6 with those in Figure 4.10. This indicates a larger variance in the runtimes of the code here which may be caused by the nature of the GigE interconnect, where contention between processes on the interconnect can cause delays in communication. This could affect the runtime enough to cause variance between each run. The fact these larger errors are seen with the pure MPI code and not either of the hybrid codes (which use a much lower number of MPI processes) and on one interconnect but not the other lends weight to this being an issue related to communication performance. Further examination of the communication performance over the interconnect would be necessary to confirm this as the cause of the variance.

Overall Timing Summary

The overall timing results reveal that on a low latency high bandwidth interconnect such as the Infiniband connection on Merlin or the slightly higher latency Infinipath connection on CSEEM64T the hybrid model shows no performance advantage over the pure MPI code. The pure MPI code is consistently faster on a full range of nodes. However, on a slower Gigabit Ethernet connection, the hybrid model outperforms the pure MPI code when running on larger numbers of nodes.

These performance differences can be attributed to two principal differences between the pure MPI and hybrid code. Firstly, the hybrid code has extra overheads caused by the shared memory threading. These overheads may be direct effects of the use of OpenMP, for instance the additional time taken to carry out `omp reduction` clauses, or they may be indirectly caused by the change in parallelisation, as some sections of the hybrid code are less parallelised than in the pure MPI code, as they fall outside of the regions to which OpenMP parallelisation has been applied. These overheads add additional runtime to the hybrid message passing + shared memory codes, making the performance worse. Secondly, the communication profiles of the pure MPI and hybrid codes are very different due to the difference in numbers of MPI processes being used; this can have an effect on the run time of the code. The hybrid message passing + shared memory model uses fewer MPI processes, which on a slower Gigabit Ethernet connection offers a performance improvement over a pure MPI code.

The next sections (Section 4.3.3 and Section 4.3.4) will explore the results to attempt to describe and visualise these effects.

4.3.3 Communication Profile

As the results on CSEEM64T have shown, significant performance differences are found between the pure MPI and hybrid message passing + shared memory models when used over different interconnects. The communication profile of the codes is the cause of this performance

difference, as when inter-node communication is involved, the hybrid code has a very different communication profile from the pure MPI code which results in these different timings for the two codes.

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

Profiling of the code allows the details of the point-to-point communication patterns to be revealed. Intel Trace Analyzer and Collector [65] is able to capture details of all MPI communication carried out within an application while it is running. Separate runs of the application were done to collect these statistics, so as not to contaminate the timing data used in the performance analysis with the effects of extra instrumentation from the ITAC software.

Figure 4.11 shows the average amount of data sent by each process per time step for the small simulation. It is easy to see that there is much more data sent per time step per process in the hybrid code than in the pure MPI code. As each process sends the same number of messages per time step, the messages being sent by the hybrid codes must be much larger in size than the messages in the pure MPI code, in order for the average for the hybrid message passing + shared memory processes to be larger. As the number of cores increases, and so the sub-domains on each process reduce in size, the message sizes in the hybrid code become much smaller, although they are still larger than in the pure MPI code.

Figure 4.12 shows the total cumulative data sent by all processes per time step. Clearly, the pure MPI code sends more total data over all processes per time step than either the hybrid (1 MPI) or hybrid (2 MPI) codes. The individual message sizes of the pure MPI code are smaller per time step (Figure 4.11), so this illustrates that the pure MPI code is sending a much larger number of messages per time step than the hybrid code.

It is worth noticing that at points in Figure 4.12 the total data sent is actually very similar

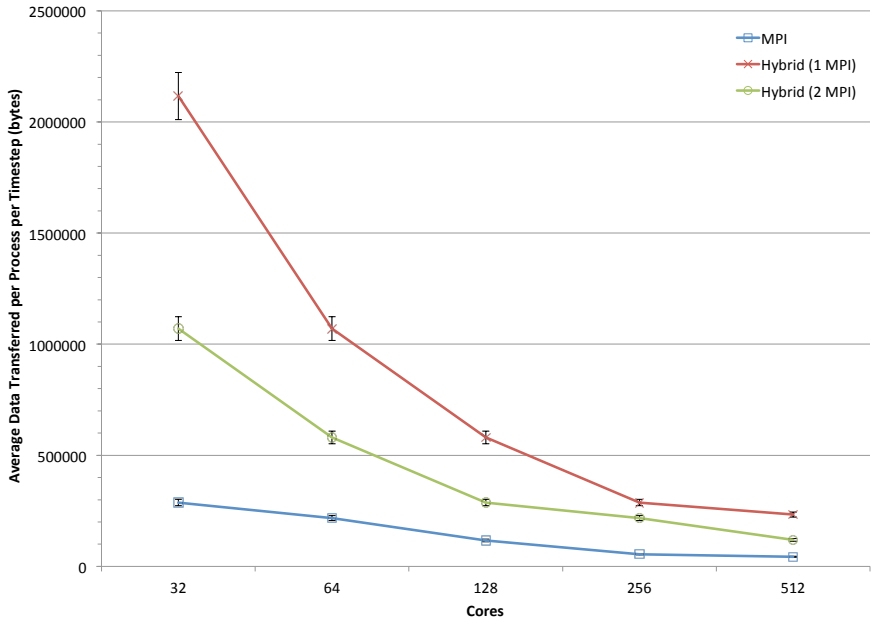


Figure 4.11: Average amount of data sent per process per time step for the small simulation. 5% error bars shown. As the number of cores increases, the amount of data sent per process per time step reduces by a larger amount in the hybrid code than in the pure MPI code.

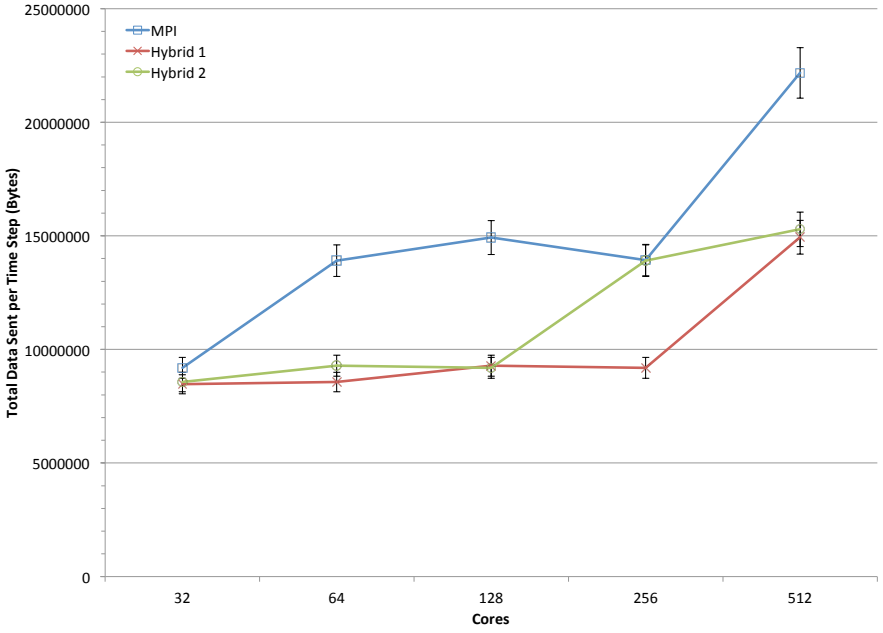


Figure 4.12: Cumulative data sent per time step for the small simulation. 5% error bars shown. The pure MPI code sends more data per time step in total than the hybrid codes.

		Pure MPI : Hybrid 1	Pure MPI : Hybrid 2	Hybrid 1 : Hybrid 2
	MPI Ratio	8:1	4:1	1:2
Cores	32	1:**7.37**	1:**3.73**	**1.98**:1
	64	1:4.93	1:2.67	1.84:1
	128	1:4.98	1:2.46	**2.02**:1
	256	1:5.28	1:**3.99**	1.32:1
	512	1:5.39	1:2.76	**1.95**:1

Table 4.5: Ratios between average amount of data sent per process for all three classes of code. Core counts where ratio is similar to number of MPI processes (so where total amount of data transferred will be similar) are highlighted in bold.

between the three classes of code. For instance, at 32 cores all three classes of code transmit a similar amount of data per time step, at 128 and 512 cores the hybrid (1 MPI) and hybrid (2 MPI) codes transmit a similar amount of data per process, and at 256 cores the pure MPI and hybrid (2 MPI) transmit a similar amount of data. This is due to the relationship between the amount of data sent per process and the number of MPI processes used for each class of code. For instance, at 32 cores, the average amount of data sent per process in the hybrid (1 MPI) code is around 7.37 times that of the pure MPI code, but the number of MPI processes in the hybrid (1 MPI) code is 8 times smaller than in the pure MPI code, so the total amount of data transmitted is quite similar. The same applies for the hybrid (2 MPI) code, which transmits around 3.73 times more data per process than the pure MPI code, but uses 4 times fewer MPI processes, so leading to a similar amount of data transmitted. Table 4.5 shows the ratio between the average data sent per process for each class of code. Whenever this ratio approaches the reverse of the ratio between the number of MPI processes used in each class of code, the total amount of data transmitted will be similar between the two.

The first communication phase of the application is clearly affected by the number of processes and the number of cells within a subdomain in each process. A pure MPI code running on a cluster of multi-core nodes will have many processes with a small number of cells. The communication phase for these processes will involve smaller message sizes, but a larger number of messages. A hybrid code running on the same cluster will have fewer processes, each

with larger numbers of cells to communicate. The communication phases for these processes in the hybrid code will therefore involve larger message sizes, but fewer messages over the whole cluster. This relationship between the relative halo sizes of subdomains and the numbers and sizes of messages has a large effect on the communication phases of the two codes; this difference in communication profile results in the communication times depending heavily on the interconnect used. There is a clear relationship between the number of messages and the interconnect latency, and the message size and the bandwidth. A faster latency will provide better performance with larger numbers of messages (and so a slower latency will perform worse with larger numbers of messages) while a larger bandwidth connection will provide better performance when dealing with large message sizes than a smaller bandwidth connection will.

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

Communication Timing Results

As the communication phases of the code are where the major differences between the pure MPI and hybrid timing profiles occur, the interconnect used for communication has a significant effect on this phase of the code and therefore the overall timing. Figure 4.13 shows the timings for the `movout` routine on Merlin for all three simulation sizes at 128 cores, while Figure 4.14 shows the same routine on CSEEM64T at 96 cores.

On Merlin the performance difference between the pure MPI and the hybrid message passing + shared memory model is clear to see, the smaller messages being sent in the pure MPI code are transmitted far more quickly than the larger messages in both the hybrid (1 MPI) and hybrid

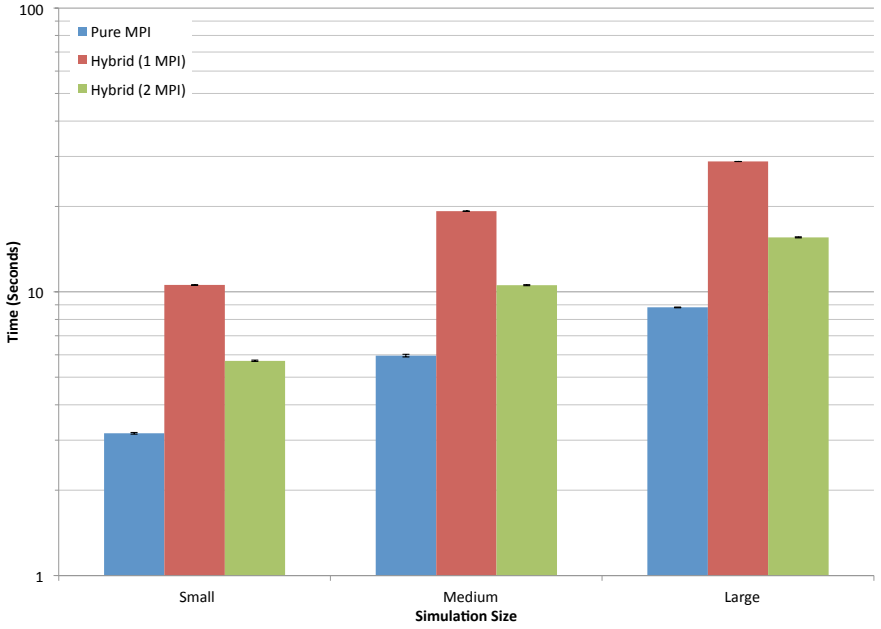


Figure 4.13: Movout routine timing on Merlin at 128 cores, presented with error bars of 1 SEM. Hybrid (1 MPI) code consistently slower than pure MPI code.

(2 MPI) codes, leading to a much faster `movout` routine timing in the pure MPI code, even though there are more MPI processes taking part in the communication. The low latency of the interconnect means that there is no advantage to having fewer processes, as communications are quick to setup. Relating the communication to the performance of the interconnect seen in Section 3.1.4 indicates that the smaller message sizes in the pure MPI code are transmitted at a higher bandwidth than the larger messages in the hybrid message passing + shared memory code. Both factors together results in the large performance difference, and in the pure MPI code outperforming both the hybrid codes. Standard errors to the mean are relatively small, showing that no large variance was seen between the runs performed on Merlin.

On CSEEM64T, as expected, the difference between the standard Gigabit Ethernet interconnect and the high-end Infinipath interconnects can be quite large, with the GigE interconnect consistently much slower than the faster HPC interconnect. For example, just comparing each class of code over both interconnects, the pure MPI code running the small simulation over the Gigabit Ethernet connection is about 17 times slower than when running over the Infinipath

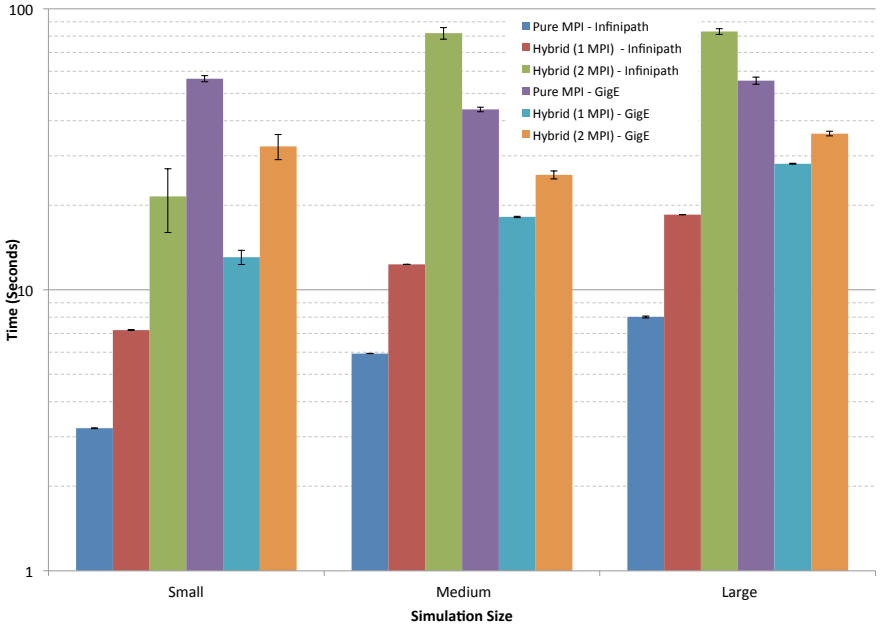


Figure 4.14: Movout routine timing on CSEEM64T at 96 Cores. Both hybrid cases slower than pure MPI over Infinipath, but faster than pure MPI over GigE.

interconnect. For the large simulation, the difference is about 6 times. The hybrid (1 MPI) code fares a little better than this however. For all size simulations on CSEEM64T, the difference is quite small for the hybrid (1 MPI) code, being somewhere between three and ten seconds when comparing the code across the two different interconnects.

These results demonstrate that the pure message passing code spends less time in the communication phase than the hybrid message passing + shared memory codes when using the Infiniband and Infinipath interconnects, but that the reverse is true when using a Gigabit Ethernet connection. Here the higher number of processes in the pure MPI code decreases the performance of the communication sections of code as more messages take far longer to set up due to the increased latency. The lower bandwidth of the connection would seem to suit the smaller messages in the message passing code, but in practice this is not the case, suggesting that the larger messages in the hybrid message passing + shared memory codes suit the performance of the interconnect better.

The results also show that in the hybrid (1 MPI) case the interconnect used is of less importance

than in the pure MPI code, as the performance is quite similar on both interconnects.

On CSEEM64T both the hybrid (1 MPI) and hybrid (2 MPI) `movout` code are faster than the pure MPI when running over the Gigabit Ethernet connection.

In some cases on CSEEM64T the standard errors to the mean are relatively large. For both the pure MPI and hybrid (1 MPI) code running on the infinipath interconnect as seen in Figure 4.14 the errors are small when compared to those seen with the hybrid (2 MPI) code on the same interconnect. This is also seen on the Gigabit Ethernet connection, where the standard errors seen with the hybrid (2 MPI) case are larger than those seen with either the pure MPI or hybrid (1 MPI) case. This suggests that much more variance is seen between the runs of the hybrid (2 MPI) code, especially when using the Infinipath connection, although the cause of this variance is not clear. For the pure MPI and hybrid (1 MPI) code, as with the overall timings, the standard errors are again larger over the Gigabit Ethernet connection than over the Infinipath connection. Again a possible cause for this is the delay tolerant nature of the network connection leading to increased variance between runs.

This difference in the `movout` routines on the two interconnects is also shown well in Figure 4.15, which clearly shows the scaling of the `movout` routine timing as the number of cores increases on the CSEEM64T cluster on both interconnects for the large simulation. For Infinipath, the MPI `movout` code is consistently faster than the hybrid (1 MPI) `movout` routine, but on Gigabit Ethernet this is only true at 16 cores. For all other core counts the hybrid (1 MPI) code spends less time in the `movout` routine than the pure MPI code. Of note is the significant increase in communication time for the pure MPI code running on the Gigabit Ethernet connection seen when moving from running on 32 cores to 48. A possible cause for this could be the increase in number of processors resulting in code running on nodes separated by a further distance in the communication network. This could be tested by running the code over a similar network, controlling the nodes on which the processes are run to see if the timing is affected by running the code on nodes that are topologically close together in the network or on nodes that are topologically further apart.

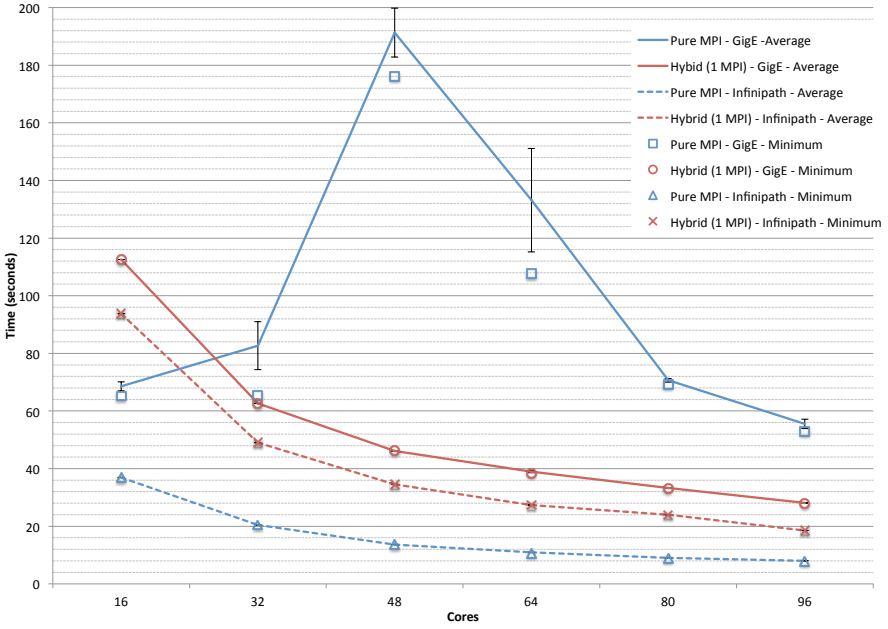


Figure 4.15: Movout routine timing in large simulation on CSEEM64T over both Infinipath and GigE interconnects. Pure MPI consistently faster over Infinipath, but hybrid (1 MPI) faster than pure MPI over GigE above 16 cores.

A second feature of note in Figure 4.15 is the relatively large standard error seen in the pure MPI results over the Gigabit Ethernet connection. Again, as already mentioned in the overall timing discussion in Section 4.3.2 this could be caused by the delay tolerant nature of the Gigabit Ethernet network, so that when the resource is contested between several MPI processes, delays in communication occur. This could account for a large variance in timing results of communication routines, leading to large errors as seen in these results. Such variance is not seen in the hybrid (1 MPI) results, which uses far fewer MPI processes (and so has reduced contention on the interconnect), suggesting the interconnect is the cause of the variance.

The same data for the Infiniband connection on Merlin is shown in Figure 4.16, showing the pure MPI code's consistently better performance. Here the standard errors to the mean are again relatively small, demonstrating that variance between the timing results observed on all three runs was small.

An interesting result is found by examining the timing of the collective communications in the

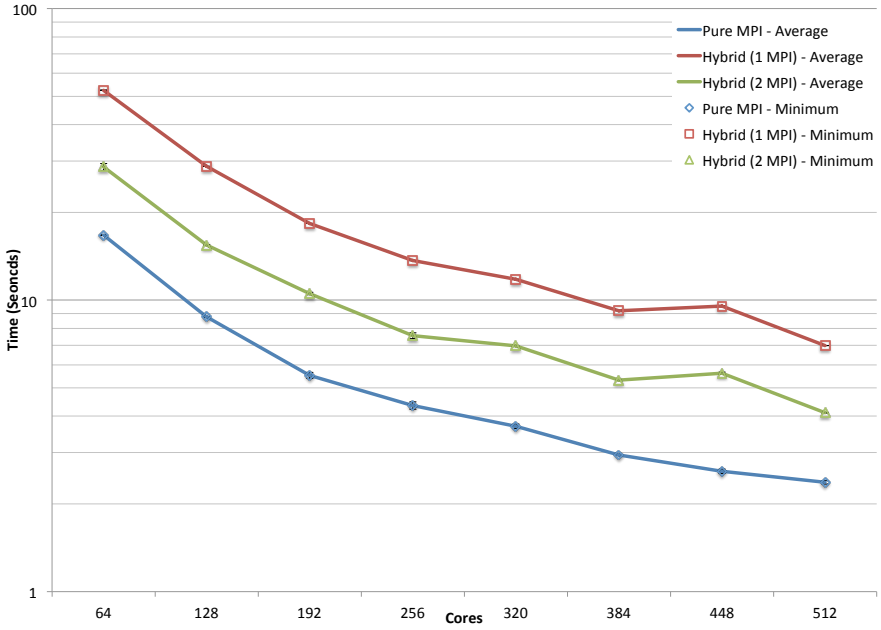


Figure 4.16: Movout routine timing in large simulation on Merlin over Infiniband interconnect. Pure MPI code consistently faster than hybrid (1 MPI) and hybrid (2 MPI).

`sum` routine. Figure 4.17 shows the timing of this routine on Merlin at all core counts. The hybrid (1 MPI) code is consistently faster than the pure MPI code for almost all core counts, as is the hybrid (2 MPI) code. It is interesting to see that at almost all core counts (and certainly all above 256 cores) the hybrid (1 MPI) and hybrid (2 MPI) codes are more than twice as fast as the pure MPI code on the Infiniband interconnect, as over this interconnect the hybrid code is much slower overall than the pure MPI. This performance is purely related to the number of processes; as there are far fewer processes needing to communicate globally in the hybrid codes the performance of collective communication performed in the `sum` routine is significantly better in these codes. However this routine and global communications as a whole are not a significant part of the total runtime, so this routine does not have a major effect on the total time of the application. This does indicate however, that for a parallel code with a higher reliance on collective communications there is a possibility that the hybrid codes could outperform the pure MPI code even over a low latency high bandwidth interconnect.

Standard errors seen with this routine are relatively large in some cases, particularly for the

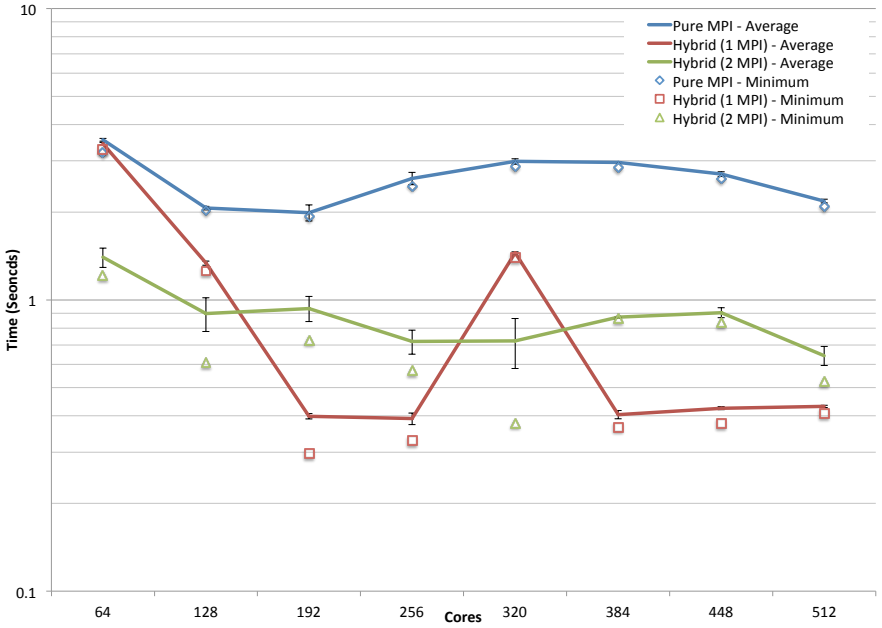


Figure 4.17: Sum routine timings for the large simulation on Merlin. Hybrid codes significantly outperform the pure MPI code due to reduced numbers of processes needed to communicate.

pure MPI and hybrid (2 MPI) code. (It is important to note the logarithmic scale distorts the standard error bars making comparison between code classes difficult - for instance at 256 cores the value of the standard error to the mean for both pure MPI and hybrid (2 MPI) is very similar, although at first glance it appears very different.) A larger variance between the timings seen with this routine might be expected, as collective communication is more sensitive to delays in communication and load imbalance and is essentially latency bound.

4.3.4 Shared Memory Overheads

Contrasting the performance of routines in the hybrid code containing OpenMP parallelisation with their performance in the pure MPI code where no OpenMP parallelisation is present reveals the effect of the OpenMP additions on performance.

The `forces` routine contains two for loops that have been parallelised with OpenMP in the hybrid version. If the performance of the MPI version is taken as a baseline, the difference in

	Hybrid (1 MPI)		Hybrid (2 MPI)	
Cores	Absolute	Percentage	Absolute	Percentage
64	16.49761	14.24%	5.65062	5.38%
128	1.66713	3.18%	0.95764	1.85%
192	0.01156	0.03%	0.55513	1.60%
256	-0.09332	-0.37%	0.69562	2.64%
320	2.25459	9.92%	3.42417	14.32%
384	0.27799	1.59%	0.76414	4.24%
448	0.08347	0.56%	1.14503	7.21%
512	0.14007	1.06%	0.69516	5.07%

Table 4.6: Absolute difference between minimum observed pure MPI timing and minimum observed hybrid timing for the forces routine on Merlin running the small simulation. Absolute differences given in seconds.

performance of the hybrid code can be calculated by subtracting the runtime of the MPI version from the runtime of the hybrid code. Table 4.6 shows the difference for the small simulation, while Table 4.7 shows the difference for the large simulation. As well as the absolute timing difference between the hybrid and MPI code, the percentage of the hybrid method runtime that this difference represents is also shown. The absolute difference for both the small and large simulation are illustrated graphically in Figure 4.18, which shows the difference between the average timing as a line plot, where error bars are given showing the sum of the errors from both hybrid and pure MPI forces measurements and also shows (as a marked point) the difference between the minimum observed performance measurements. Standard errors seen are again relatively small, suggesting that the variance between the timing of the three runs is small when considering the runtime of the forces routine. This would be expected, as the routine is deterministic and entirely node bound, involving no communication. Sources of potential variance in runtime are therefore limited.

There is a clear correlation between the number of cores and the difference in performance of the hybrid and pure MPI code. On 64 cores, the difference between the two is relatively large, but this difference shrinks rapidly as the number of cores increases. This suggests that part of the performance differences is due to the size of the problem (or subdomain) on each node. When running on a low number of cores (and therefore with a low number of MPI processes),

	Hybrid (1 MPI)		Hybrid (2 MPI)	
Cores	Absolute	Percentage	Absolute	Percentage
64	93.98499	25.76%	44.91361	8.16%
128	17.98513	11.58%	7.35597	8.00%
192	4.24421	4.36%	1.15308	4.63%
256	4.09148	5.65%	3.5239	7.85%
320	3.21845	5.57%	2.93613	9.67%
384	1.56781	3.33%	2.06792	6.99%
448	8.05073	17.13%	7.00104	37.30%
512	1.86674	5.17%	2.2139	14.21%

Table 4.7: Absolute difference between minimum observed pure MPI timing and minimum observed hybrid timing for the forces routine on Merlin running the large simulation. Absolute differences given in seconds.

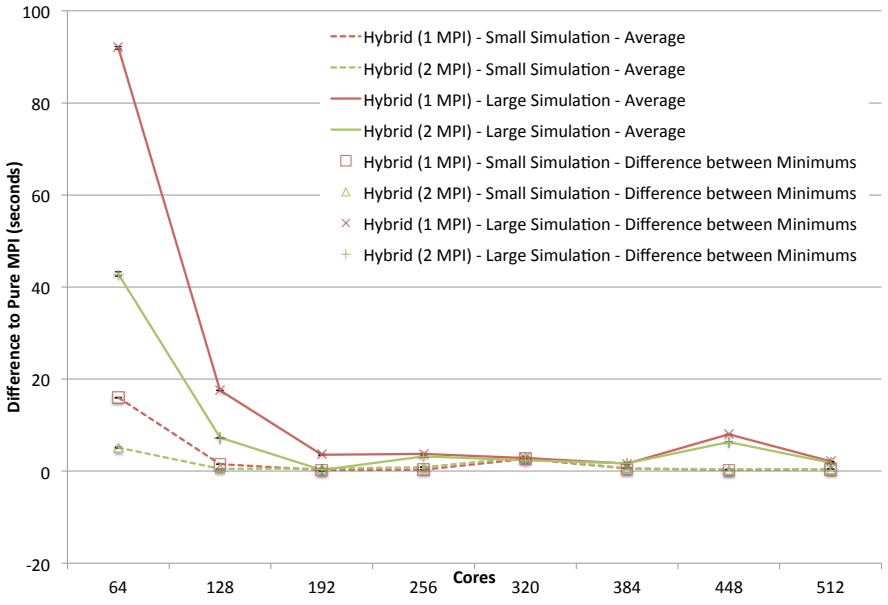


Figure 4.18: Absolute difference between average pure MPI timing and average hybrid timing for the forces routine on Merlin (line), presented with error bars representing the sum of errors on both forces timing measurements, and difference between both minimum observed timing for both hybrid and pure MPI code (markers) . Large difference between codes at small counts decreases rapidly as the number of cores increases.

the subdomain on each node in the hybrid version will be relatively large when compared to the subdomains in the pure MPI code, meaning the work needed to be done in the main work loop of the `forces` method will also be relatively large. As the number of cores increases, the

subdomain shrinks, and the work needed to be done becomes less. Similarly, the amount of data to be copied into private copies within each OpenMP thread will be much larger at lower core counts than at larger core counts; the OpenMP benchmarks in Section 3.1.4 have already shown that this adds increased overheads. The loop performance and OpenMP overheads therefore reduce as the number of cores increases and the performance approaches that of the MPI version. A further possible cause of performance differences could be the simple parallelisation strategy used in the code. In order to see if simple hybridisation approaches can deliver performance improvements, significant restructuring of the code has not been done, only simple loop level parallelisation has been added to the hybrid version. The memory structures and data layout may therefore not be optimum for shared memory parallelisation of this type, which may cause an extra performance hit in the hybrid message passing + shared memory codes. These indirect overheads of the OpenMP parallelisation are less of an issue as the numbers of nodes increases and the work done by each thread decreases. In order to test this theory it would be necessary to carry out a much more in depth restructuring of the code to see if performance improvements can be gained by altering the memory/loop structures.

4.3.5 Routine Breakdowns

Examining the breakdown of runtime between routines for the large simulation on both 512 and 128 cores (Table 4.8) shows that the main differences between the hybrid (1 MPI) and pure MPI codes occur in the `forces` and `movout` routines. Both routines have a longer runtime in the hybrid code. 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, as already discussed.

An examination of the routine timings breakdown over both interconnects on CSEEM64T, running on 96 cores (Table 4.9) shows two things. First, the `forces` routine is again slower in the hybrid code than in the pure MPI code. Second, one of the main differences between the two codes occurs in the timing of the `movout` routine, as on the Merlin cluster. Using the Infinipath

	512 Cores		128 Cores	
	Pure MPI	Hybrid (1 MPI)	Pure MPI	Hybrid (1 MPI)
Forces	34.06927	36.18235	137.7295	155.33234
Movout	2.37228	6.98469	8.75808	28.76653
MoveA	0.70046	0.94867	8.57629	8.33893
MoveB	0.39769	0.6112	4.25237	5.21269
Sum	2.09433	0.40785	2.02575	1.25527
Hloop	0.07061	0.07222	1.97784	1.99515
Startup	2.87173	3.18326	3.04294	3.42531

Table 4.8: Minimum observed routine timing breakdown for the large simulation running over the Infiniband interconnect on Merlin. (Times in seconds).

connection the `movout` routine is much faster in the pure MPI code than the hybrid code. When using the Gigabit Ethernet connection the situation is reversed, with the hybrid code performing much better in the `movout` routine than pure MPI code. 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. These results are again illustrated graphically, in Figure 4.19. As previously discussed, the increase in the timing of the Forces routine between the pure MPI and hybrid message passing + shared memory codes is due to increased overheads in the main work loop contained within this routine, overheads that are not present within the pure message passing code. Similarly, the movout routine timing differences are caused by the change in communication pattern between the two codes. Over the Infinipath connection there is little benefit seen from the reduced number of MPI processes in the hybrid code, with the fewer larger messages actually taking longer to transmit over this connection than the many smaller messages in the pure MPI code. However, over the Gigabit Ethernet connection (which has a much higher latency and is therefore less suited to communication with large numbers of messages) the hybrid message passing + shared memory code has a clear benefit, with the runtime of this routine decreasing significantly between the pure MPI and hybrid codes.

	Infinipath		Gigabit Ethernet	
	MPI	Hybrid (1 MPI)	MPI	Hybrid (1 MPI)
Forces	193.77336	207.99323	194.5966	207.18975
Movout	7.90706	18.5059	53.00865	27.87451
MoveA	7.377	12.5858	7.42247	12.57763
MoveB	3.64191	5.08047	3.48986	5.05727
Sum	0.90146	0.26923	1.3424	1.29131
Hloop	2.07302	2.3668	2.06386	2.3614
Startup	4.10984	3.794	9.39962	9.38928

Table 4.9: Minimum observed routine timing breakdown for the large simulation over both interconnects at 96 cores on CSEEM64T. Times in seconds.

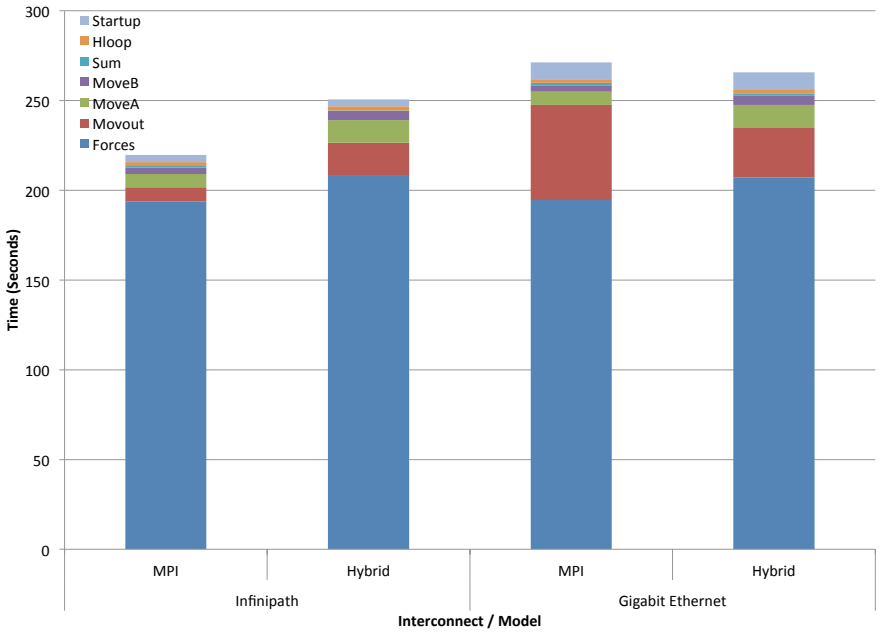


Figure 4.19: Routine timing breakdown for the large simulation over both interconnects on CSEEM64T at 96 cores.

Summary

This chapter has presented a performance analysis of a hybrid message passing + shared memory Molecular Dynamics application. It introduced the main features of the code, and the creation of a hybrid version by adding OpenMP shared memory threading on top of the existing MPI message passing version. It presented the results of a performance analysis of the code running on two multi-core clusters.

The code was not found to be significantly affected by performance issues caused by the use of multi-core processors such as cache sharing or memory bandwidth issues.

When using a high speed low latency interconnect such as the Infiniband connection on Merlin or the Infinipath connection on CSEEM64T the pure message passing code performed better than the hybrid message passing + shared memory code. When using a slower Gigabit Ethernet connection on CSEEM64T the pure MPI code again outperformed the hybrid code, but only on small numbers of processors. On larger numbers of processors the hybrid message passing + shared memory code performed better than the pure message passing code. Performance differences in the code are attributable to two main factors:

Firstly, there is a significant difference in the communication profile of the two styles of code. The message passing version contains point-to-point communication which at high numbers of cores is characterised by many small messages. The hybrid version contains communication phases characterised by far fewer, larger messages. Collective communication, while not a significant contributor to runtime in this application is greatly reduced in the hybrid code. When running on large numbers of processors over a Gigabit Ethernet connection these communication profile differences result in the communication phase of the hybrid message passing + shared memory code taking much less time than the same phase of code in the pure message passing code.

Secondly, there are additional overheads within the hybrid code, caused directly by OpenMP (forking/joining/synchronising threads and overheads introduced from `omp reduction`

and `omp private` clauses) or indirectly (portions of code running with a lower level of parallelisation in the hybrid code than the pure MPI). This results in the main work loops of the application taking longer in the hybrid message passing + shared memory code than in the pure message passing code. However, as the number of cores increases, and the problem size per core decreases, these overheads reduce and become less of a factor than the communication profile.

In general, standard errors to the mean are relatively small when considering the cases of codes running over the Infinipath and Infiniband interconnect. Errors are observed to be larger when running over the Gigabit Ethernet network, this is considered to be due to the nature of the Gigabit Ethernet network 'backing off' when network contention is high and so causing increased variance between the timing of the separate runs.

The following chapter will describe the creation and performance analysis of the second hybrid application examined in the thesis, DL_Poly. This code will attempt to see if the performance differences seen in this chapter are reproduced with a large-scale production MD code.

Chapter 5

Real-World MD Application

Overview

This chapter describes the creation of a large scale hybrid message passing + shared memory code, adapting an existing production MD code: DL_Poly. The application is introduced and its structure described. The hybrid version of the code is then introduced, before performance results on two different multi-core clusters are shown. The performance differences between pure MPI and hybrid code are then examined in detail.

5.1 DL_Poly Introduction

DL_Poly 3.0 is a general purpose serial and parallel molecular dynamics simulation package developed at Daresbury Laboratory [106]. This version of the code uses domain decomposition to parallelise the code and is suitable for large scale simulations on production HPC systems.

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. Several different test cases are available with the application. Test cases 10, 20 and 30 were used for the performance analysis in this chapter as these demonstrated acceptable scaling during initial testing. These test cases are described in Section 5.2.

DL_Poly 4.0

Recently a newer official version of DL_Poly, DL_Poly 4.0, has been released. This version is worth mentioning here as it includes (as an extra set of source files) an un-supported implementation of a hybrid version of some parts of DL_Poly 3.0, including some of the sections of code hybridised in this work. However, the hybrid implementation in DL_Poly 4.0 is targeted to be run on general purpose GPU hardware (GPGPUs) and so consists of CUDA [90] additions as well as some OpenMP parallelisation in order to create a hybrid version of the code [69]. The unsupported hybrid MPI + CUDA + OpenMP version of the DL_Poly 4.0 code is not specifically targeted at multi-core systems, but is targeted to be run on systems comprising multi-core nodes with additional GPGPU nodes. No comparison of either source code or performance has been carried out between this new implementation and the implementation created in this thesis. A performance comparison between the two would make little sense due to the different hardware architectures being targeted.

Method / Library Call	Percentage of Total
Application:User_Code	
dl_poly.f90:MAIN__	100%
dl_poly.f90:dl_poly_.md_vv_	72.38%
two_body_forces.f90:two_body_forces_	66.49%
metal_ld_compute.f90:metal_ld_compute_	27.59%
MPI:MPI_Bcast	21.74%
read_config.f90:read_config_	19.64%
metal_forces.f90:metal_forces_	15.49%
link_cell_pairs.f90:link_cell_pairs_	13.54%
numeric_container.f90:images_	12.93%
MPI:MPI_Allreduce	10.08%
metal_ld_collect_fst.f90:metal_ld_collect_fst_	9.50%
comms_module.f90:comms_module.gcheck_scalar_	7.95%
set_bounds.f90:set_bounds_	5.73%
scan_particle_density.f90:scan_particle_density_	3.23%
set_halo_particles.f90:set_halo_particles_	2.64%
relocate_particles.f90:relocate_particles_	2.50%
deport_atomic_data.f90:deport_atomic_data_	2.41%
scan_field.f90:scan_field_	2.36%
comms_module.f90:comms_module.grsum_vector_	2.13%
parse_module.f90:parse_module.get_line_	2.07%
metal_ld_set_halo.f90:metal_ld_set_halo_	2.00%

Table 5.1: Profile of the DL_Poly application running Test 20 on 4 nodes of Merlin.

5.1.1 Hybrid Version

The hybrid version of DL_Poly implemented for this thesis was created by adding OpenMP on top of the already existing message passing source code. Analysis of the original code using Intel Trace Analyzer and Collector reveals the methods within the code that take the bulk of the runtime, and these are the methods to which effort is focused for adding OpenMP parallelisation in the hybrid version. Table 5.1 shows a flat timing profile of the application code running Test 20 on 4 nodes of Merlin. For brevity, only the methods taking more than 2% of the runtime have been shown.

It is clear to see that the bulk of application time is spent within the `two_body_forces` method, and methods called from within it, so it is these methods in which OpenMP

parallelisation is applied. (Test 10 also uses some other routines including `tersoff_forces`, this is also the focus of OpenMP parallelisation). As with the `forces` routine in the simple MD application, these routines have a general structure including 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.

To add the shared memory threading a parallel region was 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 were used to synchronise and sum contributions to the data values common to all threads such as energy and stress values. An example of the OpenMP parallelisation is included in Listing 5.1, showing the additions to the `two_body_forces` main work loop. It can be seen that the OpenMP additions, in particular the `omp reduction` clause, are much more complicated than the example in the MD code in the preceding chapter. This illustrates the increase in complexity of the DL_Poly code over the previous example code.

As with the MD code in the preceding chapter, the hybrid implementation used is a *master-only* style of hybrid implementation [98]. All MPI communication occurs outside of the OpenMP parallel regions; the hybrid parallel code can therefore be used without any specific support for multi-threading in the MPI library.

It is worth noting in the flat profile in Table 5.1 that there are two collective communication methods, `MPI_Bcast` and `MPI_Allreduce` in the ten most time consuming methods, accounting for 31.82% of the runtime, even at this low count of four nodes. This highlights the increased reliance of this code on collective communication over the simpler MD code in the preceding chapter. This is an important factor in the performance results presented in this chapter. It was shown in the previous chapter that collective communication was one area in which the hybrid message passing + shared memory model was better than pure MPI even on fast low-latency interconnects and it will be shown in this chapter that for codes with a heavy reliance on collective communication this leads to a significant performance difference.

Again, there are several differences in the operation of the pure MPI and the hybrid message

```
1
2 !omp parallel default(none) &
3 !omp private(i,limit,k,j,xdf,ydf,zdf,rsqdf,engacc,viracc,numrdf
    ,new_nz,factor_nz) &
4 !omp shared(natms,list,xxx,yyy,zzz,imcon,cell,rvdw,rcut,alpha,
    epsq,lrdf,leql,nstep,nsteql,ntpmet,ntpvdw,keyfce,nstrdf,
    mxatms,fail,rho,lbook,l_fce,lexatm,megfrz,sumchg,keyens,
    engunit,twobody1start,twobody1,twobody2start,nlast,lsi,lsa,
    volm) &
5 !omp reduction(+:engmet,virmet,engvdw,virvdw,engcpe_rl,
    vircpe_rl,stress,engcpe_ex,vircpe_ex,engcpe_nz,vircpe_nz,
    engcpe_fr,vircpe_fr)
6
7 Allocate (xdf(1:mxatms),ydf(1:mxatms),zdf(1:mxatms),rsqdf(1:
    mxatms), Stat=fail(1))
8 ! outer loop over atoms
9
10 !omp do
11   Do i=1,natms
12             ! do forces calculations
13   End Do
```

Listing 5.1: OpenMP additions to two_body_forces

passing + shared memory code, matching those seen in the previous chapter. Firstly, the hybrid code adds an extra layer of overheads to the molecular dynamics simulation, as in 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. As we use a master-only implementation, only one MPI process per node will carry out message passing communication in the hybrid version. The net result of this is that, as in the previous chapter, 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 5.3).

5.2 Performance Testing

Performance analysis of the DL_Poly code was carried out on the multi-core clusters Merlin and Stella. For details of the hardware, refer to Sections 3.1.1 and 3.1.3. On Merlin the Infiniband network was used for performance testing. On Stella a 10 Gigabit Ethernet connection was used, this has a lower latency and higher bandwidth than the 1 Gigabit Ethernet connection available on CSEEM64T and used with the previous MD code.

5.2.1 Methodology

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, 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.

Test case 10 simulates 343,000 SiC atoms with a Tersoff potential [111], Test case 20 simulates 256,000 Cu_3Au atoms with a Gupta potential [33]. Test case 30 simulates 250,000 Fe atoms with a Finnis-Sinclair potential [35]. These test cases represent some of the larger simulation sizes of the provided test cases, which demonstrated acceptable scaling during initial testing.

The code has been instrumented in order to gain detailed information about the timing of certain parts of the application. Timing is once again done using the portable MPI timer `MPI_Wtime` to measure the total elapsed wall time spent by the master process (the MPI process with rank 0) in each routine. 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 in separate runs using Intel Trace Collector and Analyzer (ITAC) [65] to gather statistics on MPI communication.

5.3 Hybrid Code Performance

As with the previous work using a simpler molecular dynamics code [32] in the last chapter, the performance results show an overall pattern linking the performance of the hybrid code 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 by 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.

The previous work with a simpler molecular dynamics code did not show any benefit to the hybrid model on a fast low latency high bandwidth Infiniband or Infinipath connection, with any benefits only being seen on a slower 1 Gigabit Ethernet interconnect. 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 high bandwidth low latency interconnect such as the Infiniband connection on Merlin. We also see that the hybrid model delivers performance benefits on the slower 10 Gigabit Ethernet connection on Stella, achieving better performance than the pure MPI code at lower processor core numbers than when using the faster interconnect on Merlin.

5.3.1 Overall Timing

Figures 5.1 and 5.2 show the speedup for Test 10 and Test 20 on both clusters respectively. 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 code at the upper limits of the core counts for both clusters. Test 20 exhibits

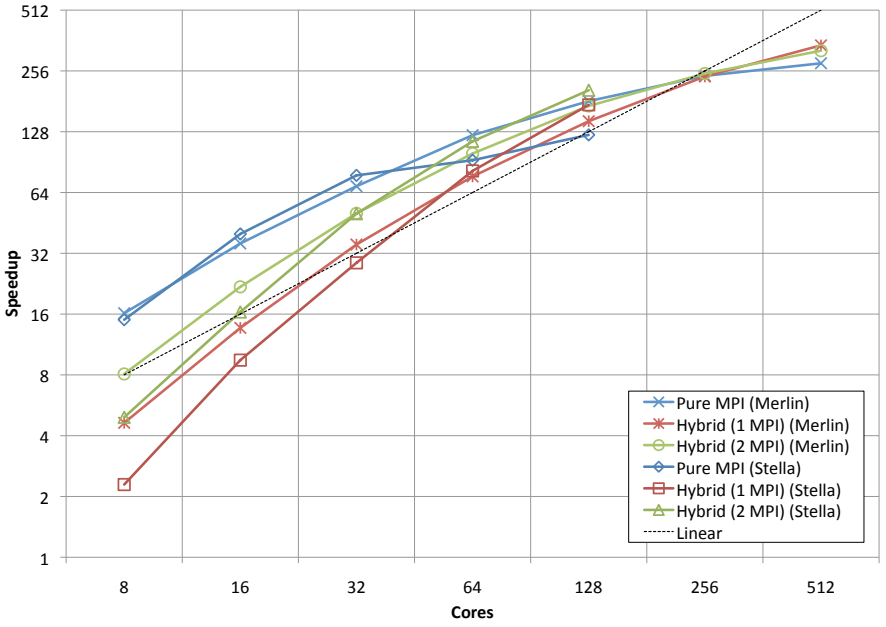


Figure 5.1: Speedup for Test 10 on both clusters for pure MPI, hybrid (1 MPI) and hybrid (2 MPI). Hybrid (1 MPI) and hybrid (2 MPI) achieve higher speedup than pure MPI at larger core counts for each cluster.

much worse scaling than Test 10, but still shows the hybrid codes outperforming the pure MPI codes at higher core counts for both clusters.

Figures 5.3 to 5.5 show the total times for all three tests at all processor core counts on Merlin. All three exhibit the behaviour described above, with pure 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. For Test 10 it occurs at 512 cores, for Test 20 it is at 256 cores and for Test 30 the hybrid performance is better above 128 cores. At these core counts the shared memory overheads are reduced to the point where they no longer have a significant negative effect on the performance of the hybrid message passing + shared memory code while the communication overheads in the pure MPI code have become a significant factor in limiting the scaling of performance. These factors lead to the hybrid message passing + shared memory code outperforming the pure MPI code at and beyond these core counts. Of interest is the scaling performance of Test 30, where at 512 cores the communication overheads of the pure MPI code mean that performance is significantly worse

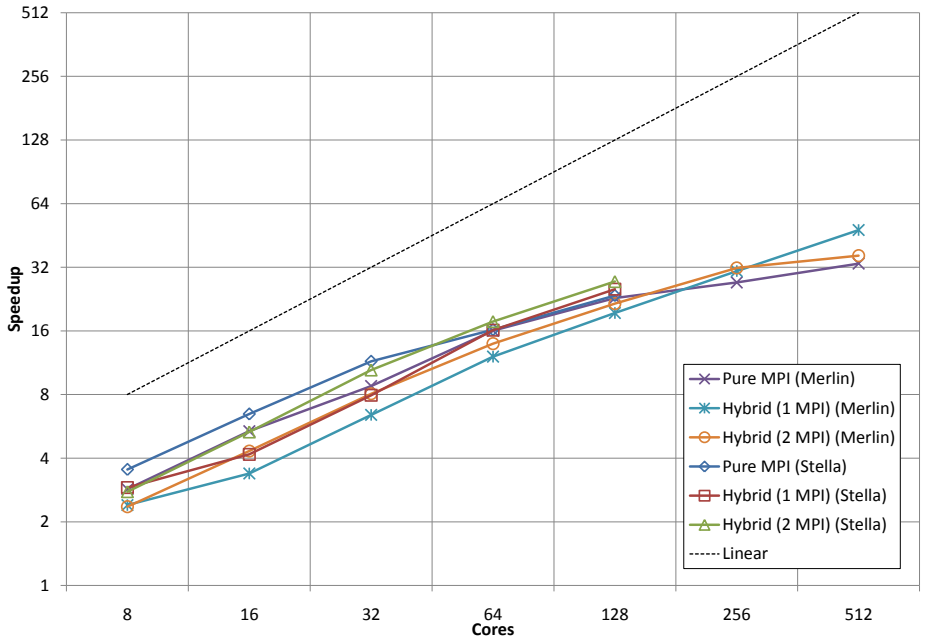


Figure 5.2: Speedup for Test 20 on both clusters for pure MPI, hybrid (1 MPI) and hybrid (2 MPI). Hybrid (1 MPI) and hybrid (2 MPI) achieve higher speedup than pure MPI at larger core counts for each cluster.

than at 256 cores, so the hybrid message passing + shared memory code (with its significantly decreased communication overheads) is able to outperform the pure MPI code by a factor of 5.

The same timing data for the tests running on Stella is shown in Figures 5.6 to 5.8, where a similar pattern can be seen. On this cluster with a slightly lower performance interconnect than the Infiniband connection on Merlin the best performance at higher numbers of cores for both Test 10 and Test 20 is with the hybrid code with two MPI processes per node, while with Test 30 the hybrid code performs better with one MPI process per node. Unlike Merlin, the point where the hybrid code outperforms the pure MPI code is static at 64 cores.

Table 5.2 shows the raw data for the total runtimes for all three tests at all processor core counts on both Merlin and Stella.

In all these results the same basic pattern is observed, attributable to the increased overheads in hybrid message passing + shared memory codes caused by the OpenMP additions, and the reduced communication overhead in the hybrid message passing + shared memory code caused

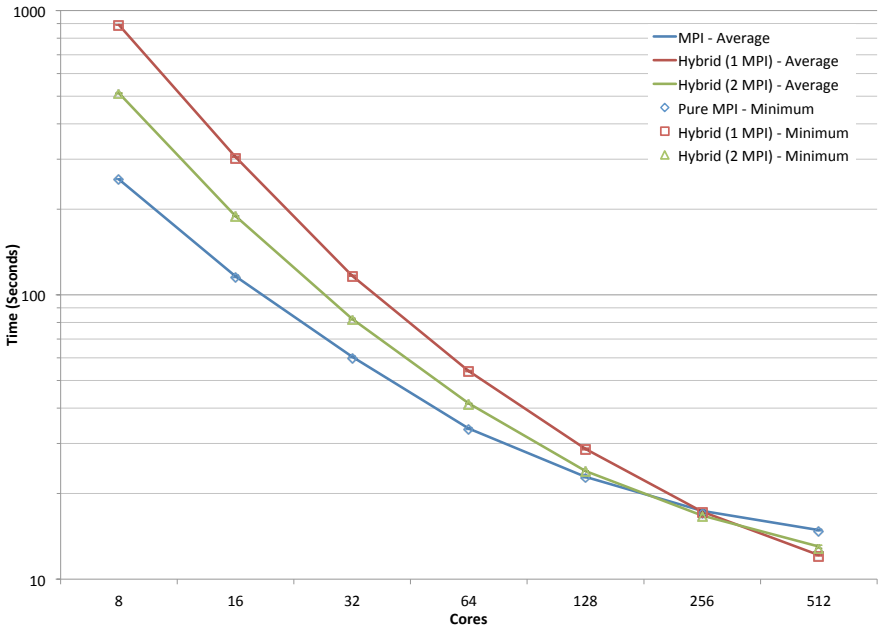


Figure 5.3: Average total time for Test 10 on Merlin, presented with minimum timing and error bars of 1 SEM. Pure MPI code performs better at lower core counts, hybrid (1 MPI) and hybrid (2 MPI) codes perform better than pure MPI above 256 cores.

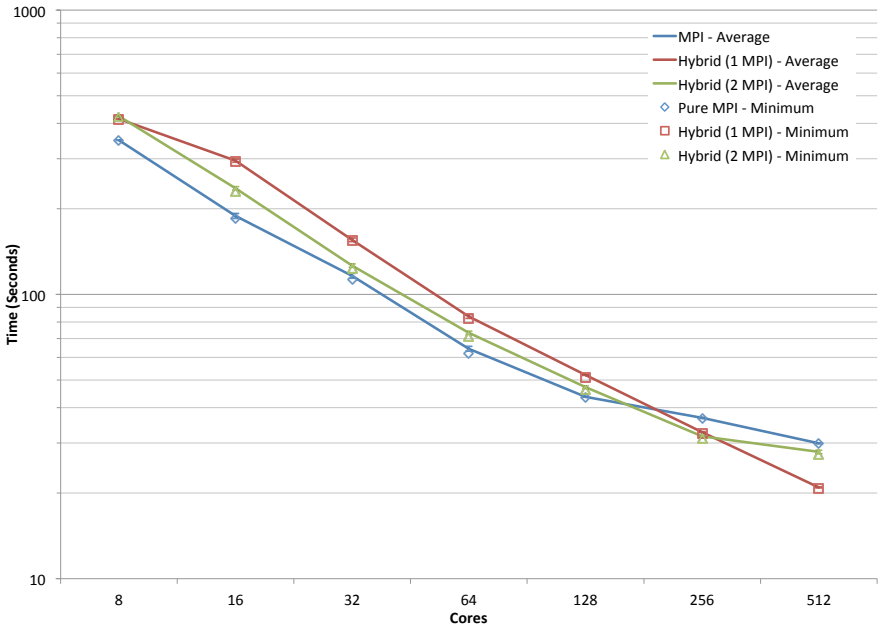


Figure 5.4: Average total time for Test 20 on Merlin, presented with minimum timing and error bars of 1 SEM. Pure MPI code performs better at lower core counts, hybrid (1 MPI) and hybrid (2 MPI) codes perform better than pure MPI above 128 cores.

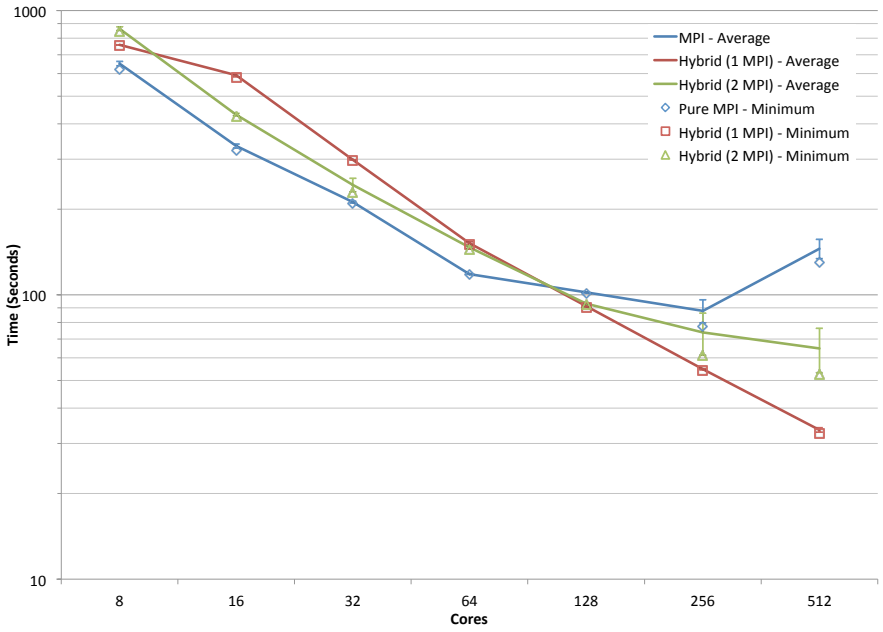


Figure 5.5: Average total time for Test 30 on Merlin, presented with minimum timing and error bars of 1 SEM. Pure MPI code performs better at lower core counts, hybrid (1 MPI) and hybrid (2 MPI) codes perform better than pure MPI above 64 cores.

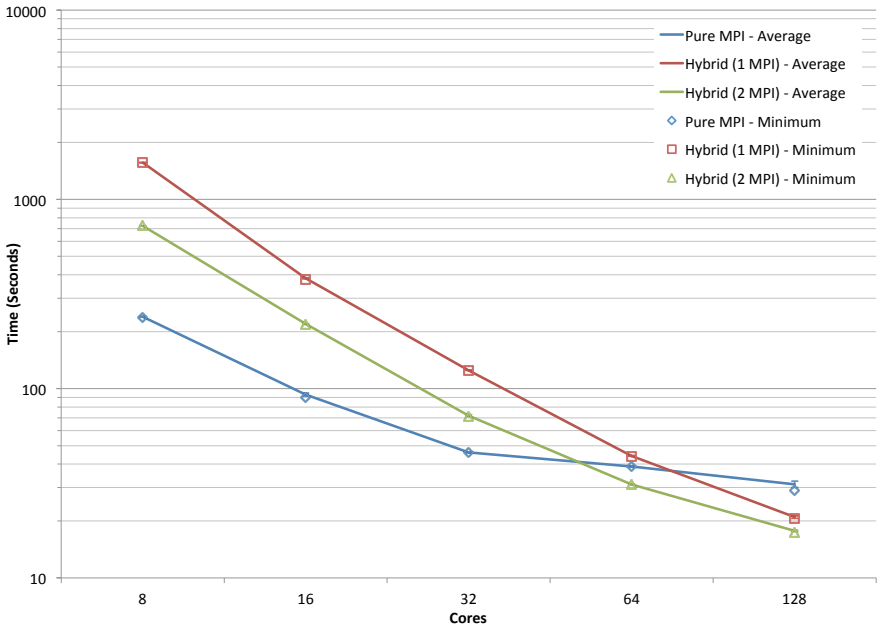


Figure 5.6: Average total time for Test 10 on Stella, presented with minimum timing and error bars of 1 SEM. Pure MPI outperforms both hybrid codes at lower core counts, hybrid (1 MPI) and hybrid (2 MPI) perform better than pure MPI above 64 cores.

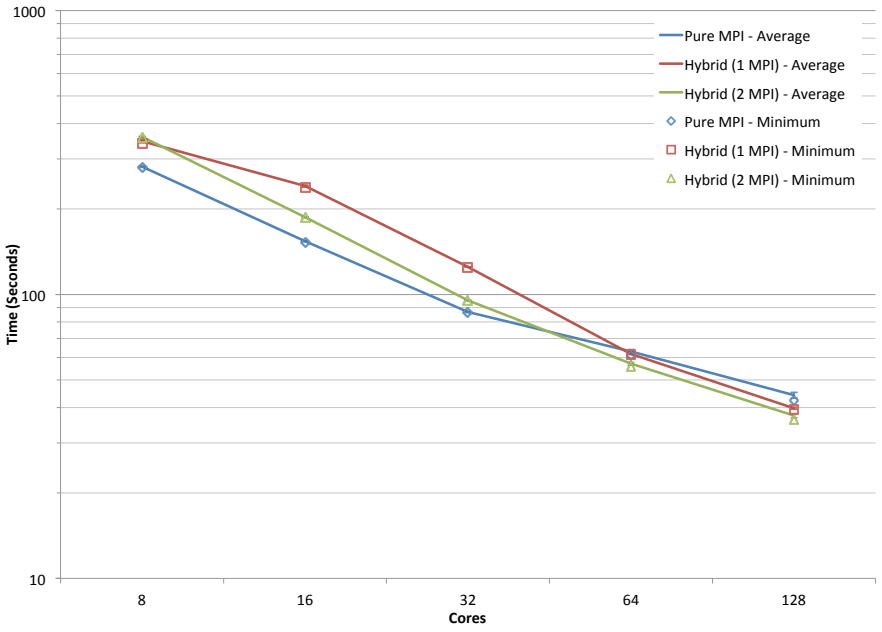


Figure 5.7: Average total time for Test 20 on Stella, presented with minimum timing and error bars of 1 SEM. Pure MPI outperforms both hybrid codes at lower core counts, hybrid (1 MPI) and hybrid (2 MPI) perform better than pure MPI above 64 cores.

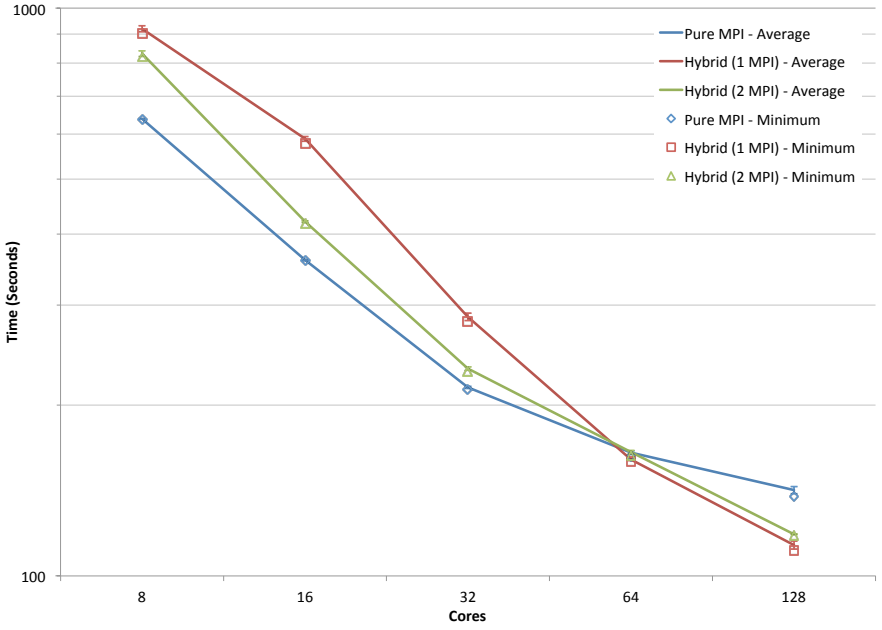


Figure 5.8: Average total time for Test 30 on Stella, presented with minimum timing and error bars of 1 SEM. Pure MPI outperforms both hybrid codes at lower core counts, hybrid (1 MPI) and hybrid (2 MPI) perform better than pure MPI above 64 cores.

by operating with far fewer MPI processes. In all the test cases (but most noticeably in Test 10 on both clusters and in Test 30 on Merlin) the pure MPI code exhibits poor scaling as the number of cores increases. This is due to the increased communication overheads at these large core counts acting as a limiting factor on performance. The hybrid codes do not suffer as much from this communication overhead, so are able to outperform the pure MPI code at these core counts. At lower core counts however the hybrid codes see no performance advantage from their reduced communication overheads, but see a significant performance disadvantage from their increased computational overheads, so perform much worse than the pure MPI code. As the number of cores increases and so the size of subdomain per thread decreases these shared memory overheads have less of an impact upon performance.

Standard errors to the mean are relatively small across all three test cases on both clusters (showing that a small variance in runtime was observed over all three samples for each code and core count) with the notable exception of Test 30 on Merlin. Here the standard error for both the pure MPI and hybrid (2 MPI) increases as the number of cores increases, so that they are relatively large at both 256 and 512 cores. This shows that greater variance in timing is seen with these codes at these core counts. However, the gap between the three classes of codes is sufficiently large at 512 cores that even with a larger uncertainty to the mean it is reasonable to conclude that the performance of the hybrid (1 MPI) code is better than the performance of the pure MPI code.

Overall Timing Summary

A similar simple pattern is observed in these performance results as was observed with the previous MD code on the Gigabit Ethernet connection: at low node counts the MPI code outperforms the hybrid codes, and at higher node counts the hybrid codes outperform the pure MPI. However, in this case, as already noted, this occurs even when using a fast low-latency high bandwidth interconnect such as Infiniband.

In order to understand the performance difference between the hybrid and pure MPI codes it is

	Merlin			Stella		
	Pure MPI					
Cores	Test 10	Test 20	Test 30	Test 10	Test 20	Test 30
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)					
Cores	Test 10	Test 20	Test 30	Test 10	Test 20	Test 30
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)					
Cores	Test 10	Test 20	Test 30	Test 10	Test 20	Test 30
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 5.2: Minimum observed total timing results for all three tests on both Stella and Merlin. Times given in seconds and represent the minimum observed total elapsed wall time as recorded by the master MPI process.

again necessary to examine the factors influencing the hybrid 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 5.3.2.

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 spawn/join threads, carry out reduction operations etc. Indirect overheads are those not caused by OpenMP itself. For instance, some parts of the code are run in serial mode 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. These overheads are examined in Section 5.3.3.

5.3.2 Communication Profile

Inter-node communication profiles are very different between the hybrid and pure MPI codes. Simple statistics collected from the running code using Intel Trace Analyzer and Collector illustrates these differences. Again, these runs were carried out separately from the runs used to collect timing data used in the rest of the performance analysis so as not to contaminate raw timing data with extra instrumentation needed for ITAC to operate. Figures 5.9 and 5.10 show the average data transferred per process per time step and the total amount of data transferred between all processes per time step for Test 10 and Test 20.

The same general pattern as is seen with the simple MD code is revealed in the DL_Poly application. 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 pure MPI code. It is interesting to see that for Test 10 below 128 cores the pure MPI has the lowest average data transferred per process, but that at 128 cores and above the pure MPI has the highest average data transferred per process. Overall the communication profile could be generalised by saying that the hybrid code has fewer large messages per time step while the pure MPI code has more smaller messages per time step. 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.

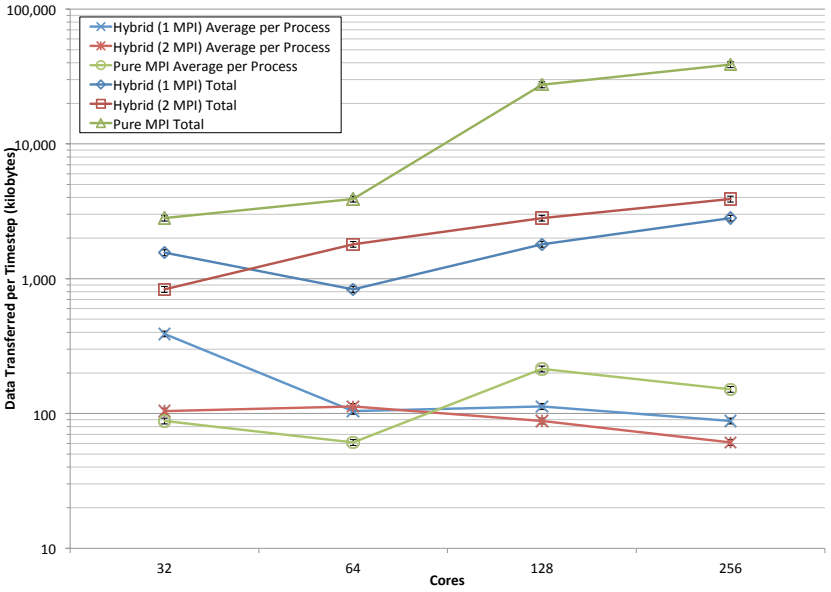


Figure 5.9: Communication profile for Test 10, showing average data transferred per process per time step and total data transferred per time step. 5% error bars shown. Hybrid codes have lower total data transfer per time step than the pure MPI code at all core counts, and lower average data transferred per process per time step above 64 cores.

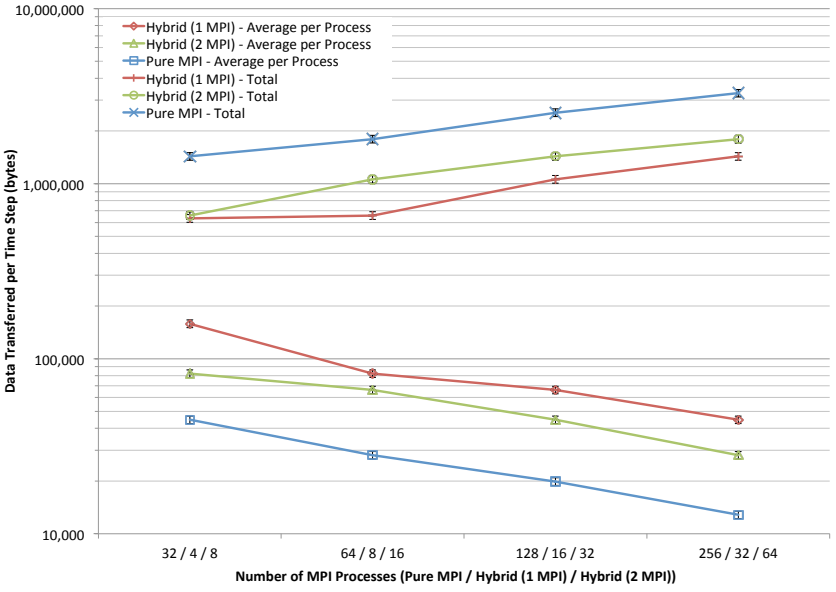


Figure 5.10: Communication profile for Test 20, showing average data transferred per process per time step and total data transferred per process per time step. 5% error bars shown. Hybrid codes have lower total data transfer per time step than the pure MPI code but higher average data transferred per process per time step at all core counts.

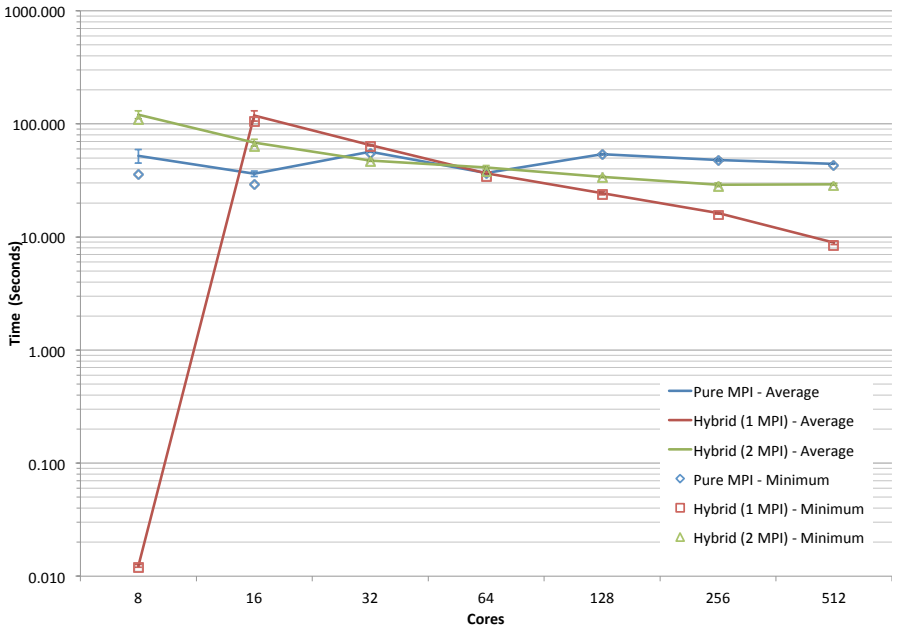


Figure 5.11: Average total communication time for Test 30 on Merlin, presented with minimum timing and error bars of 1 SEM. Above 64 cores the hybrid codes spend less time carrying out communication than the pure MPI code.

Communication Timing Results

Examining the time spent in 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 on Merlin (Figure 5.11) 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, or even increases slightly as the number of MPI processes increases. So, for the pure MPI code, although point-to-point message sizes are decreasing as the number of cores increase, the larger number of MPI processes needing to communicate both point-to-point and collectively negates any advantage gained from the smaller point-to-point message transfers. At 64 cores and above, both the hybrid codes spend less time carrying out communication than the pure MPI code. This illustrates the large advantage provided to the hybrid codes by running with significantly fewer MPI processes: communication time is substantially reduced, even though the size of point-to-point messages to be transmitted are

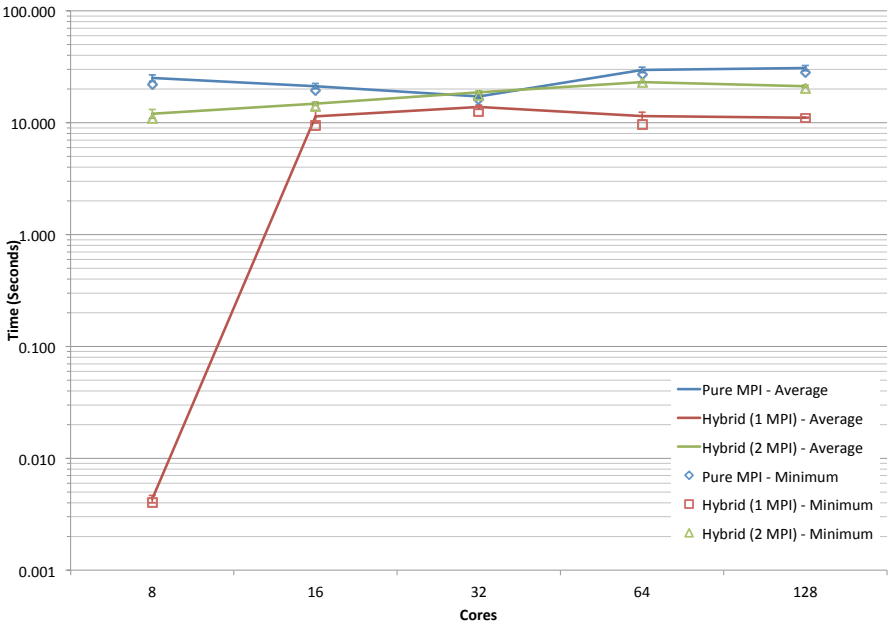


Figure 5.12: Average total communication time for Test 30 on Stella, presented with minimum timing and error bars of 1 SEM. Hybrid (1 MPI) code consistently spends less time carrying out communication than the pure MPI code.

larger in the hybrid code. It is also worth noting that in the pure MPI case the bandwidth available to a node is shared between all 8 MPI processes, while in the hybrid (1 MPI) case the bandwidth is not shared at all, and in the hybrid (2 MPI) case it is shared between only 2 MPI processes. This will have an effect on the time taken to transfer messages, as an individual MPI process in the hybrid codes should be able to transfer data faster than an individual MPI process in the pure MPI case.

Figure 5.12 shows that on the 10 Gigabit Ethernet connection on Stella the total communication time has an overall slight upward trend for both hybrid and pure MPI codes, but that the hybrid message passing + shared memory code spends consistently less time carrying out communication than the pure message passing code. This overall upwards trend in communication time suggests that the interconnect on Stella is less able to cope with increased numbers of communicating processes. On both clusters the hybrid (1 MPI) code has a significant advantage in terms of communication at the lowest core count, as it is running only 1 MPI process. This means there is effectively no communication to be done across the network,

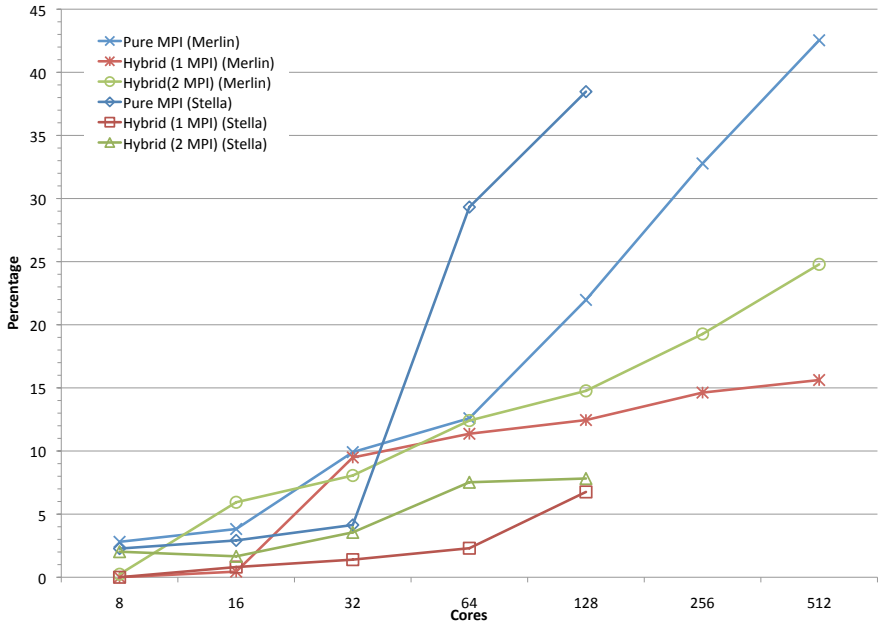


Figure 5.13: Communication time as a percentage of total time for Test 10 on both clusters. Communication makes up less of the total runtime in the hybrid codes on both clusters, with the gap between pure MPI and hybrid codes increasing as the number of cores increases.

so the communication time is very short. Once the code is run on more than one node, the time increases significantly, as this is the first case in which any data will actually need to be transferred.

For all communication timing results the standard errors to the mean are relatively small, on both the Infiniband interconnect on Merlin and the 10 Gigabit Ethernet connection on Stella, indicating a small variance between the runs observed.

Looking at the total time spent carrying out communication as a percentage of the total run time illustrates further the difference in communication pattern between the MPI and hybrid codes. Figure 5.13 shows these values 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. This reveals that the total amount of communication is more of a limiting factor to overall performance in the pure message passing code than in the hybrid message passing + shared memory code. This causes the scalability of the pure message passing code

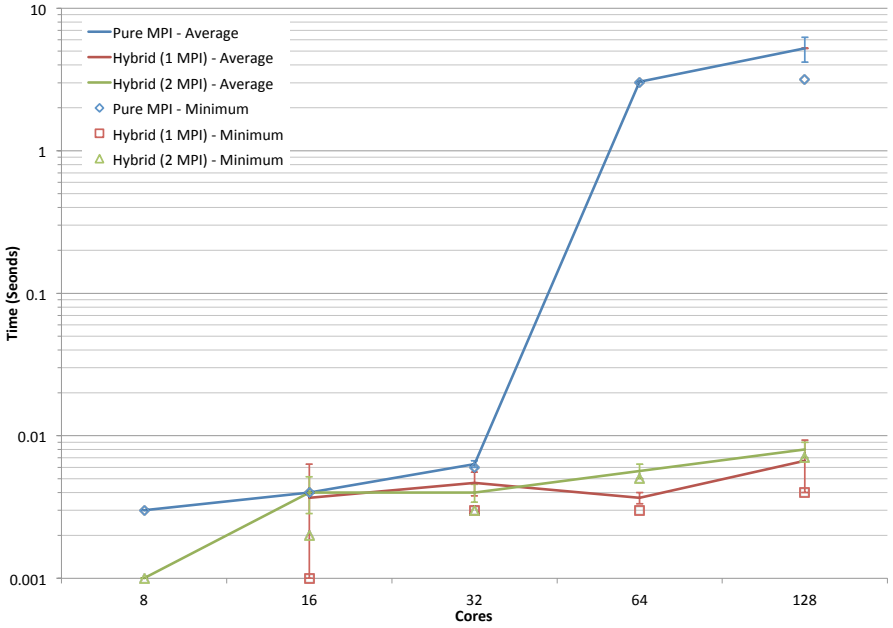


Figure 5.14: Average synchronisation time for Test 10 on Stella, presented with minimum timing and error bars of 1 SEM. Hybrid codes consistently perform better than pure MPI over 10GigE connection.

to be less than the scalability of the hybrid code. As the number of cores used increases, the problem subdomain on each core becomes smaller, and so takes less time to run. The number of processes involved in communication increases however, so the communication time either increases (on Stella) or stays approximately the same (on Merlin). The percentage of runtime spent carrying out communication therefore increases as the number of cores used increases.

In order to examine exactly what is causing these communication performance patterns it is necessary to break down the communication times to see the difference between collective communication, point-to-point communication and barrier synchronisation time, in order to describe their relative differences between hybrid message passing + shared memory and pure message passing codes, and to understand their contribution to the total communication time for both codes.

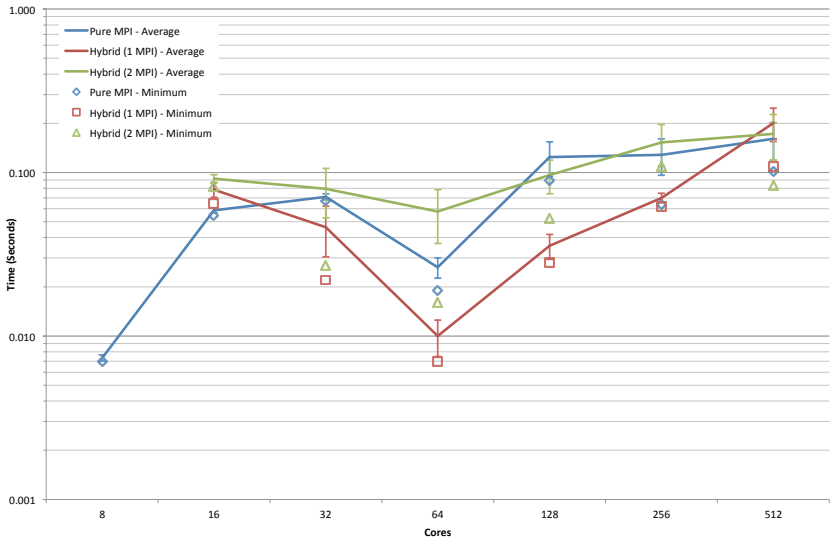


Figure 5.15: Average synchronisation time for Test 10 on Merlin, presented with minimum timing and error bars of 1 SEM. Hybrid codes perform better at most core counts over Infiniband connection.

Synchronisation

Figures 5.14 and 5.15 show the time spent in synchronisation at `MPI_Barrier` calls during a run of Test 10 on both clusters. These times are in general very small, usually making up less than a second of the total runtime. However, they do reveal some interesting differences between the two classes of code and so are worth exploring briefly.

Examining the values of the time spent in synchronisation shows that the hybrid code performs much better on the slower 10 Gigabit Ethernet connection on Stella than the pure MPI code does. This is to be expected, as the smaller number of MPI processes in the hybrid code take less time to synchronise. This is especially true at 64 and 128 cores, where the performance of the synchronisation for the pure MPI code is very poor due to the very large number of MPI processes needing to communicate with one another at the same instance.

Over the faster Infiniband connection on Merlin there is less difference between the codes, but in general the hybrid code performs better here too, except for at larger numbers of cores.

Here, the faster connection with lower latency means there is less difference between the two codes. Also, any imbalance between threads may become more prominent as the work per core becomes smaller, leading to possible imbalance between MPI processes causing slower performance of barrier synchronisation. Overall, synchronisation time is not a large contributor to overall communication time for either of the codes, with the notable exception of the pure MPI code running at a large number of cores over the 10 Gigabit Ethernet connection on Stella, where the performance is much worse than in any other case. This is due to the higher latency of the 10 Gigabit Ethernet connection, as synchronisation is primarily a latency bound operation so any increase in latency will dramatically affect the performance of such synchronisation operations.

Standard errors to the mean in these results are clearly relatively large for most of these results. This is to be expected, the time taken for these operations is very short, so any slight variance between runs will result in larger errors due to the small sample size. It is also worth noting that in several cases (particularly at low core counts on Stella and at both low and high core counts on Merlin) there is an overlap between the standard errors observed for the three cases of code, indicating that the differences between the codes are not statistically significant and that further testing is required to confirm the difference between the codes.

Collective Communication

The time spent carrying out collective communication (other than synchronisation) is shown in Figures 5.16 and 5.17, showing the collective communication time on both Merlin and Stella. A similar pattern to the overall runtime is seen, in terms of the relative performance of both hybrid and pure message passing codes. 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. This is primarily due to the reduced number of MPI processes in the hybrid code allowing better performance for the hybrid code as the number of cores increases.

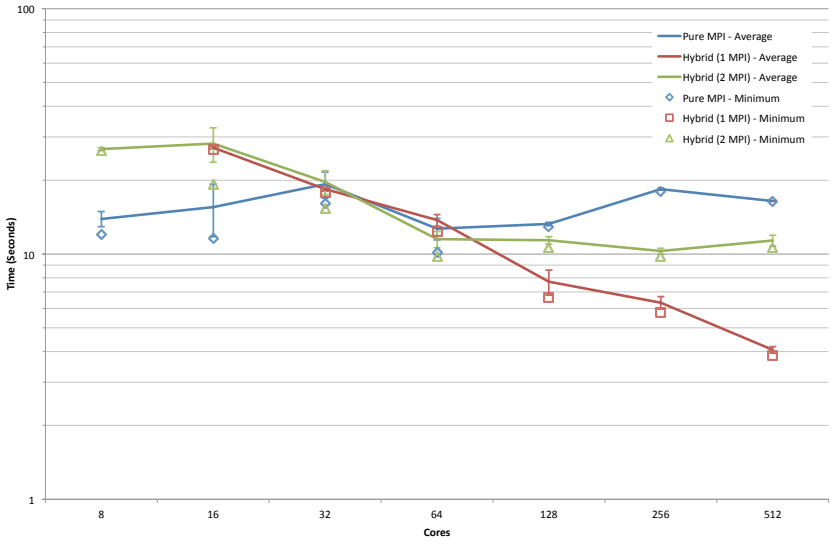


Figure 5.16: Average collective communication time for Test 20 on Merlin, presented with minimum timing and error bars of 1 SEM. Below 64 cores the pure MPI code performs better than both hybrid codes, while above 64 cores the hybrid codes perform better than pure MPI.

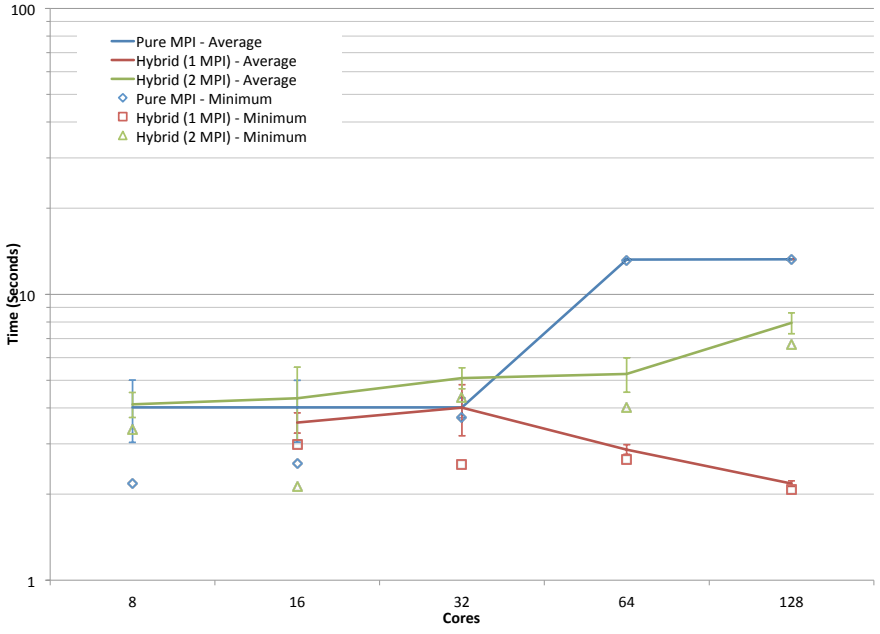


Figure 5.17: Average ollective communication time for Test 20 on Stella, presented with minimum timing and error bars of 1 SEM. Hybrid (1 MPI) code performs better than pure MPI above 16 cores, while hybrid (2 MPI) code performs better above 32 cores.

On Merlin, the poorer performance of the hybrid codes at lower core counts can be attributed to effects of threading overheads causing imbalance between the runtime of the master MPI processes for each thread group. Since the latency of the Infiniband connection is so low, any imbalance between MPI processes is shown in the collective communication performance - it would be hidden on a connection with higher latency (such as the 10 Gigabit Ethernet connection on Stella). At higher core counts, as collective communication is primarily latency bound the reduced number of MPI processes in the hybrid codes are able to perform collective communication more efficiently than the increased number in the pure MPI code (since a smaller number of messages will take a shorter time to set up). Unlike in the previous simple MD code, the message sizes of some of the collective communications in the DL_Poly code do change as the sub-domain size per process changes. This indicates that as the sub-domains decrease in size, the size of messages in the collective communications reduces. However, as can be seen, on Merlin this has little benefit in the pure MPI code, where the performance is relatively stable between 10 and 20 seconds for all core sizes, suggesting that again the decreased performance caused by an increased number of processes is outweighing or balancing the increased performance found with smaller message sizes. This is not the case for the hybrid codes, hence the improved performance. On Stella the performance of the pure MPI code is even worse, suggesting that the negative performance impact of attempting collective communication with many processes over a connection with much higher latency is far outweighing the benefits of transmitting smaller messages. This is even seen in the hybrid (2 MPI) code, which also takes longer to carry out collective communication as the number of cores increases. The hybrid (1 MPI) code is able to decrease the communication time slightly as the number of cores increases, but not as dramatically as on Merlin.

Again, the standard error to the mean can be relatively large for the collective communication, although not to the extent seen with the synchronisation times. An increased variance between different runs may be expected due to the low tolerance of collective communication to communication delays or load imbalance, as described in the previous chapter.

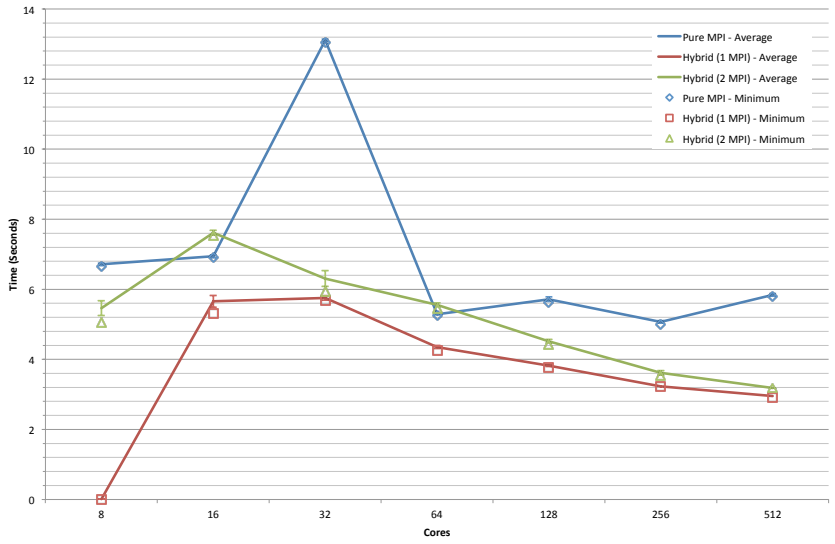


Figure 5.18: Average Point-to-Point communication time for Test 30 on Merlin, presented with minimum timing and error bars of 1 SEM. Pure MPI spends more time carrying out point-to-point communication than hybrid (1 MPI) code at all core counts, and than hybrid (2 MPI) code at all core counts other than 16 and 64.

Point-to-Point Communication

A different pattern is seen in the point-to-point communication time to that observed in collective communication. Test 30 results for the point-to-point communication time are shown in Figures 5.18 and 5.19 for both Merlin and Stella. Here we see that overall the hybrid code spends less time on point-to-point communication than the pure MPI code even at lower core numbers. This is due to the reduced numbers of MPI processes in the hybrid message passing + shared memory code. The pure message passing code has many small messages to be communicated, which takes longer due to extra overheads in setting up messages, and the fact that the available bandwidth is shared between more MPI processes per node than in the hybrid message passing + shared memory code. So, while the bandwidth of a node is shared between 8 MPI processes per node in the pure MPI code, it is not shared at all in the hybrid (1 MPI) code, and is split between only 2 MPI processes in the hybrid (2 MPI) code. While the messages are larger per process in the hybrid code, there is less data in total to be transmitted

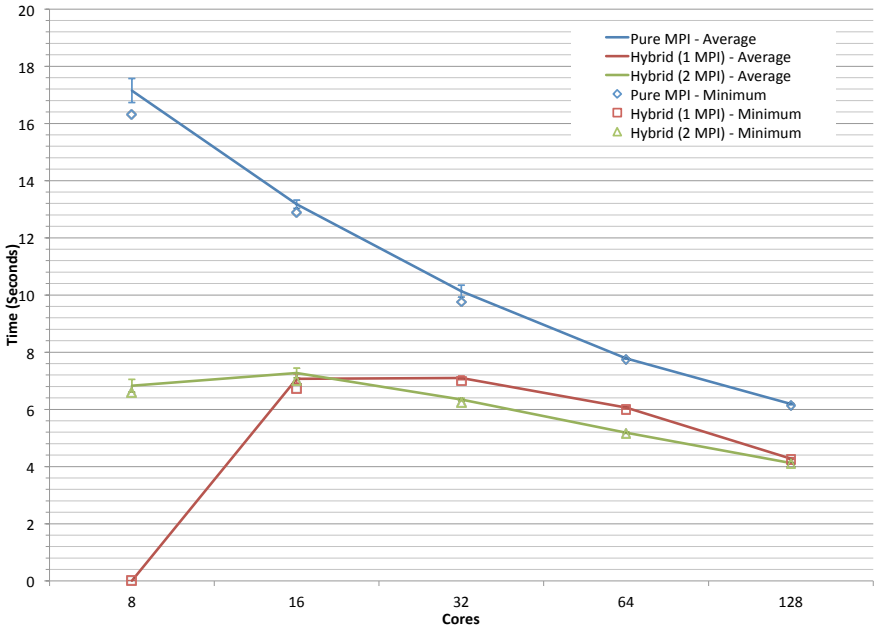


Figure 5.19: Average Point-to-Point communication time for Test 30 on Stella, presented with minimum timing and error bars of 1 SEM. Both hybrid codes spend less time carrying out point-to-point communication than the pure MPI code at all core counts.

per node, as the total data transferred in the pure MPI code is larger due to the increased amount of halo data caused by having a larger number of smaller sub domains. When considered on a node level, this results in the pure MPI code having more data per node to transfer over a connection being shared by more MPI processes, resulting in worse performance than observed in the hybrid codes.

5.3.3 Threading Overheads

An understanding of the direct and indirect overhead introduced by the shared memory threading can be gained 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 directly from the shared memory threading of the loop compared to the pure MPI

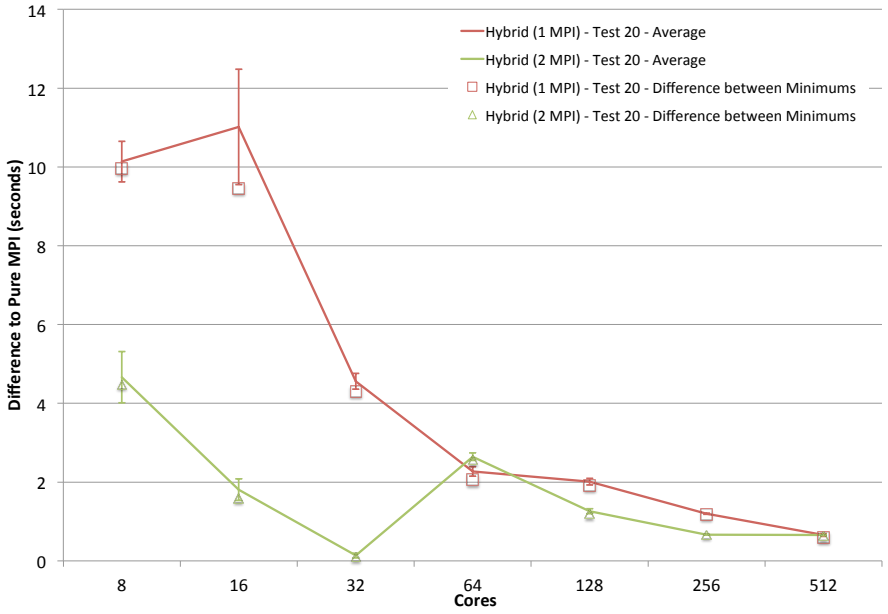


Figure 5.20: two_body_forces loop timing for Test 20 on Merlin. Difference between average pure MPI and average hybrid timings presented as line with errors bars representing combined errors from both measurements. Difference between minimum observed timings presented as marker. Overheads shrink rapidly as the number of cores increases (and subdomain sizes per core decrease).

parallelisation.

Figures 5.20 and 5.21 show this data for Test 20 and 30 on Merlin, while Figure 5.22 shows the same data for Stella.

For Test 20 on both clusters it is possible to observe 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. The same is true for Test 30 on Stella, while on Merlin the average overhead for Test 30 remains fairly constant for all core counts, while the difference between the minimum observed timing follows the same pattern as in the other results. These overheads are therefore less of an issue at the large core counts, where the better performance of the hybrid code is seen. Larger standard errors to the mean are seen at lower core counts, showing that a greater variance in runtime was observed at these core counts than at the upper end of the range. A possible cause for this could be load imbalance between the threads carrying out the work within the loop, which would be a more significant issue at smaller core counts where the subdomains assigned to each thread

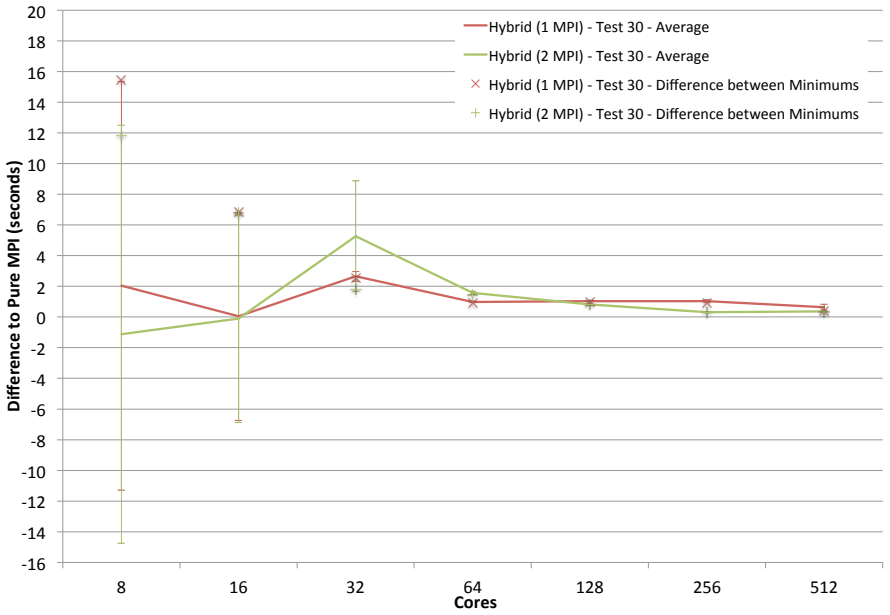


Figure 5.21: two_body_forces loop timing for Test 30 on Merlin. Difference between average pure MPI and average hybrid timings presented as line with errors bars representing combined errors from both measurements. Difference between minimum observed timings presented as marker. Differences between minimums shrink rapidly as the number of cores increases (and subdomain sizes per core decrease).

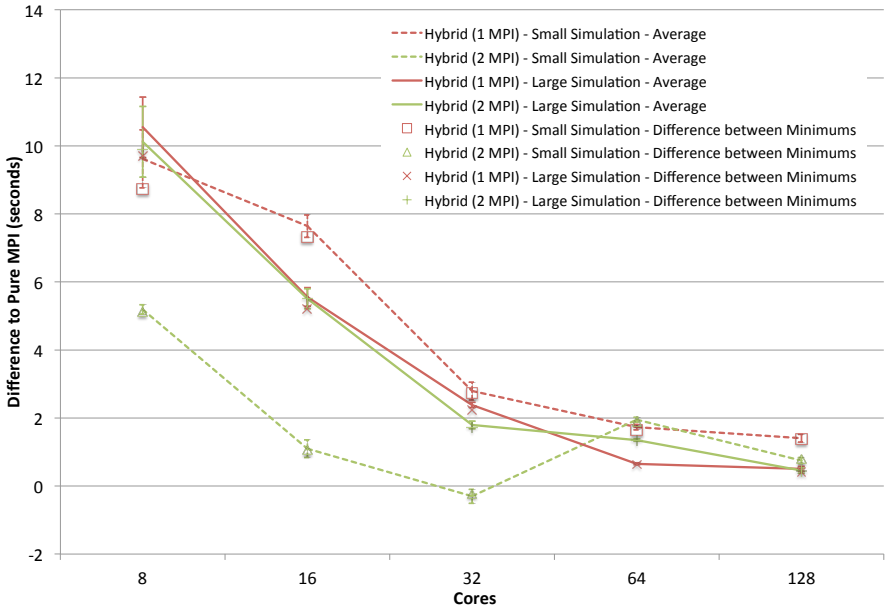


Figure 5.22: two_body_forces loop timing for Test 20 & 30 on Stella. Overheads shrink rapidly as the number of cores increases (and subdomain sizes per core decrease).

	Hybrid (1 MPI)		Hybrid (2 MPI)	
Cores	Absolute	Percentage	Absolute	Percentage
8	9.712	4.71%	9.897	4.80%
16	5.205	4.92%	5.519	5.20%
32	2.238	4.16%	1.72	3.23%
64	0.631	2.33%	1.339	4.81%
128	0.404	2.84%	0.437	3.06%

Table 5.3: Absolute difference between pure MPI and hybrid timing for the main work loop in Test 30 running on Stella. Absolute differences given in seconds.

are larger than it would be at higher core counts where they are much smaller. Further testing examining the amount of data assigned to each thread would allow this to be confirmed or ruled out as the cause of variance.

Data from Test 30 running on Stella and Merlin is given in Tables 5.3 and 5.4, where the absolute timing difference is presented with the percentage of the total runtime this represents. This shows that the percentage of the runtime attributable to extra overheads caused by the hybrid parallelisation fluctuates as the number of cores changes, but that in general they become less important on Stella as the number of cores increases, while they take up a larger percentage of the runtime on Merlin as the number of cores increases, particularly in the Hybrid (1 MPI) case.

This pattern is very similar to that seen in the simple MD code in the preceding chapter. When running on a small number of nodes the subdomain for each MPI process is quite large, and the increased OpenMP overheads (which are more complex in this code, involving a larger number of reductions of data and increased numbers of arrays to be copied into private copies) are a significant drag on performance. As the size of the subdomains decrease, the overheads of reducing arrays and creating temporary private copies are decreased, so the performance becomes closer to that of the pure MPI code.

Rather than just look at the timing of the main work loop, the timing of an individual routine that has been mostly parallelised in the hybrid version can also be examined. The `metal_ld_compute` routine is called from within the `two_body_forces` routine before

	Hybrid (1 MPI)		Hybrid (2 MPI)	
Cores	Absolute	Percentage	Absolute	Percentage
8	15.478	6.48%	11.834	5.03%
16	6.866	5.67%	6.771	5.59%
32	2.538	4.15%	1.806	2.99%
64	0.927	3.02%	1.447	4.63%
128	0.991	6.06%	0.801	4.95%
256	0.914	10.15%	0.285	3.40%
512	0.377	8.24%	0.332	7.33%

Table 5.4: Absolute difference between pure MPI and hybrid timing for the main work loop in Test 30 running on Merlin. Absolute differences given in seconds.

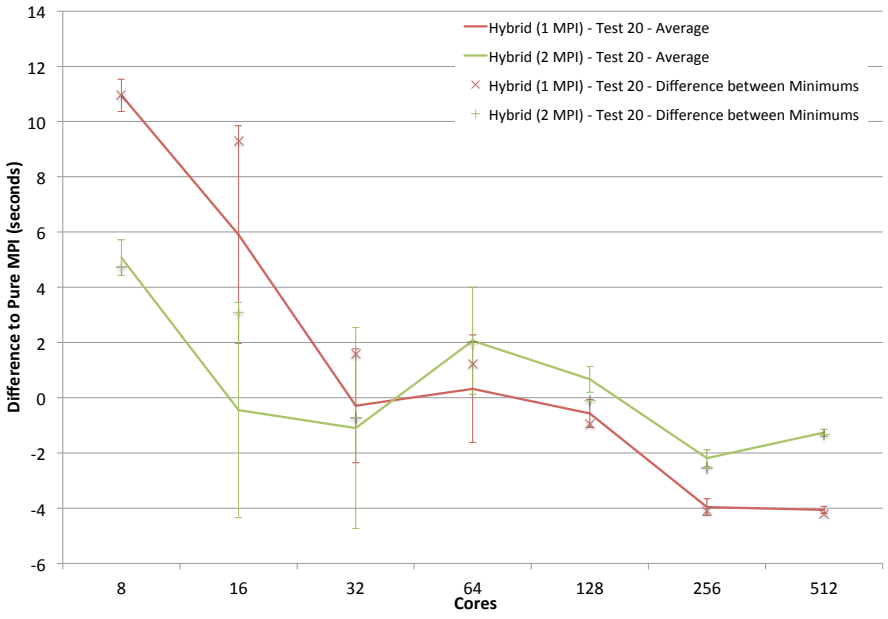


Figure 5.23: Average metal_ld_compute routine timing for Test 20 on Merlin, presented with minimum timing and error bars of 1 SEM. Hybrid code performs better than pure MPI code at high core counts, taking less time to run. Performance difference shrinks rapidly as number of cores increases.

the main work loop and parallel region starts. It contains two loops that have also been parallelised with OpenMP, and some global communication outside of the parallel region. Again, the pure MPI timing is taken as a baseline and the difference between that and the timing of the routine in the hybrid code is calculated. Figures 5.23 to 5.25 show the results for Test 20 and 30 on both clusters.

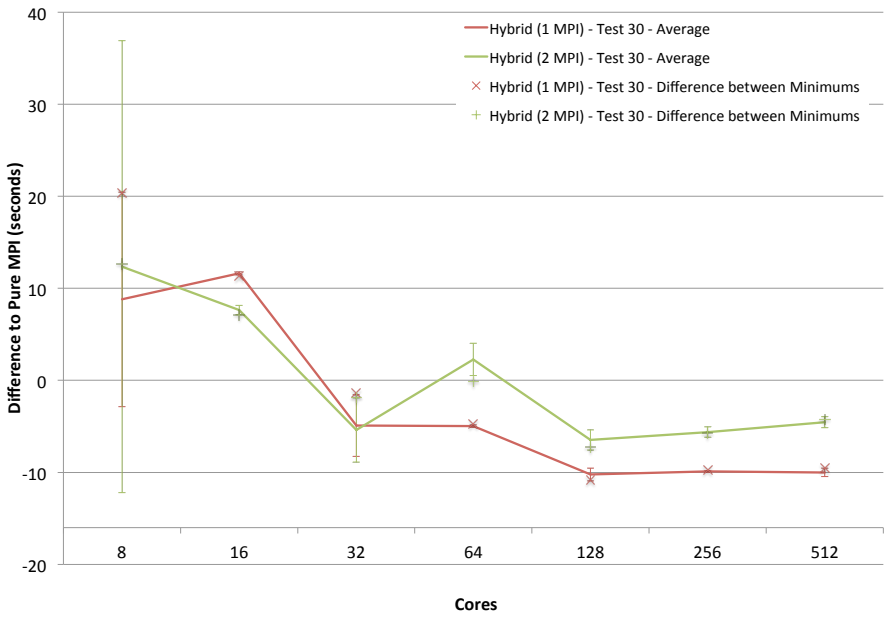


Figure 5.24: Average metal_ld_compute routine timing for Test 30 on Merlin, presented with minimum timing and error bars of 1 SEM, presented with minimum timing and error bars of 1 SEM. Hybrid code performs better than pure MPI code at high core counts, taking less time to run. Performance difference shrinks rapidly as number of cores increases.

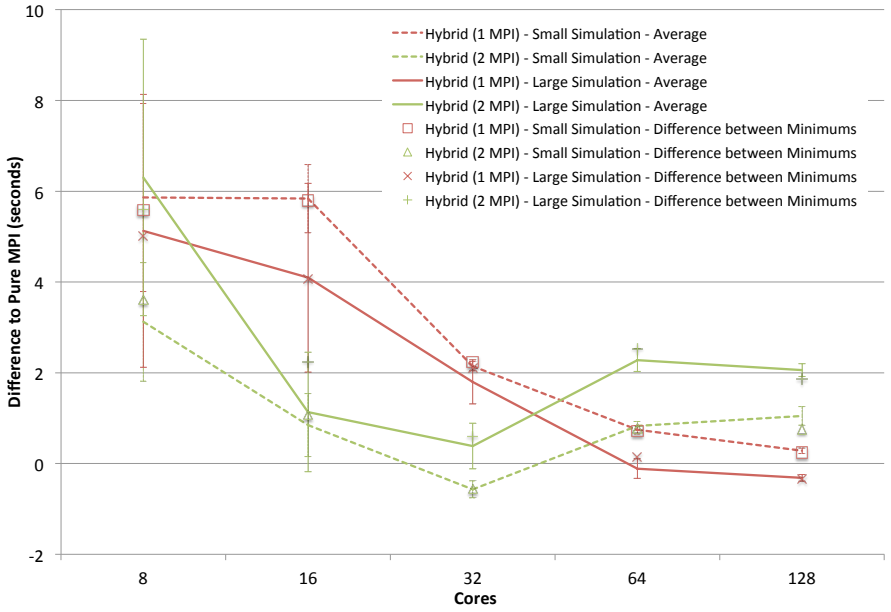


Figure 5.25: Average metal_ld_compute routine timing for Test 20 & 30 on Stella, presented with minimum timing and error bars of 1 SEM. Overall the pure MPI code performs better than the hybrid codes, except for in two cases. Performance difference shrinks as the number of cores increases.

Here it can be seen that on Merlin, the hybrid message passing + shared memory code outperforms the pure message passing code at high core counts; this routine takes less time in the hybrid code than the pure message passing code. This difference is attributable to the lower collective communication times in the hybrid code, rather than any benefit from shared memory threading. This is borne out by the results from Stella, where the situation is different, with the hybrid code only performing better than the pure message passing code at 32 cores for Test 20 with the hybrid (2 MPI) case and at 128 cores for Test 30 with the Hybrid (1 MPI) case. At all other core counts, the pure message passing code performs better.

As with the error analysis of the `two_body_forces` loop, the standard errors to the mean here are very large at small core counts, but decrease as the core counts increase. Again, the cause of this could be load imbalance between the threads which decreases as the subdomain size decreases, although, as already stated at the beginning of Section 5.3.3 further testing would be required to confirm or rule this out.

Summary

This chapter has presented the results of performance testing of a large scale Molecular Dynamics application: DL_Poly. It introduced the code and covered the creation of the hybrid version, before presenting results of a performance analysis carried out on two HPC clusters. The hybrid version was found to have better performance than the pure MPI version at high numbers of processors when using either an Infiniband or a 10 Gigabit Ethernet interconnect. The pure MPI version was again found to deliver better performance at lower processor numbers on both clusters.

The results obtained in this chapter are similar to those in the previous chapter, suggesting that the factors affecting hybrid code performance identified in that chapter also affect the results of the large scale real-world application. However, with this real world code the hybrid message passing + shared memory model performs better than the pure message passing model at high numbers of cores even over a fast low latency interconnect, a result not seen with the previous MD code.

As in the previous chapter the differences in the performance of the hybrid message passing + shared memory code when compared to the pure message passing code are attributed to two main differences between the codes.

Firstly the communication profile of both codes is very different, and in this code collective communication is a significant factor in performance. The hybrid code is able to perform better at collective communication due to the reduced number of MPI processes, and so performs much better than the pure message passing code at higher numbers of processors.

Secondly, the shared memory additions in the hybrid code have introduced overheads not present in the pure message passing code. These overheads are again quite large when running on small numbers of processors, but reduce as the problem size per core decreases.

Standard errors to the mean are relatively small on overall timing results, showing that little variance was observed between overall timing runs. However, larger standard errors are seen

with synchronisation timing and collective communication timing, particularly at lower core counts. This indicates some variability in timing, a possible cause of which is the relative size of subdomains at smaller core counts compared to larger. These larger errors tie in with those seen when observing the runtime of the `metal_ld_compute` routine, which contains some collective communication. Relatively large errors are also seen at small core counts when examining the timing of the main work loop.

The next chapter will describe the factors affecting hybrid code performance, and analyse the performance of the hybrid model as it is applied to the codes in this chapter and Chapter 4.

Chapter 6

Hybrid Model Performance

Overview

This chapter discusses the results obtained in the previous chapters and highlights the most important factors influencing the performance of the hybrid shared memory + message passing model as it compares to the pure message passing model. It then analytically describes each of these factors so that performance of the two types of code may be understood. Breaking down the performance of the hybrid code into separate factors allows the differences between the pure message passing and hybrid message passing + shared memory codes to be understood before an analytic description is made of the overall performance. The results obtained in the previous chapters are used to characterise the factors and inform the analytical models, which are parameterised and compared to the observed code performance. The models presented are intended to act as a guide to allow understanding of where a hybrid message passing + shared memory version of a parallel code could offer performance improvements over a pure message passing version. These models are then used to draw conclusions about the general applicability of the hybrid message passing + shared memory model as it applies to other types of parallel code.

6.1 Introduction

The previous chapters have demonstrated the performance differences between hybrid message passing + shared memory codes when compared to pure MPI, and have allowed these differences to be characterised into separate factors which each affect the performance of such codes. These factors may be classified as:

- **Communication Profile.** The communication profile of hybrid message passing + shared memory codes differs from that of pure MPI codes in phases of both point-to-point and collective communication.
  - **Point-to-Point Communication.** Hybrid message passing + shared memory codes using a domain decomposition parallel model have fewer messages during communication phases than pure MPI codes due to the reduced numbers of MPI processes. However, the payload of (or data transferred in) these messages may be significantly larger in the hybrid codes due to larger halo sizes in each MPI process. Overall the amount of data transferred at each communication step varies between the two classes of code, with hybrid codes having to communicate less data between all processes than the pure message passing codes.
  - **Collective Communication.** Pure MPI codes contain significantly more collective communication than hybrid message passing + shared memory codes. Greater numbers of MPI processes take longer to perform collective communication. Depending on the type of collective communication (and the data being collectively reduced or broadcast) a similar effect may be seen as in point-to-point communication, where the hybrid code contains few large messages, while the pure MPI code contains a higher number of smaller messages.
- **Threading Overheads.** Overheads are introduced into a hybrid message passing + shared memory code that do not exist in a pure message passing code. These may be broken down as follows:

- **OpenMP Overheads.** Shared memory threading parallelism as implemented through OpenMP introduces overheads relating to the forking and joining of threads. The reduction operations carried out at the end of work loops add an additional synchronisation overhead. Additional overheads are added due to the need to create private copies of data for each thread at the beginning of a parallel section; these overheads vary as the size of data needed to be copied varies, with the overheads reducing as the size of subdomain per thread decreases.
- **Serial Code.** As the OpenMP parallelisation occurs within specific points of the existing parallel MPI regions, portions of the hybrid shared memory + message passing code will be run in serial that would run in parallel in the pure MPI code. This introduces an extra overhead to the hybrid code.

Analytic description of these factors allows the creation of models describing these features. Together they allow the performance of both hybrid message passing + shared memory and pure message passing models to be understood and predicted. The main aim of these models is to describe the different performance behaviours of the classes of code (pure message passing versus hybrid message passing + shared memory), so showing where a particular class of code delivers best performance. Given their simplistic nature and the comparative complexity of the parallel software and hardware being modelled it is not expected that the models will accurately predict the actual time observed, more that they will describe the general behaviour of the codes.

Throughout this chapter, although reference will be made to the performance of the first simple MD code, the focus will be the performance of the large scale real-world application DL_Poly, as understanding of this performance will be of more use in general than the understanding and prediction of performance of the simpler example code. Also, the performance profiling of the simple MD code has not been performed to as detailed a level as the profiling of the DL_Poly code, so it is not possible to separate the point-to-point communication from the total communication for instance, as profiling was only done to the level of individual routines in the code, not calls to the MPI library, or the timing of individual loops. All three test cases are used

to evaluate the performance model, although to avoid repetition not all test cases are reported in detail. Instead, test cases are chosen to show the performance model as it compares across the range of tests.

The actual timing results for DL_Poly will be used to assess the performance models created in this chapter. However, the models will aim to be generic enough to describe the performance of any code with a similar parallel data decomposition and overall parallel structure. The applicability of models and assumptions to general parallel codes will be discussed throughout the chapter.

6.2 Communication Profile

The communication profile of both the small MD code and the real-world DL_Poly application have been shown in the previous chapters to vary in a similar way between the pure MPI and hybrid codes. This section will attempt to quantify those differences, by examining the point-to-point and collective communication separately.

The models presented in this section are able to be used as performance predictors for a hybrid code. By estimating parameters for the model using the actual performance results of a pure MPI version of a parallel code, it is possible to estimate the performance of a hybrid version of that code in order to capture the general performance trend. This allows a prediction to be made as to whether a hybrid message passing + shared memory version of a code outperforms a pure message passing version. Parameter estimation is carried out by fitting the performance model curve to the actual observed pure message passing performance data. The model can then be assessed by examining its prediction of the hybrid code performance, separating the data used to tune the model from the data used to assess it. The models are not intended to accurately predict timing performance to the second, although in some cases they can be accurate to within a few percent of the actual runtime. This is assessed throughout the section.

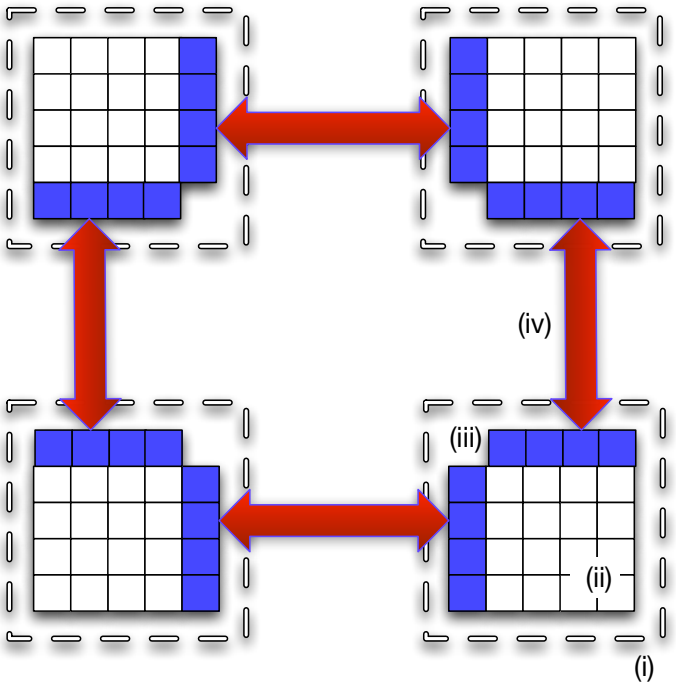
6.2.1 Point-to-Point Communication

Although point-to-point communication is not as significant a factor in the performance of DL_Poly as collective communication is, it is still necessary to examine the point-to-point communication patterns in order to understand how the hybrid model affects the performance of point-to-point communication in general parallel applications

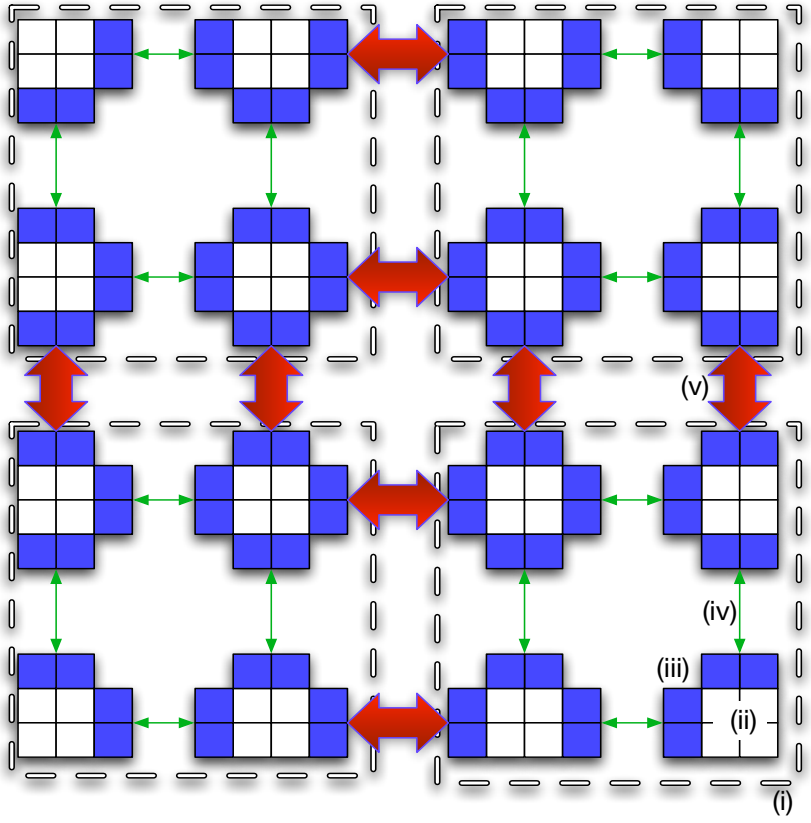
Both codes examined in this thesis use three-dimensional domain decompositions to achieve data parallelism. As each code is originally a pure message passing code, this results in a need for communication at each time step of a simulation to exchange halo data between processes.

An important result of the hybridisation of both codes is that the subdomains and halo sizes change significantly, resulting in a corresponding change in the amount of data communicated by each process, and therefore a change in the communication times of both codes. Figure 6.1 shows this for a simplified two dimensional example, where a data domain is decomposed over 4 nodes, each containing one quad-core processor. Figure 6.1a shows a data domain decomposed over the four nodes in a hybrid fashion, with 1 MPI process per node, and shared memory threading used within the nodes. Each node (i) has a portion of the domain (ii), which may be divided over the four cores in some fashion using shared memory. Each node requires halo information from its neighbours (iii) which will need to be updated at each time step using explicit inter-node communication (iv). Figure 6.1b shows this domain decomposed in a pure message passing code, where an MPI process is started for each core. Each node (i) has a portion of the domain, which is decomposed further so that each core has a smaller sub domain than in the hybrid case (ii). Each process again has halo data for its neighbours (iii), which will need to be updated at each time step via explicit communication; either through intra-node communication (iv) or inter-node communication (v).

It is clear that even in this simplified 4-node (4 cores per node) two-dimensional example that the pure MPI version requires more halo data to be stored for each node, and that this in turn requires a higher number of messages per time step to keep it up to date. However, each of these messages is smaller than in the hybrid case, where although there are fewer messages,



(a) Hybrid Code Decomposition & Communication.



(b) MPI Code Decomposition & Communication.

Figure 6.1: Differences between halo sizes and communication for hybrid message passing + shared memory and pure MPI codes.

Message Direction	Message Size (number of cells)
Up - p_{up}	$(sd_h \times sd_d) \times h_d$
Down - p_{down}	$(sd_h \times sd_d) \times h_d$
Left - p_{left}	$(sd_w \times sd_h + (2 \times h_d \times sd_h)) \times h_d$
Right - p_{right}	$(sd_w \times sd_h + (2 \times h_d \times sd_h)) \times h_d$
North - p_{north}	$((sd_w + 2 \times h_d) \times (sd_d + 2 \times h_d)) \times h_d$
South - p_{south}	$((sd_w + 2 \times h_d) \times (sd_d + 2 \times h_d)) \times h_d$

Table 6.1: Message sizes in each direction for a three-dimensional decomposition of a domain of cells.

they are larger in size. This general pattern will apply to any parallel code written using a message passing style to implement a data decomposition model that is subsequently hybridised by adding shared memory constructs on top of the message passing implementation.

Domain Size, Halo Size and Message Size

There is a clear relationship between the size of the problem domain, the number of processors it is decomposed over, the halo sizes of those domains, and the resulting message sizes for point-to-point communication of halo updates between processes. As already introduced in Section 2.1.3 in a three-dimensional decomposition halo data is kept up to date with six distinct communications between each process, one for each direction of neighbouring processes (up/down/left/right/north/south).

The data transferred in each message is easily calculable. Given a subdomain of width sd_w, height of sd_h and depth of sd_d, and a halo depth of h_d, the messages will each have to transfer the number of cells, $p_{direction}$ of the subdomain given in Table 6.1.

This assumption of data transfer sizes should hold true for any message passing code using a three dimensional data decomposition. Given these numbers of cells that must be communicated, and assuming that particles are evenly distributed throughout the domain of the simulation in both the small scale MD code and DL_Poly, it is possible to work out how the communication profile changes between the pure message passing and hybrid message passing + shared memory codes, in terms of the numbers of particles that are communicated at each

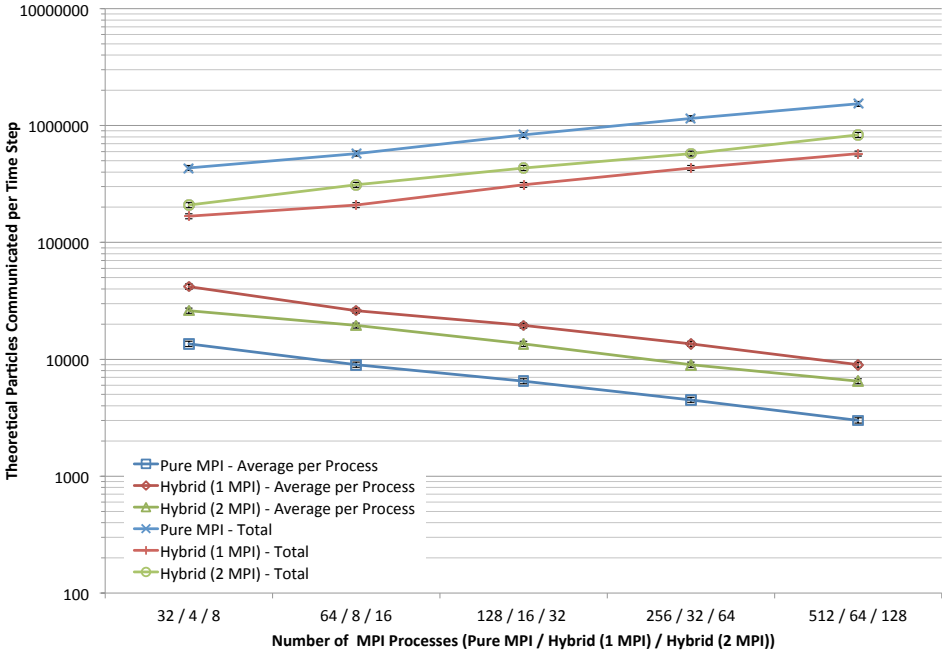


Figure 6.2: Predicted number of particles communicated per time step for DL_Poly Test 20 on 4-64 nodes of a two-socket quad-core cluster. 5% error bars shown.

time step, and therefore the message sizes.

The calculated theoretical number of particles to be communicated at each time step based on the domain decomposition at each process count and number of particles during a run of DL_Poly Test 20 is shown in Figure 6.2. Comparing this to the actual recorded message sizes from the ITAC profiling of the code in Figure 6.3 shows that the theoretical message sizes match the pattern of actual communication in terms of the relative differences between the three classes of code.

This analysis could be used to calculate the theoretical communication time for both pure message passing and hybrid message passing + shared memory applications. Each process, at each time step, needs to communicate a total number of particles, p_{total}:

$$p_{total} = p_{north} + p_{south} + p_{east} + p_{west} + p_{up} + p_{down} \tag{6.1}$$

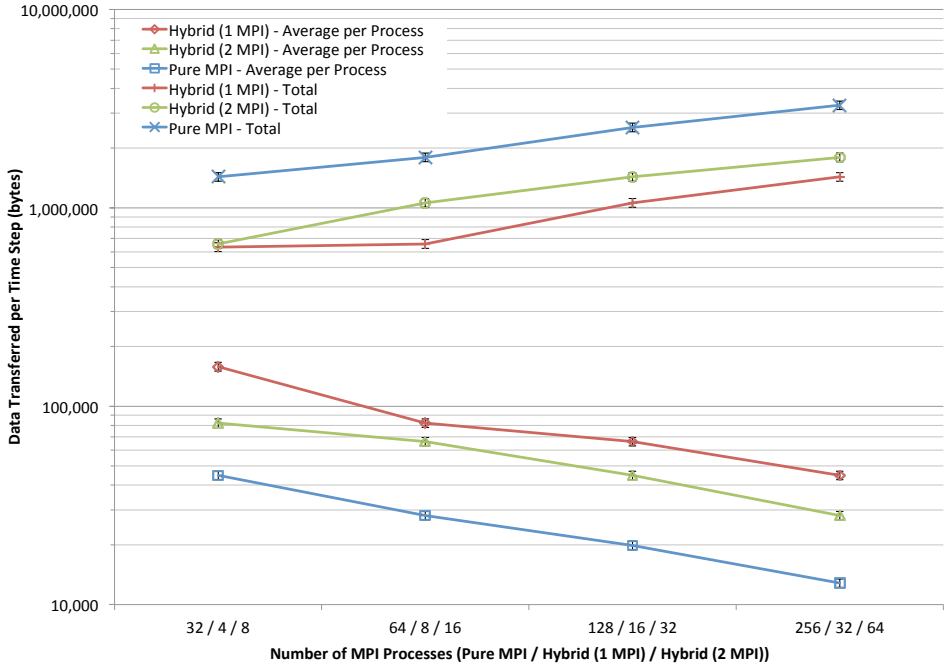


Figure 6.3: Actual data communicated per time step for DL_Poly Test 20 on 4-32 nodes of a two-socket quad-core cluster. 5% error bars shown.

By multiplying this number of particles by the size of their representation in memory, it is possible to calculate the total amount of $data$ to be communicated at each time step. Using the common Hockney model of communication (see Section 2.4), the point-to-point communication time $t_{p2pcomm}$ per process per time step may therefore be written as:

$$t_{p2pcomm} = \alpha + data\ \beta \tag{6.2}$$

However, for calculating the actual communication time this is overly simplistic for a number of reasons. Firstly, the data is transmitted in 6 messages, so there are 6 contributions to message overhead, not just one. Also, while this model takes into account the differing message sizes between pure message passing and hybrid message passing + shared memory codes (through the calculation of the total data size based on the decomposition of data) it does not take into account the differing numbers of processes between the two models. In the pure MPI case, the bandwidth available must be reduced by the number of processes, as all MPI processes within

the node will (depending on the messages being inter-node or intra-node) be communicating over the same network link. In the hybrid (1 MPI) case, the MPI process will be able to utilise the link exclusively, while in the hybrid (2 MPI) case, the link is shared between the two MPI processes on each node. Similarly the amount of overhead must be increased, as rather than 6 messages to be sent per node, there are 6 for each MPI process running on the node. In this way, the communication time is viewed as a 'per node' calculation, rather than 'per process', allowing the difference between the hybrid and pure message passing codes to be captured effectively.

This simple model also fails to take into account process placement, and intra/inter-node communication. For the hybrid (1 MPI) case, all communication will be inter-node, as there is only one MPI process per node. For the pure MPI case, some of the communication will be intra-node, and will therefore take far less time than the inter-node communication.

The amount of inter-node vs intra-node communication obviously depends on process placement. At best, a process will be located on the same node as all six of its neighbours, and so will not use inter-node communication for the updating of halo data. In the worst case, a process will be located on a separate node to all of its neighbours, and will have to use inter-node communication for all updates. Without being able to control the location of subdomains within and across nodes, it is difficult to predict exactly the proportion of inter/intra node communication.

In actual fact, it is easier to calculate the point-to-point communication time using an approximation of the amount of data to be transferred. If the number of particles in a simulation is P, the total number of MPI processes is N_{mpi}, the number of MPI processes per node is N_{ppn}, the time taken to communicate data is β, the message overhead is α and the proportion of inter-node communication is K, the point-to-point communication time can be modelled as:

$$t_{p2pcomm} = \alpha \sqrt[3]{N_{mpi}} + \left(\frac{P}{\log N_{mpi}} K \right) \beta N_{ppn} \tag{6.3}$$

The first term of the equation, $\alpha \sqrt[3]{N_{mpi}}$ adds an overhead to the communication time that increases as the number of MPI processes increases. The amount of data to be transferred in the brackets is not a literal representation of the amount in bytes to be transferred per process, as in reality this is far smaller than $\frac{P}{logN_{mpi}}$. However, this represents the relative difference between the size of subdomains in the pure message passing and both hybrid message passing + shared memory cases, as $\log N_{mpi}$ is obviously larger in the pure MPI case than either the hybrid (1 MPI) or hybrid (2 MPI) case. This factor captures the non-linear relationship between the number of MPI processes and the size of the halo data to be communicated; there is a larger difference between the halo sizes at small core counts than at larger core counts. The factor for inter-node communication K allows for the fact that in the pure message passing code and the hybrid (2 MPI) case some of the data transfer will be intra-node, and will take far less time than an inter-node transfer. Finally, the data to be transferred is multiplied by the time to transfer the data, β, and the number of MPI processes per node, N_{ppn}, representing the fact that in the pure message passing case and the hybrid (2 MPI) case, the bandwidth available per node is shared between more than one MPI process per node. This model is relatively simplistic, and allows for comparison of performance of the pure message passing model with the hybrid message passing + shared memory model without needing to perform a lot of analysis to discover actual bandwidth figures and typical overheads for a system. Estimating the model parameters to give approximately the correct performance of either of the pure MPI, hybrid (1 MPI) or hybrid (2 MPI) cases will give the an approximation of the performance of the other cases.

Taking the model in Equation 6.3 and using the actual performance of the pure MPI version of DL_Poly allows parameter estimation to be carried out using simple curve-fitting software. Throughout this chapter the `FindFit` function in Mathematica v8.0 [118] is used for curve fitting and parameter estimation. Assuming values of K to be 1 in the hybrid (1 MPI) case (where all communication is inter-node), $\frac{5}{6}$ in the hybrid (2 MPI) case (where most communication is inter-node) and $\frac{1}{2}$ in the pure MPI case (where half the communication will be intra-node and half will be inter-node; the average of the best and worst case) allows estimation of α and β and assessment of the performance model prediction to the actual hybrid results.

	Estimate	Error	Percentage Error
	Test 10		
α	0.000613357	0.000145171	23.67%
β	3.76131×10^{-10}	2.19523×10^{-10}	58.36%
	Test 20		
α	0.00040936	0.000293669	71.74%
β	1.72115×10^{-9}	5.94994×10^{-10}	34.57%
	Test 30		
α	0.000297444	0.000242125	81.40%
β	1.44818×10^{-9}	5.02336×10^{-10}	34.69%

Table 6.2: Parameter estimates for Point-to-Point performance model on Merlin, with errors of estimates, and the percentage error these errors represent.

Performing parameter estimation with the pure MPI results of DL_Poly on Merlin gives values for the parameters as in Table 6.2

Using these values in the performance model given in Equation 6.3 gives the performance estimates for Test 20 as seen in Figure 6.4, shown with maximum and minimum estimates based on the parameter error figures. Comparing this to the actual point-to-point communication times in Figure 6.5 shows that the overall pattern of performance, with the hybrid (1 MPI) code performing better than the pure MPI, is shown by the model, along with the general decreasing trend in timing for all three codes up to around 128 cores. After this point the model captures the upwards trend of the pure MPI code, while also suggesting the times of the hybrid codes will also increase slightly, whereas they actually stay the same or slightly decrease. While the model lacks accuracy at lower number of cores, at the higher ranges the values estimated are well within the error bounds of the performance model. At 512 cores, the model predicts the runtime of the hybrid (1 MPI) code to within 4.4% of the actual runtime, and the hybrid (2 MPI) code is predicted to within 0.674% of the actual runtime. Arguably it is more important to be able to predict the performance at higher core ranges, as high core counts (where the runtime is lowest) are where the code is more likely to be run in production use. Similarly, although the model over estimates the communication time at lower core counts this is not too significant a

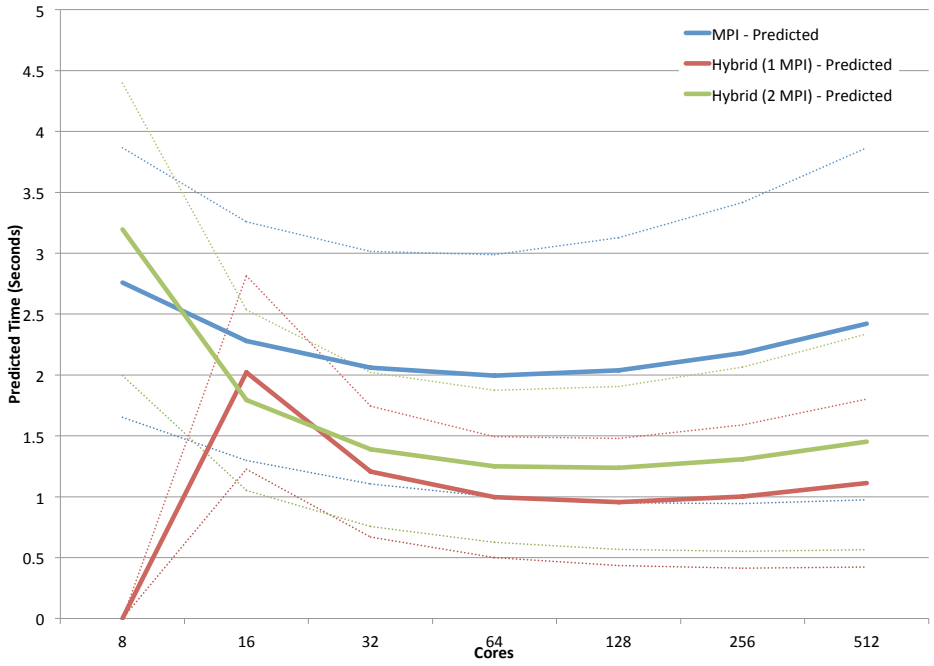


Figure 6.4: Predicted point-to-point communication time for DL_Poly Test 20 on 1 to 64 nodes of a dual-socket quad-core cluster, shown with max and min estimates based on parameter error.

problem as it has already been shown that at low core counts the point-to-point communication is not a significant part of the runtime; therefore it is not as important to capture the point-to-point communication time accurately at low core counts as it is at the higher core counts. At high core counts (> 128) the actual timing results (t_m) and associated errors fit comfortably within the minimum (t_p^{min}) and maximum (t_p^{max}) predicted times.

As already stated, the models are intended to show general behavioural trends of the three classes of code rather than accurately estimating time. The percentage errors of the model predictions to the actual minimum recorded times are given in Table 6.3. In this table, core counts where the actual minimum observed runtime (t_m) falls between the minimum (t_p^{min}) and maximum (t_p^{max}) bounds of the predicted runtime (t^p) (so where $t_p^{min} \leq t_m \leq t_p^{max}$) are highlighted in bold. As might be expected the percentage errors are comparatively low for the pure MPI estimates, as the pure MPI performance data has been used to tune the model. For both the hybrid codes the percentage errors are much higher, especially for Test 10, although it is worth noticing that for Test 20 and 30 these percentages decrease significantly at the highest

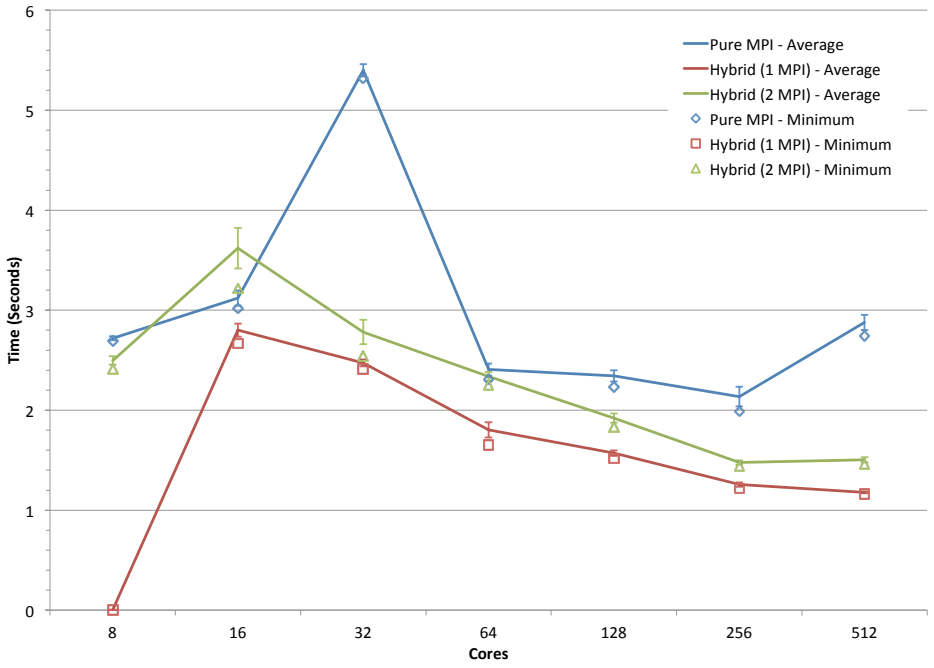


Figure 6.5: Actual point-to-point communication time for DL_Poly Test 20 on 1 to 64 nodes of a dual-socket quad-core cluster.

core counts. As already discussed, these higher core counts are where a more accurate estimate of performance is more useful as these core counts are where a code is likely to be run in production. These higher core counts (particularly for Test 20 and Test 30) are where the actual runtime falls between the maximum and minimum predicted runtimes.

Applicability to General Hybrid Codes

The assumptions forming this model are primarily based around the pattern of changes in the number of messages that must be communicated and the size of those messages between a pure message passing and a hybrid message passing + shared memory code. As already stated, the differences between the communication in these codes should be applicable to any parallel message passing code using a three dimensional data decomposition. For example LAMMPS [96] and GROMACS [14], two other molecular dynamics codes in production use both use three dimensional domain decompositions.

Cores	Pure MPI	Hybrid (1 MPI)	Hybrid (2 MPI)
	Test 10		
8	**6.87%**	6033.57%	**65.64%**
16	**11.87%**	34.61%	41.51%
32	47.93%	**9.16%**	**21.22%**
64	**12.80%**	**11.26%**	**5.09%**
128	34.58%	49.06%	43.66%
256	14.12%	81.75%	70.19%
512	**18.02%**	89.56%	68.10%
	Test 20		
8	**2.54%**	5017%	32.36%
16	**24.49%**	**24.30%**	**44.28%**
32	61.25%	49.97%	45.42%
64	**13.66%**	39.59%	44.41%
128	**8.66%**	36.87%	32.39%
256	9.81%	**17.64%**	**9.21%**
512	**11.62%**	**4.43%**	**0.67%**
	Test 30		
8	**0.63%**	3618.05%	54.64%
16	**20.71%**	**7.73%**	42.63%
32	62.52%	49.22%	44.33%
64	**10.96%**	44.96%	46.08%
128	**15.79%**	41.59%	**35.16%**
256	**0.03%**	**29.09%**	**15.54%**
512	**4.91%**	**14.05%**	**3.27%**

Table 6.3: Percentage errors for Point-to-Point communication performance model estimates to actual minimum observed performance times, Test 10, 20 and 30 on Merlin. Values in bold indicate core counts where the actual runtime falls between the bounds given by the maximum and minimum predicted runtimes.

As long as a reasonable estimate is available for the amount of data to be communicated per process (P in Equation 6.3) and the proportion of inter-node communication (K) can be estimated from the process placement, this model should be able to describe the performance of the point-to-point communication of a hybridised version of the code, provided it uses a master-only style of hybrid implementation.

6.2.2 Collective Communication

The differences between hybrid and pure MPI code for collective communication are more straightforward than for point-to-point communication, as the communication time is largely dependent on the number of MPI processes being used. All three cases of the application, pure message passing, hybrid (1 MPI) and hybrid (2 MPI) will be communicating the same amount of data, so the domain decomposition is not an issue in collective communication, the only significant factor is the number of processes that are taking part in the collective communication. Because of this, the overhead of communication is the most significant part of the collective communication; the actual time taken to transmit data over the network is fairly insignificant in comparison. Using the same naming standards as in the point-to-point model, collective communication may be modelled as:

$$t_{collcomm} = \left(\frac{data}{logN_{mpi}} \right) \beta + \alpha \, N_{ppn}^2 \, \log N_{mpi} \tag{6.4}$$

The first term covers the data transfer, which, while not a significant part of the collective communication, will take time, and will again be affected by factors such as network congestion and the sharing of bandwidth between MPI processes. The $logN_{mpi}$ term serves to model this difference between the pure message passing and the hybrid message passing + shared memory codes. The more significant term involves the overhead α. This is first multiplied by the number of MPI processes per node squared to add the additional factor of the significant problem of collectively communicating between multiple MPI processes, then by $logN_{mpi}$ to differentiate between the pure MPI, the hybrid (1 MPI) and hybrid (2 MPI) cases.

Using the performance results for Merlin, estimates for the parameters of the collective communication model are calculated as in Table 6.4, again with parameter estimate errors.

Figure 6.6 shows the predicted collective communication time for Test 10 of DL_Poly on Merlin, while Figure 6.7 shows the actual recorded communication time. The general trend of the hybrid codes performing worse than pure MPI at low core counts then becoming faster than

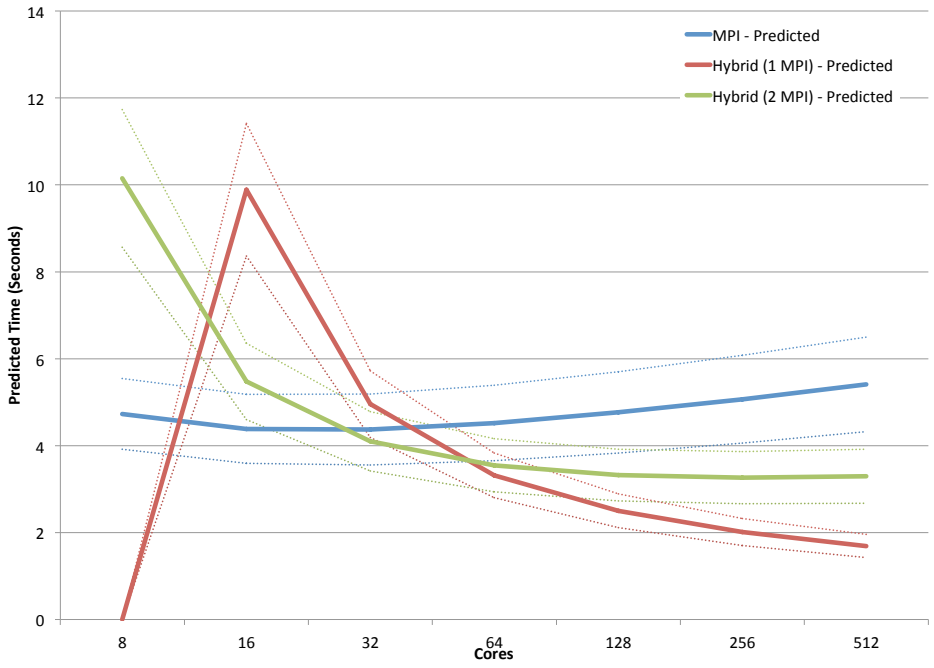


Figure 6.6: Predicted collective communication time for DL_Poly Test 10 on 1 to 64 nodes of a dual-socket quad-core cluster, shown with min and max estimates based on parameter error.

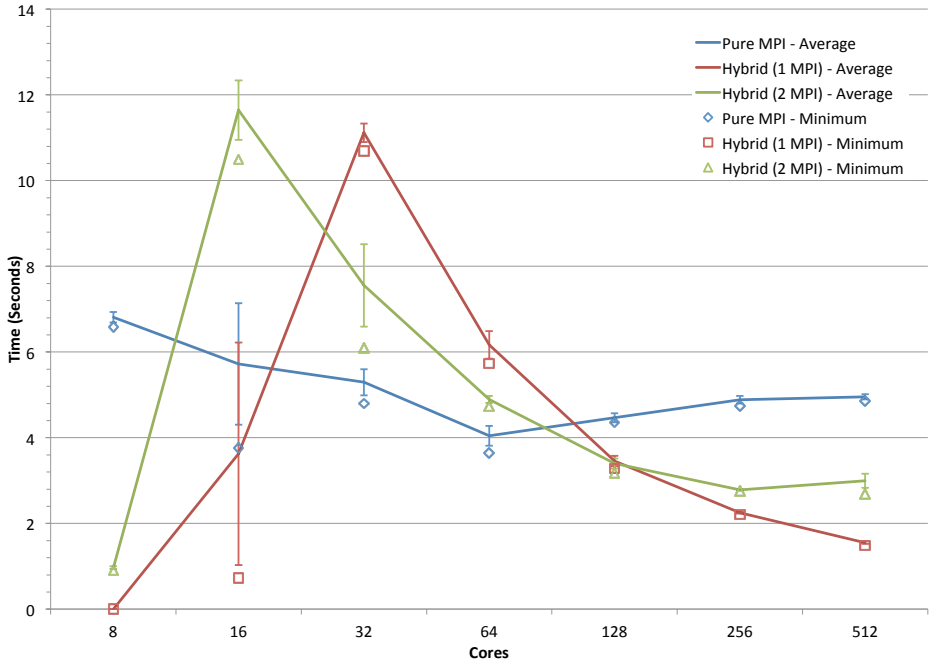


Figure 6.7: Actual collective communication time for DL_Poly Test 10 on 1 to 64 nodes of a dual-socket quad-core cluster.

	Estimate	Error	Percentage Error
	Test 10		
α	3.74358×10^{-5}	7.9885×10^{-6}	21.34%
β	0.000144023	2.22616×10^{-5}	15.46%
	Test 20		
α	7.20633×10^{-5}	1.08607×10^{-5}	15.07%
β	0.000122207	4.05512×10^{-5}	33.18%
	Test 30		
α	6.60724×10^{-5}	1.16731×10^{-5}	17.67%
β	0.000090021	4.46306×10^{-5}	49.58%

Table 6.4: Parameter estimates for collective communication performance model, pure MPI data, Merlin with errors of estimates, and the percentage error these errors represent.

pure MPI at higher core counts is captured. However, the model underestimates how soon this occurs, showing the hybrid codes becoming the better performing at 32 and 64 cores, whereas the actual results show this happening at 128 cores. Also, the large peak in hybrid (1 MPI) code is predicted early, occurring at 16 cores in the model but at 32 cores in the actual results. Again, the model produces smoother results than the recorded communication times, but that is to be expected from a simplistic performance model. Similarly to the previous point-to-point performance model, the time is more accurately predicted by the model at higher core counts, again arguably where accurate prediction is more important. At 256 and 512 cores the actual recorded times fit within the error bounds of the predicted times from the performance model.

Figure 6.8 shows the actual collective communication times against the predicted collective communication times from 128 to 512 cores of Merlin. As can be seen, not only is the general performance trend captured by the performance model, but the timing estimates are within 23% of the actual timing results as well. Actual and predicted times are given in Table 6.5 where the small differences between the two can be seen.

Percentage errors for the performance model estimates compared to the actual observed minimum timings are given in Table 6.6. Again, the model estimates become more accurate at the higher core counts, particularly for the hybrid codes. As collective communication is

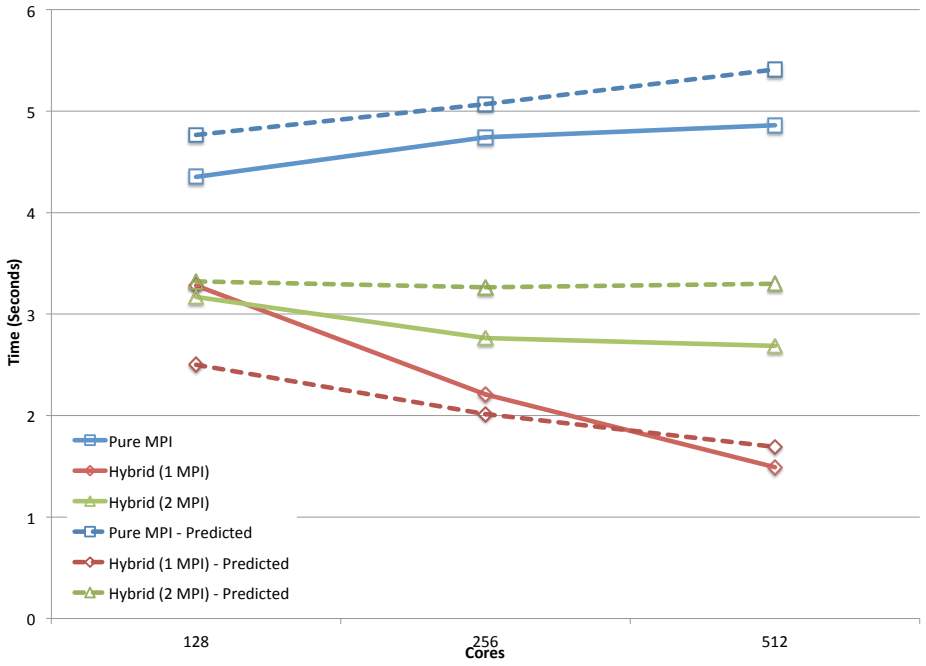


Figure 6.8: Comparison of predicted and actual collective communication time, 128-512 cores of a dual-socket quad-core cluster

	MPI			Hybrid (1 MPI)		
	Predicted	Actual	Difference	Predicted	Actual	Difference
128	4.766	4.351	0.415	2.450	3.283	−0.783
256	5.068	4.743	0.325	2.013	2.209	−0.196
512	5.410	4.861	0.549	1.693	1.491	0.202

Table 6.5: Comparison of predicted and actual collective communication time, 128-512 cores of a dual-socket quad-core cluster

an important contribution to the differences between the pure message passing and hybrid message passing + shared memory codes it is important to be able to model this with some degree of accuracy. It is worth remembering though that the aim of the models is to capture the behaviour of the codes in comparison to one another, rather than accurately estimating the actual performance time to the second. For Test 10 again at higher core counts ($\geq$ 256 cores) the actual runtime falls between the minimum and maximum predicted time of the Hybrid (1 MPI) code. However, this is not the case for Test 20 and Test 30.

Cores	Pure MPI	Hybrid (1 MPI)	Hybrid (2 MPI)
	Test 10		
8	28.15%		1017.79%
16	**16.82%**	1271.35%	47.79%
32	**8.99%**	53.68%	32.67%
64	24.26%	42.12%	24.97%
128	**9.53%**	23.85%	**4.85%**
256	**6.86%**	**8.85%**	**18.13%**
512	**11.30%**	**13.45%**	**22.75%**
	Test 20		
8	**0.80%**		35.67%
16	**13.47%**	40.94%	45.64%
32	**8.99%**	55.71%	40.41%
64	**62.55%**	56.95%	**6.67%**
128	**42.62%**	38.86%	**9.09%**
256	**13.18%**	42.72%	**6.07%**
512	**37.27%**	26.63%	**6.83%**
	Test 30		
8	**3.45%**		64.38%
16	**54.46%**	66.19%	56.50%
32	**7.33%**	70.18%	45.08%
64	**40.35%**	61.63%	30.27%
128	**3.11%**	55.16%	**14.90%**
256	**30.96%**	**41.33%**	**12.48%**
512	**64.75%**	17.71%	**19.59%**

Table 6.6: Percentage errors for collective communication performance model estimates to actual minimum observed performance times, Test 10, 20 and 30 on Merlin. Collective communication has time of 0 seconds for Hybrid (1 MPI) code at 8 cores (only 1 MPI process is used), so no relative percentage error is calculable. Values in bold indicate core counts where the actual runtime falls between the bounds given by the maximum and minimum predicted runtimes.

Applicability to General Hybrid Codes

Again this model is particularly applicable to other hybrid message passing + shared memory codes, provided they use a master-only style of hybrid implementation. As shown, the main factor in the performance of the collective communication is the number of processes involved in the communication, and this model captures the different performance trends between a pure message passing code and a hybrid message passing + shared memory code by focusing on the

number of MPI processes per node and the total number of MPI processes. It should therefore be able to describe the performance of the collective communication of both a pure message passing code and the hybrid message passing + shared memory code created from it. Hybrid codes that do not use a master-only style will not benefit from the reduced number of MPI processes taking part in the collective communication, and this model will overestimate their performance compared to the pure message passing code.

6.2.3 Total Communication

Given the two approximations of communication time it would be simple to approximate the total communication time for the code as:

$$t_{totalcomm} = t_{p2pcomm} + t_{collcomm} \tag{6.5}$$

However, examining the actual total communication time, an example of which is given in Figure 6.9, it is fairly clear to see that the collective communication is the stronger pattern seen in the total communication, controlling the major differences between the three cases of code, while the point-to-point communication only really serves to add additional time to the total communication. The overall pattern of behaviour for the total communication time resembles the collective communication pattern far more than the point-to-point behaviour.

The model for total communication time can be simplified significantly with this knowledge. In the point-to-point model, the contribution of the overhead to the total time is relatively small compared to the contribution from the time taken to transmit data. In the collective communication model the reverse is true: the contribution from the overhead portion is more significant than the portion relating to the transmitting of data. By combining the largest contribution from both models, the total communication time can be modelled as:

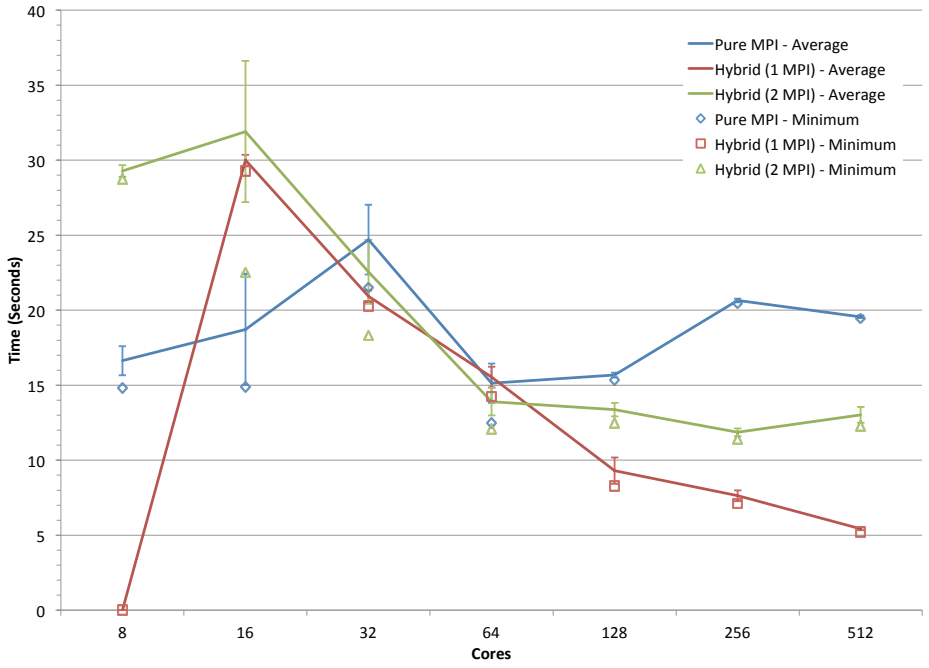


Figure 6.9: Actual total communication time for DL_Poly Test 20 on 1 to 64 nodes of a dual-socket quad-core cluster.

$$t_{comm} = \left(\frac{P}{logN_{mpi}} K \right) \beta + \alpha \, N_{ppn}^2 \, logN_{mpi} \tag{6.6}$$

Parameter estimation using the pure MPI performance timings provides parameter estimates as in Table 6.7. Using these parameter estimates, the total communication performance model accurately describes the general pattern of timing behaviour between the three classes of codes, as seen in Figures 6.10 and 6.11.

Figure 6.10 shows the predicted total communication time for Test 20, which again shows that compared to the actual total communication time in Figure 6.9 the overall pattern of communication differences between the three classes of code is accurately shown by the model. Similarly, the results for Test 30 are shown in Figure 6.11a, which shows the predicted time, and Figure 6.11b which shows the actual recorded communication time. The predicted time is slightly overestimated at lower core counts, but on the whole the model represents the observed behaviour in communication timing: at lower core counts the pure message passing code

	Estimate	Error	Percentage Error
	Test 10		
α	5.00413×10^{-5}	8.71952×10^{-6}	17.42%
β	2.97275×10^{-7}	4.89575×10^{-8}	16.35%
	Test 20		
α	0.0000811498	0.0000147874	18.22%
β	3.42411×10^{-7}	1.10425×10^{-7}	32.25%
	Test 30		
α	0.000072484	0.0000145393	20.06%
β	2.63805×10^{-7}	1.11178×10^{-7}	42.14%

Table 6.7: Parameter estimates for total communication performance model, Tests 10, 20 and 30, pure MPI data, Merlin with errors of estimates, and the percentage error these errors represent.

outperforms the hybrid message passing + shared memory code, while at higher core counts the hybrid message passing + shared memory codes are faster, with hybrid (1 MPI) outperforming hybrid (2 MPI), and both outperforming pure MPI. For the hybrid (2 MPI) code the actual times are within the bounds of the minimum and maximum model prediction at all core counts. For the hybrid (1 MPI) code the actual times are within the bounds for 32 - 256 cores.

Percentage errors of the performance model estimates to the actual recorded times are given in Table 6.8. Unlike with earlier models, the total communication performance model estimates do not improve for the hybrid codes as the number of cores increases. However, as can be seen, for both hybrid models the actual runtimes fall within the maximum and minimum predicted times in the mid-range of core counts (32 - 64 cores for Test 10, 32 - 256 cores for Test 20 and 16 - 256 cores for Test 30 as indicated by bold values in the table). This is the range of core counts where the change over in fastest code type occurs (as evidenced by actual timing results in Figures 6.9 and 6.11b). It is interesting to see that this area of core counts is most accurately predicted by the performance model, as this could prove useful when determining which code type would be best suited to such a range of core counts.

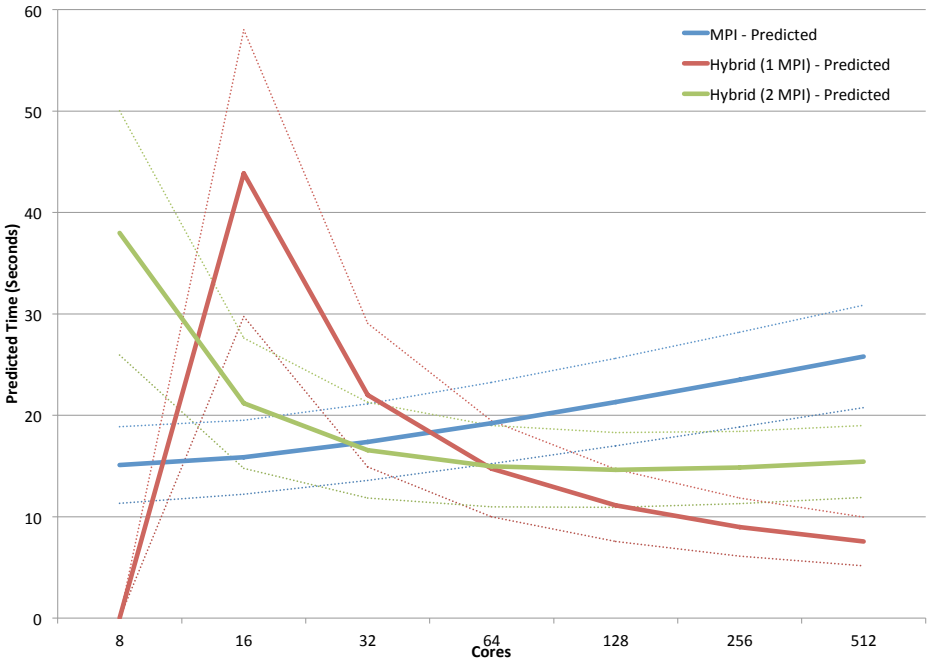
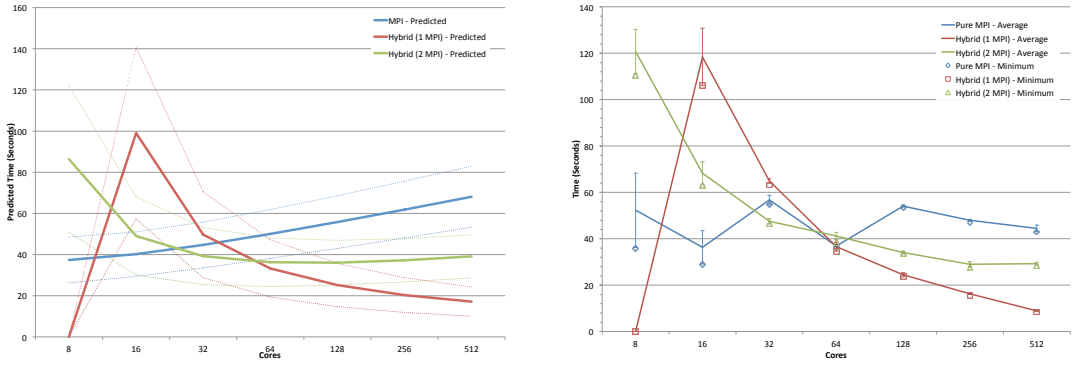


Figure 6.10: Predicted communication time for DL_Poly Test 20 on 1 to 64 nodes of a dual-socket quad-core cluster.



(a) Predicted total comms time, DL_Poly Test 30 (b) Actual total comms time, DL_Poly Test 30

Figure 6.11: Predicted and actual total communication time for DL_Poly Test 30 on 1 to 64 nodes of a dual-socket quad-core cluster.

Applicability to General Hybrid Codes

Unlike the individual models for point-to-point communication and collective communication, the model for total communication is not generally applicable to other hybrid codes, as it

Cores	Pure MPI	Hybrid (1 MPI)	Hybrid (2 MPI)
	Test 10		
8	25.59%	100%	1335.44%
16	**16.30%**	1424.89%	17.72%
32	**11.69%**	**7.52%**	**2.58%**
64	31.50%	**12.11%**	**11.11%**
128	**19.12%**	44.62%	46.86%
256	**14.67%**	64.62%	56.36%
512	**9.99%**	84.28%	56.29%
	Test 20		
8	**2.47%**	100%	**32.14%**
16	**8.31%**	49.87%	**5.67%**
32	19.13%	**8.60%**	**7.55%**
64	54.58%	**5.09%**	**24.05%**
128	40.69%	**36.10%**	**17.19%**
256	**17.16%**	**27.58%**	**31.29%**
512	33.56%	48.12%	**27.24%**
	Test 30		
8	**4.01%**	100%	**21.38%**
16	39.36%	**6.14%**	**21.92%**
32	**18.20%**	**21.10%**	**14.81%**
64	37.33%	**3.10%**	**4.69%**
128	**4.48%**	**7.06%**	**8.05%**
256	31.51%	**29.99%**	**34.56%**
512	59.09%	106.16%	38.67%

Table 6.8: Percentage errors for total communication performance model estimates to actual minimum observed performance times, Test 10, 20 and 30 on Merlin. Values in bold indicate core counts where the actual runtime falls between the bounds given by the maximum and minimum predicted runtimes.

is designed primarily around the total communication performance of DL_Poly, which is dominated by the collective communication performance. Other parallel codes may exhibit different behaviour and be dominated by point-to-point communication or a mix of both types of communication. A model for the total communication of these codes may be to sum the individual point-to-point communication models as in equation 6.5 at the beginning of this chapter. A better approximation may be achieved by weighting the individual models:

$$t_{totalcomm} = \eta\, t_{p2pcomm} + (1 - \eta)\, t_{collcomm} \tag{6.7}$$

where η is the proportion of the total communication attributable to point-to-point communication and $1 - \eta$ is the proportion attributable to collective communication. By observing and combining those aspects of the individual models that are dominant with a particular code, a better model of performance may be achieved.

6.2.4 Communication Timing Performance Models

The performance models described in this section allow for a prediction of the hybrid code performance to be made from the pure MPI code performance. Performing simple curve fitting of the performance models to observed timing data from the pure message passing code allows the parameters controlling performance to be estimated. These estimates can be used with the performance models to generate predicted performance timings for the hybrid message passing + shared memory codes.

In general, these models capture the performance trends of the communication behaviour between the three classes of code. Core counts where pure MPI codes perform better than hybrid codes are shown, and vice versa. In general the models fit the actual data best at higher core counts, where it is more important to have an idea of how well a hybrid version of a code may run, although total time is captured best at mid-range core counts. The models are not intended to act as accurate 'to the second' predictors of performance timing but they are capable in some cases of predicting the performance time. For instance, the point-to-point performance model is able to predict hybrid code timing to within 4.4% for Test 20 at 512 cores and to within 14% for Test 30 at 512 cores, while the collective communication model is able to predict the minimum runtime of the hybrid codes at 512 cores to within 23% for Test 10, 27% for Test 20 and 20% for Test 30

The total communication model is the most useful of the communication models, describing the total difference between pure message passing and hybrid codes. If accurate performance data is held for a message passing version of a code from which parameters can be estimated, it can be used to give an indication of the level of difference in communication performance that

might be seen in any hybrid message passing + shared memory version of that code.

6.3 Computation

This section will attempt to describe the differences between pure message passing and hybrid message passing + shared memory codes in terms of the differences in computation effort between the codes. Again, the code performance will be analysed and described in an effort to produce a model allowing prediction of the runtime.

The performance models in this section typically involve a term relating to shared memory overheads. In order to estimate parameters relating to these overheads, it is necessary to fit the model to the timing data for both pure message passing as well as the hybrid message passing + shared memory codes. The models are therefore not able to be used as prediction tools for estimating the difference between the classes of codes based on the performance of the pure message passing version, but do allow an understanding of the factors affecting the performance.

6.3.1 Direct Overheads

Several extra overheads are present in the hybrid message passing + shared memory codes that are not present in the original message passing code. The spawning of threads at the start of each parallel region (and subsequent joining at the end of each parallel region) takes an amount of runtime. Also, the reduction clauses added to the main work loops of the application add an extra amount of overhead to the code. Further overheads are introduced by the need to create private copies of large data arrays at the start of each parallel section.

The performance of the main work loops can be modelled simply, taking these extra overheads into account. The $work$ to be completed in the loop must be divided between all MPI processes (N_{mpi}). This 'chunk' of work must then be divided further in the hybrid code over the OpenMP

threads spawned from each process (N_{omp}). Multiplying by the time taken to complete a single 'unit' of the work (ϕ) gives a measure of the time taken to complete the loop. Overheads can then be added that are formed as a factor of some overhead value (μ) multiplied by both the number of OpenMP threads (relating to the spawning, forking, synchronising and reduction clauses), and the amount of work to be completed per thread ($work_{omp}$) (relating to the size of the private data copying). This gives a model of:

$$\left(\frac{work}{N_{mpi} \times N_{omp}}\right)\phi + \mu\, N_{omp}\, work_{omp} \tag{6.8}$$

The amount of work to be completed is obviously related to the number of particles in a system, and the flow of control through the main work loop, as different test cases will exercise different functionality within the DL_Poly application.

As the first term deals with the time taken to complete any work in a main work loop, the parameter ϕ can be estimated by fitting this model to the actual data from the pure message passing code. The message passing code has no OpenMP threads, so has no contribution to its performance from the second term of the model. This parameter can then be used along with the recorded data from the hybrid message passing + shared memory code to fit the total model to the recorded data and estimate the parameter μ. Doing so for DL_Poly on Merlin for example provides the parameter estimates as in Table 6.9.

As would be expected as the model has been fitted to recorded data in order to estimate parameters, the resulting predicted performance matches the actual performance pattern. Figure 6.12 shows the predicted execution time for the `two_body_forces` loop in Test 10 of DL_Poly, while Figure 6.13 shows the actual execution time, and the overall performance pattern is matched.

The model is also accurate at describing the differences between the pure message passing and the hybrid message passing + shared memory code. Figure 6.14 shows the predicted difference between the runtime of the `two_body_forces` work loop in the hybrid (1 MPI) and hybrid

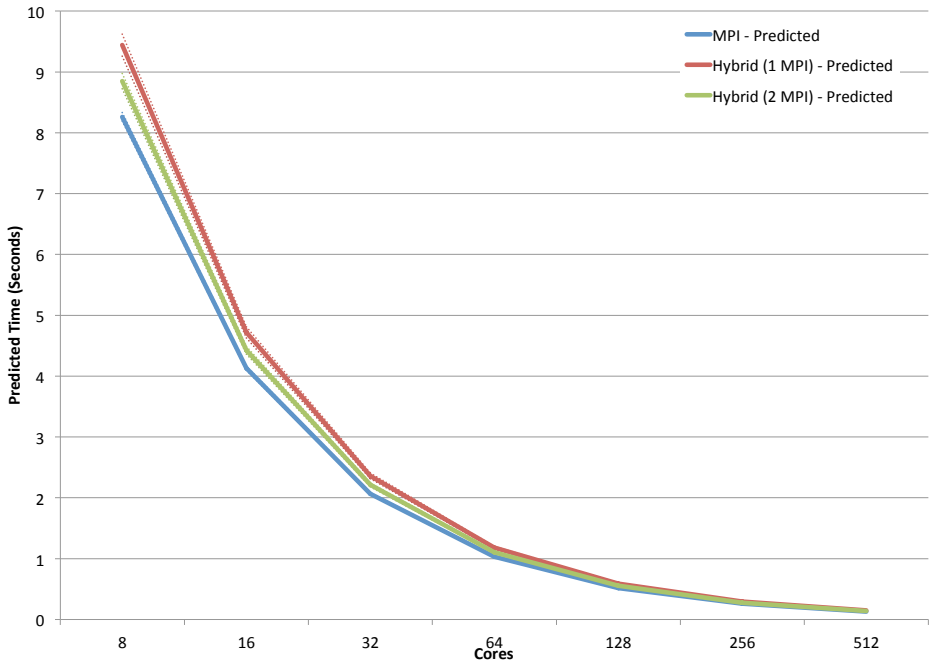


Figure 6.12: Predicted execution time for main work loop of DL_Poly Test 10 on 1 to 64 nodes of a dual-socket quad-core cluster. Maximum and minimum bounds very close to predicted values.

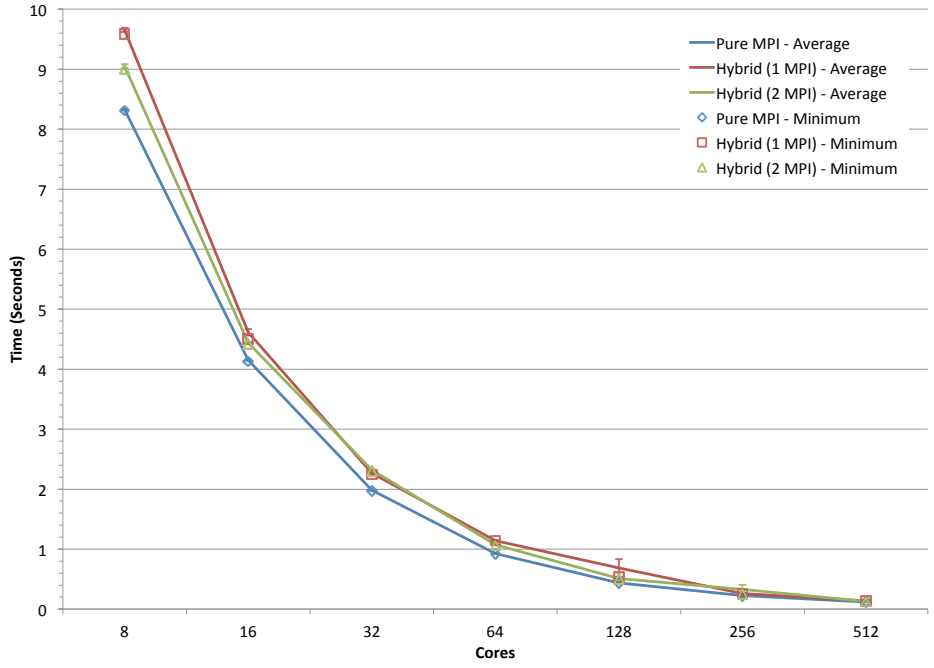


Figure 6.13: Actual execution time for main work loop of DL_Poly Test 10 on 1 to 64 nodes of a dual-socket quad-core cluster.

	Estimate	Error	Percentage Error
	Test 10		
ϕ	9.63482×10^{-7}	8.02585×10^{-9}	0.83%
μ	1.71644×10^{-8}	1.64734×10^{-9}	9.60%
	Test 20		
ϕ	8.64837×10^{-6}	7.34525×10^{-8}	0.85%
μ	9.48473×10^{-8}	2.19222×10^{-8}	23.11%
	Test 30		
ϕ	4.80354×10^{-6}	3.43762×10^{-8}	0.72%
μ	3.95185×10^{-8}	3.13746×10^{-9}	7.94%

Table 6.9: Parameter estimates for work loop overheads, Merlin with errors of estimates, and the percentage error these errors represent.

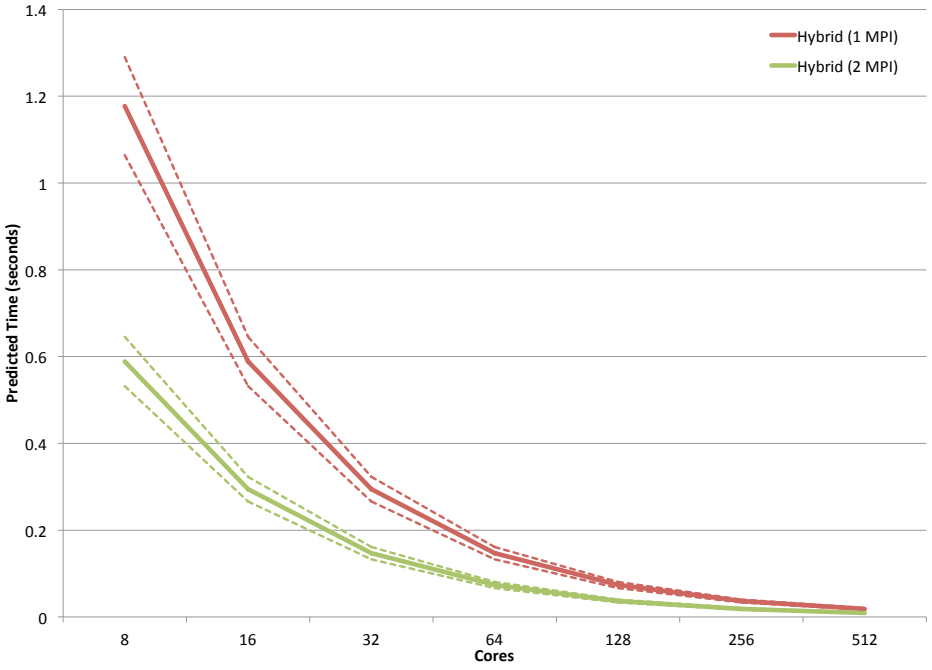


Figure 6.14: Predicted difference in execution time of work loop of DL_Poly Test 10 on 1 to 64 nodes of a dual-socket quad-core cluster.

(2 MPI) cases against the pure MPI case, while Figure 6.15 shows the actual differences. The model predicts that the overheads will be relatively large at low core counts, decreasing as the number of cores increase, which is seen to be the case in the actual results.

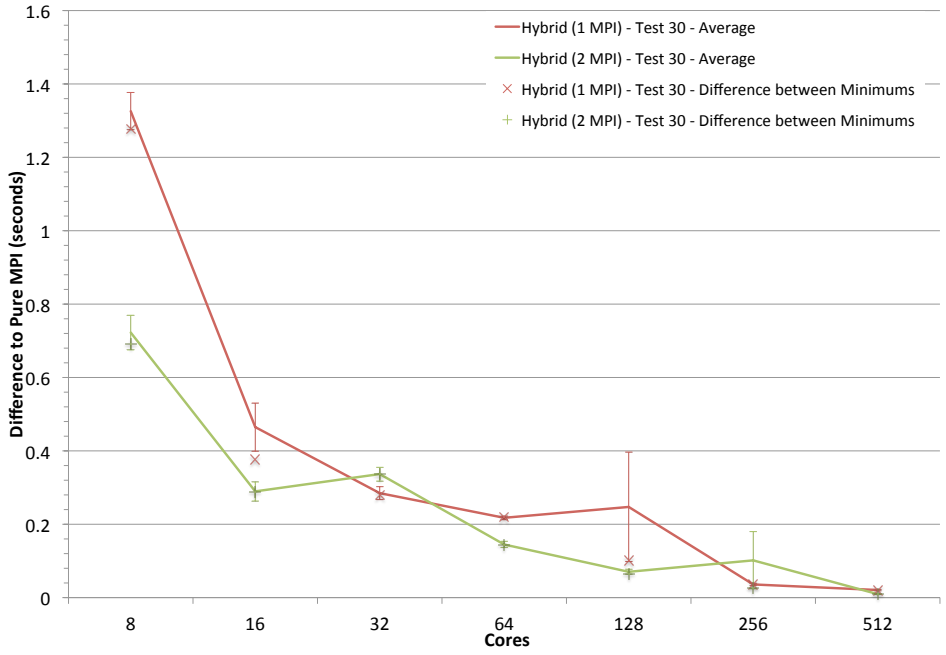


Figure 6.15: Actual difference in execution time of work loop of DL_Poly Test 10 on 1 to 64 nodes of a dual-socket quad-core cluster.

This model can be used for any of the work loops that have been parallelised using OpenMP in the hybrid message passing + shared memory code.

As this model requires curve fitting to both the pure message passing and the hybrid message passing + shared memory code, its usefulness as a predictor of performance is limited, as accurate OpenMP overhead data is needed to estimate the parameters, so at least an OpenMP version of a work loop must exist (if not a full hybrid code). However, it does show that the performance differences between the three classes of code can be modelled in a relatively straightforward manner.

Percentage errors for all three tests estimates when compared to the minimum observed times are given in Table 6.10. As could be expected as both pure MPI and hybrid timing data has been used to fit parameters for the model, the percentage errors are comparatively small, especially when compared to some of the earlier communication models (Section 6.2). Of interest is that the model is more accurate at lower core counts, decreasing in accuracy as the core count

increases. Also of interest is that the actual runtimes observed fall between the minimum and maximum predicted runtimes in only a few cases (as indicated by bold values in the table). This could limit the usefulness of the model, as the higher core counts are where production code is more likely to be run. However, even these larger errors are much smaller than some seen with earlier models, and importantly although the estimates of actual timing are not as accurate, the overall behaviour of the code is modelled well, with the codes starting off at low core counts separated by some distance, then becoming more similar in performance as the core count increases.

Cores	Pure MPI	Hybrid (1 MPI)	Hybrid (2 MPI)
	Test 10		
8	**0.54%**	**1.51%**	1.64%
16	**0.05%**	4.77%	**0.19%**
32	4.85%	4.93%	4.05%
64	12.13%	3.50%	3.98%
128	18.70%	10.07%	10.63%
256	15.26%	13.45%	10.19%
512	10.33%	7.66%	8.89%
	Test 20		
8	0.93%	**2.35%**	2.03%
16	1.67%	5.71%	**0.39%**
32	5.08%	7.66%	**1.20%**
64	6.16%	8.19%	13.98%
128	10.29%	18.57%	16.59%
256	12.25%	23.02%	19.28%
512	13.93%	24.36%	28.33%
	Test 30		
8	0.79%	**0.46%**	1.13%
16	1.52%	0.99%	3.97%
32	3.88%	1.81%	3.69%
64	5.53%	2.35%	6.94%
128	8.46%	8.34%	10.13%
256	13.06%	16.74%	13.25%
512	16.21%	18.06%	19.80%

Table 6.10: Percentage errors for direct overheads performance model estimates to actual minimum observed performance times, Test 10, 20 and 30 on Merlin. Values in bold indicate core counts where the actual runtime falls between the bounds given by the maximum and minimum predicted runtimes.

Applicability to General Hybrid Codes

As long as a measure of overhead for the OpenMP additions to a pure message passing code is known, it should be possible to apply this model to any general hybrid code. The only real difference between hybrid message passing + shared memory codes and the pure message passing code comes in the second term of the model, where an overhead is added that relates to the number of shared memory threads per node and the work that must be done by each thread. This overhead will vary depending on what OpenMP additions have been made to the code. As long as these are understood and can be measured, the model can be applied successfully.

6.3.2 Indirect Overheads

The remaining contribution to the performance of the hybrid message passing and shared memory code is a contribution from indirect overheads. These are primarily a result of portions of the code running with less parallelism in the hybrid message passing + shared memory code than in the pure message passing code. For instance, a routine that has not been parallelised in any way using OpenMP in the hybrid code will be run over 64 cores in the hybrid (1 MPI) case when running on 64 nodes. However, in the pure MPI case this routine will be run over 512 cores, and will therefore theoretically take less time. This should be relatively simple to model, simply being comprised of a factor representing the amount of $work$ to do in these 'serial' sections and the number of MPI processes the work is divided over (N_{mpi}), multiplied by the time taken to complete a unit of this work (ρ). This allows any work not modelled as part of the main work loops in the model above to be captured.

$$t_{indirectoverhead} = \frac{work}{N_{mpi}} \rho \tag{6.9}$$

Applicability to General Hybrid Codes

Again, given an understanding of the architecture of a hybrid application this model could be applied to any general hybrid message passing + shared memory code. While it does not tell us a vast amount about the code performance relative to a pure message passing code, it may assist in creating a model of the overall performance of a code, if the model presented in the following section cannot be applied.

6.4 Overall Performance

A comparison of the overall performance of the pure message passing and hybrid message passing + shared memory codes could be calculated using the models discussed in the preceding sections. The total communications time (or separate point-to-point and collective models) could be added to a contribution for each of the main work loops that are parallelised using OpenMP in the hybrid version, and then an extra contribution for the 'serial' portion of the hybrid code made as in the previous section. However, this is unnecessarily complex.

The performance results of both hybrid codes and the analysis of those results in this chapter have shown that there are two main significant differences between the pure message passing codes and the hybrid message passing + shared memory codes.

Firstly, the hybrid code results include extra overheads from the shared memory additions, which are a large factor at lower numbers of cores, but become less of an issue at higher core counts, as the amount of work shared between threads in each MPI process decreases.

Secondly, the pure message passing code suffers from increasing communication overheads as the number of cores (and hence MPI processes) increases, which is not such a factor in the hybrid code, having far fewer MPI processes per node.

This allows a simple model to be developed describing the total runtime of a hybrid application:

$$t = \left(\frac{work}{N_{mpi} \times N_{omp}} \right) \phi + \mu \, N_{omp} \, work_{omp} + \alpha \, N_{ppn}^2 \, N_{mpi} \tag{6.10}$$

The first term of the model covers the sharing of the $work$ in the application over all the cores being used, which are either running an MPI process or an OpenMP thread spawned from an MPI process. The second term groups all the OpenMP overheads μ, both direct and indirect into a single term, relating these overheads to the number of OpenMP threads (capturing direct and indirect overheads), and also the amount of work per thread, (decreasing the overhead as the number of cores increases). The final term includes an overhead for message passing communication, α, relating this to the number of MPI processes per node and the total number of MPI processes being run.

Again, these parameters can be estimated by using curve fitting to apply this model to the actual observed data. The pure message passing code will have no addition to runtime from OpenMP overheads, so the performance data from this code can be used to estimate the parameters ϕ and α. These parameters can then be used with the model and the recorded performance data of the hybrid message passing + shared memory code to estimate the parameter μ. Doing so for DL_Poly on Stella gives the parameter estimates as in Table 6.11.

This model simply captures the behaviour of the pure message passing and hybrid message passing and shared memory codes. Figure 6.16 shows the predicted runtime for Test 20 on Stella, while the actual runtime is shown in Figure 6.17. Although the overall performance pattern is captured, the total model is not as accurate as some earlier models, overestimating the timing of all three codes at lower core counts, and underestimating the performance of the pure message passing code at high core counts. Again, it is to be expected that the model matches the timing pattern as the parameter estimates from the model have been derived from the actual recorded data, but it is interesting to see that it is possible to model performance in this simple manner.

Performing the same parameter estimation for DL_Poly on Merlin gives the parameter estimates as in Table 6.12. These parameters give the model predicted performance for Test 30 as in

	Estimate	Error	Percentage Error
	Test 10		
ϕ	2.60062×10^{-5}	2.26679×10^{-6}	8.72%
μ	1.62816×10^{-5}	2.80446×10^{-6}	17.22%
α	7.35995×10^{-6}	1.18638×10^{-5}	161.19%
	Test 20		
ϕ	1.77667×10^{-5}	6.68057×10^{-7}	3.76%
μ	7.18838×10^{-7}	3.40214×10^{-7}	47.33%
α	7.70955×10^{-6}	2.6096×10^{-6}	33.85%
	Test 30		
ϕ	1.38261×10^{-5}	6.17472×10^{-7}	4.47%
μ	8.4743×10^{-7}	1.56596×10^{-7}	18.48%
α	1.18638×10^{-6}	2.35547×10^{-6}	24.95%

Table 6.11: Parameter estimates for overall performance, Stella with errors of estimates, and the percentage error these errors represent.

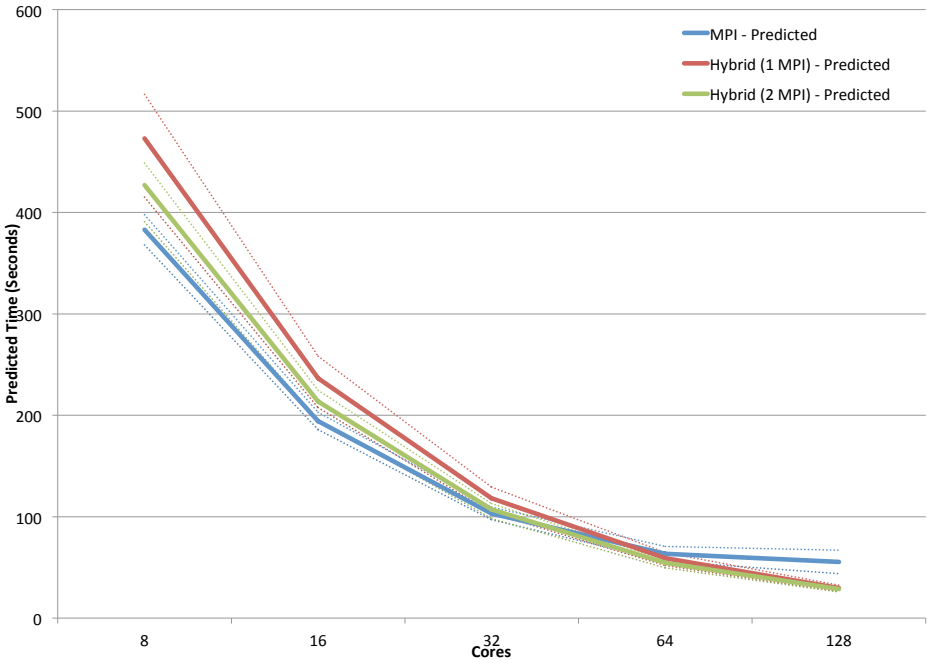


Figure 6.16: Predicted total runtime of DL_Poly Test 20 on 1 to 16 nodes of a dual-socket quad-core cluster.

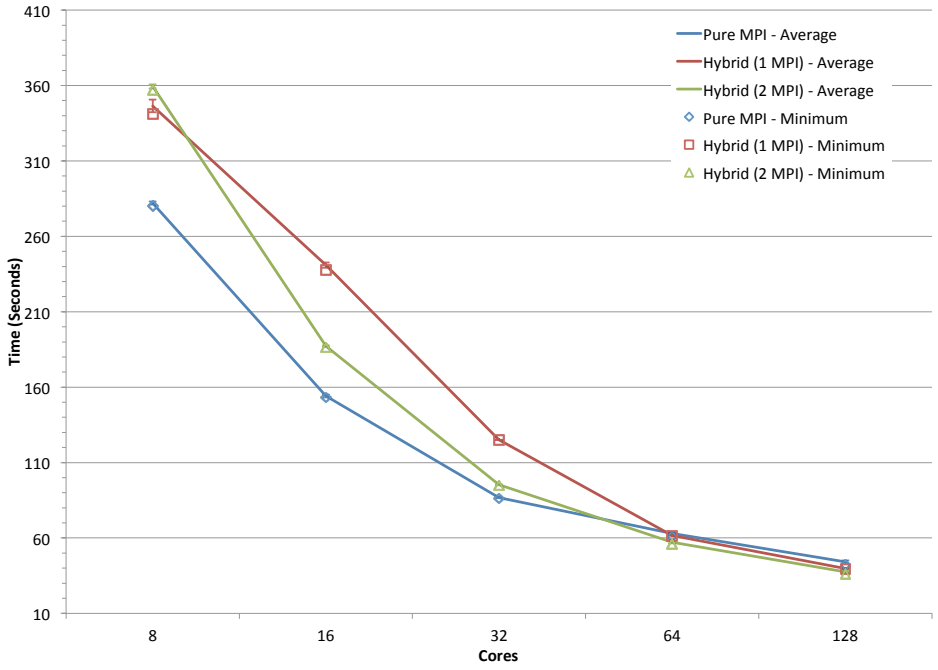


Figure 6.17: Actual total runtime of DL_Poly Test 20 on 1 to 16 nodes of a dual-socket quad-core cluster.

Figure 6.18, which when compared to the actual performance in Figure 6.19 shows that again the model captures the overall trends in performance, showing pure MPI to be faster at low core counts and the hybrid codes performing better at higher core counts, but that it underestimates performance at low core counts and over estimates performance at the higher core counts. The point at which the change in best performance occurs is predicted at 128 cores, matching the actual performance.

Percentage errors for the overall model estimates for all three tests on both clusters are given in Tables 6.13 and 6.14. As can be seen, the percentage errors vary, with Test 20 on Stella having the smaller overall percentage errors on that cluster while on Merlin all three tests actual times are estimated poorly. On both clusters, the number of cases in which the actual minimum runtime falls between the minimum and maximum predicted times is limited.

	Estimate	Error	Percentage Error
	Test 10		
ϕ	2.93210×10^{-5}	7.16911×10^{-7}	2.45%
μ	8.12522×10^{-6}	8.52506×10^{-7}	10.49%
α	1.94296×10^{-6}	9.38035×10^{-7}	48.28%
	Test 20		
ϕ	2.23534×10^{-5}	8.95506×10^{-7}	4.01%
μ	8.02244×10^{-7}	2.66485×10^{-7}	33.22%
α	2.09967×10^{-6}	8.74517×10^{-7}	41.65%
	Test 30		
ϕ	1.35561×10^{-5}	5.53398×10^{-7}	4.08%
μ	6.0127×10^{-7}	1.9887×10^{-7}	33.07%
α	2.62944×10^{-6}	5.27761×10^{-7}	20.07%

Table 6.12: Parameter estimates for overall performance, Merlin with errors of estimates, and the percentage error these errors represent.

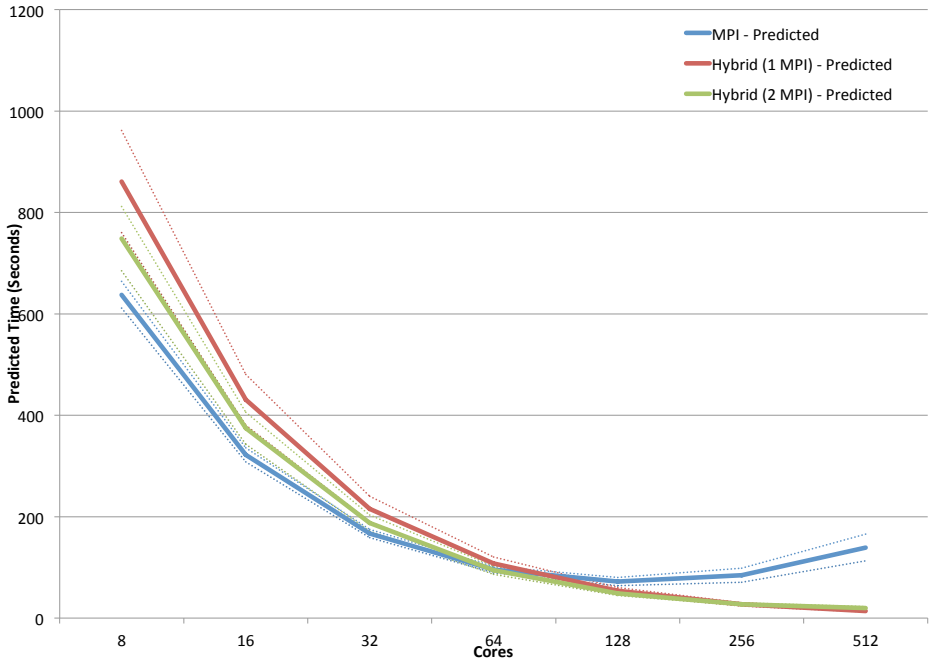


Figure 6.18: Predicted total runtime of DL_Poly Test 30 on 1 to 64 nodes of a dual-socket quad-core cluster.

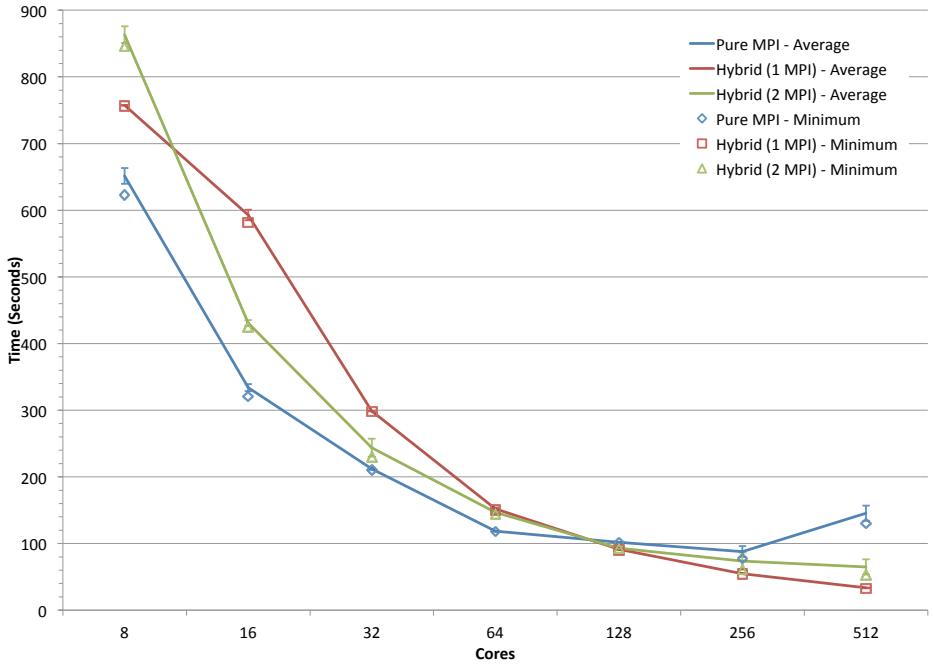


Figure 6.19: Actual total runtime of DL_Poly Test 30 on 1 to 64 nodes of a dual-socket quad-core cluster.

Applicability to General Hybrid Performance

This model should allow the prediction of overall performance of a general hybrid message passing + shared memory code. However, in order to be used it is necessary to understand the overheads created from OpenMP additions (μ) to the code and the overhead related to communication (α), which may be large in the case of a code heavily dependent on collective communication, or smaller in the case of a code which uses more point-to-point communication. A good understanding of the code is therefore necessary, as well as the implementation of OpenMP to be used.

6.5 Applicability to General Hybrid Code Performance

Although the performance models in this chapter have been created with the specific applications discussed in this thesis in mind, there is no compelling reason why they cannot be

Cores	Pure MPI	Hybrid (1 MPI)	Hybrid (2 MPI)
	Test 10		
8	**6.35%**	16.31%	**7.39%**
16	20.81%	43.69%	44.25%
32	21.91%	62.87%	63.67%
64	**13.88%**	73.96%	68.26%
128	**11.22%**	75.48%	64.95%
	Test 20		
8	26.83%	27.85%	16.40%
16	21.23%	**0.50%**	12.82%
32	16.35%	**5.60%**	11.54%
64	**3.63%**	**3.90%**	**2.93%**
128	23.92%	32.72%	26.27%
	Test 30		
8	28.87%	25.32%	21.78%
16	21.82%	**4.28%**	20.39%
32	15.18%	**7.08%**	13.29%
64	**3.97%**	5.35%	21.04%
128	19.68%	46.65%	62.15%

Table 6.13: Percentage errors for overall performance model estimates to actual minimum observed performance times, Test 10, 20 and 30 on Stella. Values in bold indicate core counts where the actual runtime falls between the bounds given by the maximum and minimum predicted runtimes.

used as a guide to explain the performance for other hybrid parallel codes. As discussed in Chapter 2 (and mentioned with examples in Section 6.2.1) the data decomposition model of parallel programming is a common style of writing a message passing parallel application. These applications are characterised by alternating periods of calculation and computation at each time step, as are the applications examined in this thesis. If there is a measure of the amount of work that needs to be done at each time step, and the average amount of data that must be communicated between each MPI process at each time step, the models could be applied to describe the difference between an existing pure MPI message passing code and any hybrid message passing + shared memory version that might be created. This is provided that the hybridisation is to take the form of an application of OpenMP to the main work loops of the application and the hybrid program is to operate in a master-only style with all communication funnelled through the master thread of each MPI process + OpenMP thread group.

Cores	Pure MPI	Hybrid (1 MPI)	Hybrid (2 MPI)
	Test 10		
8	**1.23%**	9.07%	**4.10%**
16	9.62%	34.31%	40.54%
32	6.28%	73.67%	62.50%
64	**1.32%**	88.70%	60.83%
128	16.75%	77.19%	39.00%
256	**16.44%**	47.49%	**2.35%**
512	**13.02%**	**5.40%**	28.94%
	Test 20		
8	**2.98%**	**11.28%**	**2.82%**
16	**2.72%**	21.51%	10.94%
32	19.03%	25.69%	16.79%
64	20.81%	29.94%	27.92%
128	28.65%	43.61%	43.37%
256	22.46%	55.41%	55.66%
512	**34.09%**	64.90%	68.76%
	Test 30		
8	**2.34%**	13.82%	11.48%
16	**0.28%**	25.94%	11.91%
32	20.53%	27.71%	18.38%
64	18.95%	28.29%	34.48%
128	28.96%	40.24%	47.36%
256	**8.89%**	50.04%	55.27%
512	**7.01%**	57.89%	62.38%

Table 6.14: Percentage errors for overall performance model estimates to actual minimum observed performance times, Test 10, 20 and 30 on Merlin. Values in bold indicate core counts where the actual runtime falls between the bounds given by the maximum and minimum predicted runtimes.

As mentioned in their individual sections, the models for point-to-point and collective communication should be applicable to any general hybrid code. These models relate specifically to the number of MPI processes being used to run the code, and are driven by an understanding of the bandwidth and latency of the connection being used in any cluster on which the code is running. As such, the models can be tuned using a pure message passing version of the code and then used to predict the communication performance of a hybrid message passing + shared memory code.

The total communication model is more specific to the performance of DL_Poly, where

it is clear that the collective communication has a larger effect on overall communication performance than the point-to-point communication. However, as already discussed, for other applications the individual point-to-point and collective models could be used to generate a model for the total communication. Again, this could be tuned using an existing message passing version of a code and the performance of a hybrid version could then be predicted.

With the models used to predict the overheads from shared memory additions the applicability is not as straightforward. The model for direct shared memory overheads requires some knowledge of the OpenMP overheads or the performance of an OpenMP version of the work loop being modelled in order to be used to model performance of the hybrid message passing + shared memory code. It may therefore be necessary to prototype these shared memory sections, or to carry out an in-depth profiling of the OpenMP software and hardware ecosystem being used before this model can be applied to any degree of accuracy. A similar situation exists with the overall performance model, as stated above. A possible use for these models could be in applying existing hybrid codes to larger clusters. If an existing piece of hardware was to be expanded, the models could be used to predict the runtime on an enlarged cluster by using the existing performance data to tune the model to the current cluster size, then extrapolating this forwards onto larger cluster counts.

Overall, the communication performance models can be used with an existing pure message passing version of a code to estimate parameters using curve fitting that will give a prediction of how a hybrid message passing + shared memory version of a code will perform in terms of the point-to-point or collective communication.

6.6 Future Hybrid Code Performance

These performance models may be useful in predicting the future performance of hybrid message passing + shared memory codes when compared to pure message passing codes. In order to predict this performance we must make some assumptions about the type and

performance of future generations of HPC hardware. Predicting the performance and types of hardware architecture in future HPC systems is a difficult task [109], but it seems reasonable to make some basic assumptions. It has already been shown that the most important factors in performance when considering hybrid message passing + shared memory codes are the number of cores per node (which affects the ratio between number of MPI processes and the number of OpenMP threads) and the communication performance of the cluster interconnect. It is therefore sensible to focus the assumptions on these hardware characteristics. Assuming that a cluster architecture based around nodes with multiple processors with multiple cores per node remains the dominant architecture for the foreseeable future, the following assumptions seem reasonable and logical given the direction of hardware development over the last few years:

- The number of cores per node is likely to increase as the number of cores per processor increases.
- The bandwidth of network interconnects is likely to increase.
- The latency of network interconnects is likely to drop.

The results examined in this thesis and the performance models in this chapter may suggest that the hybrid message passing + shared memory model would behave in a predictable way on such a future architecture. The increase in the number of cores per node would result in an increase in the number of OpenMP threads per node. This would result in an increase in shared memory overheads (direct or otherwise), suggesting that the main work of the application would take longer on lower processor numbers (with a larger problem grain size) than in the pure message passing code. However, these overheads would reduce as the number of nodes increased (and grain size decreased). The increase in the bandwidth and reduction of the latency of the interconnect should benefit the point-to-point and collective communication of both the pure message passing and hybrid message passing + shared memory codes. However, the large increase in number of MPI processes in the pure message passing version is likely to lead to a much poorer collective communication performance than in the hybrid message passing +

shared memory version, where the number of MPI processes does not actually increase as the number of cores per node increases.

Given the assumptions on the type of future hardware above it is possible to suggest a potential future architecture of cluster that can be used for performance predictions. Taking the Merlin cluster as a basis, an architecture is proposed consisting of nodes, each containing two eight core processors, linked by some interconnect with a higher bandwidth and lower latency than the current interconnect, specifically twice the bandwidth (giving 40 Gb/s) and half the latency (0.9 microseconds). This effectively represents a doubling of the characteristics of the Merlin cluster. This possible hardware configuration can be used to estimate future performance. For brevity, the pure MPI and hybrid (1 MPI) case will be considered, as these are the most accurate in the performance model study above.

Problem sizes will be doubled when considering data transferred or work completed to take the better theoretical performance of the cluster into account, supposing that with a higher performance cluster, larger problems can be considered.

The overall performance model can then be used to predict a possible future performance for both the hybrid message passing + shared memory code and pure message passing code. It is reasonable to assume that the overheads of communication will drop given the faster interconnect. It is also reasonable to assume that the shared memory overheads will remain constant, as faster hardware may reduce the overheads, but the larger number of threads per node (and threads per MPI process) may increase them. The parameters estimated using the current code performance can then be adjusted to predict future performance of the code. Halving the parameters related to communication terms represents an increase in the performance of the cluster interconnect, while the parameters concerning shared memory overheads remain the same as discussed above.

Using the overall performance model, adjusting for a larger problem size, double the number of processor cores per node and the faster interconnect with the lower latency, a possible predicted performance for Test 20 of DL_Poly can be seen in Figure 6.20. As can be seen, it tallies

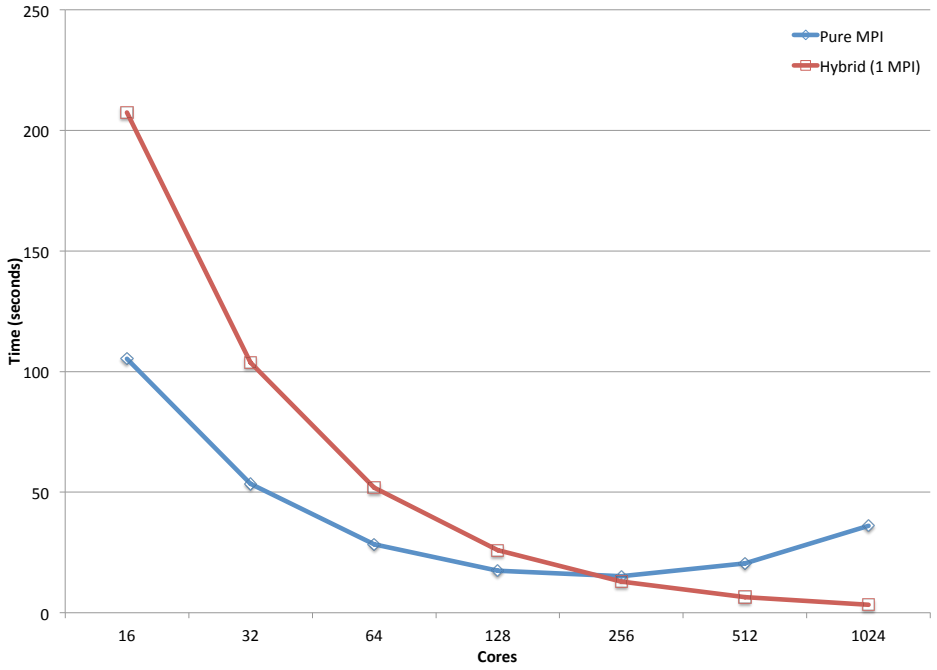


Figure 6.20: Predicted performance on possible future hardware, 16 cores per node, doubled communication performance.

closely with the suggested performance above. The hybrid message passing + shared memory code performs worse than the pure message passing code at lower numbers of cores, while it performs much better at higher numbers, where the pure message passing code scalability is very poor. The difference seen between the two codes is larger at both ends of the scale than the current performance results show.

Of course, the models presented in this chapter are quite simplistic, and only allow tuning of the models in terms of the communication overhead and the shared memory overheads, with the other parts of the models being functions of how the code is run (number of MPI processes in total, number of MPI processes per node, number of OpenMP threads etc.) It is unknown how future hardware architecture changes may affect these overheads. For instance, a change in the architecture of memory access from processor cores could significantly reduce shared memory overheads, while an increase in the speed and use of direct remote memory access in MPI implementations could result in the collective communication performance of pure message passing codes improving greatly. In order to capture these effects fully the models would need

to be much more complicated. In order to predict future performance accurately the future hardware characteristics and performance would need to be known.

Overall, this future performance prediction is speculative, and is unable to be verified on current hardware.

6.7 Performance Model Evaluation

The analysis of the differences between pure message passing and hybrid message passing + shared memory codes has allowed the creation of performance models that aim to describe the performance of these codes. In the case of the communication models, estimating the parameters of the models to match the performance of the pure message passing code allows the difference in performance of the hybrid message passing + shared memory models to be observed. For the other models, parameters must be estimated using both pure message passing performance results and the timing performance of the hybrid message passing + shared memory code. This reduces the usefulness of the models as tools to estimate the potential performance of a hybrid code, but does show that performance can be modelled simply.

Typically the hybrid (2 MPI) case is the most incorrect estimate from the models. That the hybrid (2 MPI) model case provides the most incorrect estimate is of little concern, as this middle case typically provides performance between the other two cases, hybrid (1 MPI) and pure MPI, being neither best or worst at high or low numbers of processors. It is therefore the least useful of the three cases, and may not be used when running code.

The usefulness of the models is not perfect, in order to estimate model parameters some code must have already been run; there is no way to estimate a particular factor of performance (or overall performance) with any reliability using just theoretical statistics on hardware performance. Detailed knowledge of the code is also needed in order to accurately calculate the amount of ‘work’ that must be done at each time step and the amount of data that needs to be transferred in each communication phase.

Each of the separate models has been assessed for its applicability to other hybrid codes. The point-to-point and collective communication models should be applicable to any master-only hybrid code. The total communication model may not be as applicable due to the specific ratio of point-to-point and collective communication within the DL_Poly application, but a total communication model has been described that should be applicable to other hybrid codes. The direct OpenMP overheads model can be applied as long as the overheads caused by the OpenMP implementation used are understood. The overall performance model is quite specific to the applications examined in the thesis, particularly DL_Poly, but the individual models (point-to-point, collective, direct overheads and indirect overheads) can all be combined to get an overall picture of general hybrid code performance.

The performance models presented here are very simple in their design. This allows a general idea of the performance differences between pure message passing and hybrid message passing + shared memory code to be gained relatively easily. However, this may lead to inaccuracies when predicting the performance of other general parallel applications, or the actual precise performance of the applications used to create the models. How well the model parameters allow changes in hardware architecture to be captured is unknown, but significant architectural changes in HPC hardware will almost certainly result in different performance effects on hybrid codes.

Summary

This chapter has analysed the performance of the hybrid model as it applies to the Molecular Dynamics applications used in this thesis. The analysis has allowed the creation of models representing the performance of the codes.

Separate parts of the applications have been modelled, with models created and compared to actual results for point-to-point communication, collective communication, and both direct and indirect shared memory overheads. A general model describing the total runtime has also been

created and the accuracy of the prediction of general performance trends between the pure message passing and hybrid message passing + shared memory codes has been assessed.

The overall performance model has been used to predict a possible future performance of the hybrid message passing + shared memory model as compared to the pure message passing model, finding that the performance gap between the two becomes larger, with the pure message passing model performing poorly as the number of cores per node increases and the number of nodes used increases.

Chapter 7

Conclusion

Overview

This chapter presents the conclusions that have been drawn from the work presented in this thesis. Overall general findings are presented followed by detailed conclusions. Future directions for this work are also discussed.

7.1 Introduction

The hybrid message passing + shared memory model has often been discussed as a possible model for improving code performance on clusters of SMP's or clusters of multi-core nodes. The applications examined in this thesis have shown that performance improvements over a pure message passing code are achievable when using the hybrid message passing + shared memory model. However, these improvements are only possible under certain conditions.

When running code on small numbers of cores on all systems and over all interconnect types used for performance testing the pure message passing codes in this thesis perform better than the hybrid shared memory + message passing codes. At large numbers of cores, the performance is determined by the interconnect and application used.

Two different applications have been performance tested in this thesis. Hybrid versions were created of both by adding shared memory OpenMP parallelism on top of existing message passing MPI codes. A small scale MD code written in C was tested as an initial example, followed by the real world large scale MD application DL_Poly.

Using the small scale MD code, at large numbers of cores, the pure message passing code performs best over a fast, low latency interconnect such as Infiniband. Using a slower, higher latency such as 1 Gigabit Ethernet, the hybrid shared memory + message passing code outperforms the pure message passing code at large numbers of cores

When using the real-world DL_Poly application on large numbers of cores, the hybrid message passing + shared memory code performs better than the pure message passing code even over fast low latency interconnects such as Infiniband or the slightly slower 10 Gigabit Ethernet.

The performance differences between the pure message passing and hybrid message passing + shared memory codes have been found to be attributable to a number of factors. Firstly, the communication pattern between the two types of code is very different. The use of collective communication versus point-to-point communication is also an important factor, as a hybrid message passing + shared memory code is able to perform better than a pure message passing

code when large amounts of collective communication are used. Secondly, the shared memory overheads are an important issue, as the pure message passing code does not have any shared memory overhead, while the hybrid message passing + shared memory code has increased overheads caused directly or indirectly by the shared memory additions.

7.2 Shared Memory Threading

No improvement in performance has been found that can be attributed to parts of the code running faster using shared memory threading. This is sometimes suggested as a possible area that may deliver improvements in a hybrid message passing + shared memory code over a pure message passing code. However, the simple shared memory threading implementation in the applications used in this thesis (utilising loop level parallelism) offers no performance improvements.

The addition of shared memory threading to the pure message passing code introduces extra overheads to the code.

Direct overheads are added as a result of OpenMP parallelisation, caused by the spawning, joining and synchronising of extra threads on top of the message passing processes. Other OpenMP clauses such as `omp reduction` also add overheads that are not present in the pure message passing codes. These overheads increase the runtime of the main work loops of the application to which the OpenMP parallelisation has been applied.

Indirect overheads are caused by effects such as reduced parallelism. In a hybrid message passing + shared memory code the amount of parallelism throughout the whole application is reduced; the application will only use the full resources of the system during those parts of the code which have benefited from OpenMP additions. At other times the code will only run using however many MPI processes are running per node, resulting in a slower runtime and inefficient use of computing power.

Shared memory overheads (whether direct or indirect) have been found to be a significant

addition to runtime when running on lower numbers of cores. At low processor counts the problem grain size per thread is relatively large, thus inefficiencies in the shared memory implementation are a bigger issue than at higher processor numbers where the grain size is smaller. As the number of cores used increases, the problem size per core reduces, and the overheads from shared memory additions reduce. At very high numbers of cores the extra overheads are negligible when compared to the runtime of the pure message passing code.

Cache sharing and memory bandwidth issues were shown to be minimal with the small MD code but this may be due to the specific implementation of the MD algorithm used in this code, and cannot be generalised to the hybrid model itself.

7.3 Communication Profile

Significant performance differences have been found between the pure message passing and hybrid shared memory + message passing codes.

The point-to-point communication phase of a three dimensional domain decomposition code running in a pure message passing style is characterised by many messages of a relatively small size when compared to the same code running in a hybrid message passing + shared memory style, where the point-to-point communication is characterised by a smaller number of larger messages. Additionally, in a pure message passing code a proportion of the point-to-point communication will be intra-node, while the rest will be inter-node. Inter-node bandwidth will therefore be shared by many processes, while in a hybrid message passing + shared memory code running one MPI process per node, all communication will be inter-node, but only one MPI process is used, so will be able to use the full inter-node bandwidth.

When using a fast, low latency interconnect such as Infiniband, the hybrid model shows no improvement over the pure message passing model in terms of point to point communication time in the small scale MD code. However, in the larger scale DL_Poly application, the point to point communication is consistently faster in the hybrid message passing + shared memory

code. The hybrid message passing + shared memory model therefore offers advantages over a pure message passing code in a real world code with a complicated communication pattern, but offers little advantage in a smaller example code.

Collective communication is consistently faster in a hybrid message passing + shared memory code than in a pure message passing code. Reduced numbers of MPI processes in a hybrid message passing + shared memory code when compared to the pure message passing code result in much faster global communication. Other codes and applications that make heavy use of collective communication should therefore see much better performance with a hybrid message passing + shared memory model than with a pure message passing model. These results are dependent on the code being relatively well load balanced.

7.4 Performance Models

Analysis of the performance results from the hybrid message passing + shared memory and pure message passing codes used in this thesis has informed the creation of performance models describing the relative performance of each code. Models have been created describing the point to point communication profiles, the collective communication profiles, the total communication profiles, the additional overheads on work loops from threading overheads, and the general overall performance.

On the whole these models predict the general performance differences between the three cases (pure MPI, hybrid (1 MPI) and hybrid (2 MPI)), showing correctly the points where one case performs better than the others, although the accuracy of timing predictions varies.

The models are individually applicable to other hybrid codes depending on the situation, with the separate communication models and the shared memory overhead models being most applicable, and the overall communication and overall performance models being less applicable to general codes.

The performance models have been used to estimate the performance of the hybrid model

on future generations of HPC hardware. It is suggested, by examining a hypothetical future hardware architecture that the overall pattern of performance seen in this thesis will hold as the number of cores per node and interconnect bandwidth increases while interconnect latency decreases. The hybrid message passing + shared memory code performs much better than pure message passing code at higher processor core numbers, while the reverse is true at lower processor core numbers. This is however speculative and cannot be currently verified.

7.5 Hybrid Code Performance

The hybrid message passing + shared memory codes and performance models presented in this thesis have shown that there are specific situations in which the hybrid model performs much better than the pure message passing model. The combination of factors that result in the hybrid message passing + shared memory model performing better are:

- When running on large numbers of cores. Here the reduced communication overheads from the smaller number of MPI processes in a hybrid message passing + shared memory code make up for the increased shared memory overheads of the OpenMP additions and the performance is improved.

- When running on slower interconnects. When bandwidth is reduced and latency increased, the pure message passing model performs worse due to extra communication overheads. As above these are reduced in the hybrid message passing + shared memory model, so the hybrid model performs better over these interconnects.

- When the application code contains large amounts of collective communication. The reduced numbers of MPI processes in the hybrid message passing + shared memory code are able to perform collective communication faster than the large number of processors when running the pure message passing code. In this case, better performance is seen over a faster low latency interconnect.

7.6 Future Work

This section presents future work to be completed on the hybrid message passing + shared memory model, and describes directions in which the research could be taken in future.

7.6.1 Hybrid Implementations

The implementations of the hybrid message passing + shared memory model used in this thesis were kept intentionally simple, utilising only loop level parallelism in order to examine if this simple approach allowed performance improvements over the pure message passing model.

It has been seen that the threading overheads are not inconsiderable in the work portions of the hybrid code. Work could be done to examine and attempt to reduce these overheads. The general structure of the code and main work loops could be changed to improve data locality for the shared memory threads, hopefully improving performance of the work portions of the code. This would allow the hybrid model to perform better, possibly even performing as well as or better than the pure message passing model on lower numbers of cores.

The hybrid model should also be examined with other types of computational science codes to see how they perform with the model when compared to pure message passing codes. It may be that other codes exhibit different behaviour, particularly those codes with simpler or more complex relationships between the number of processor cores being used and the domain decomposition being used to implement data parallelism. Similarly, it would be interesting to see if the hybrid model can be used with codes that utilise a task parallelism style of parallel programming.

Appendix A

Practical Notes on Running Hybrid Code

The hybrid shared memory + message passing model, although not a new idea itself, is relatively unknown in terms of support and knowledge within the HPC community.

Although many clusters, compilers and MPI implementations state compatibility with the hybrid model, in practice this is not always the case. While running hybrid code on multi-core clusters there are several potential issues that can affect performance or even stop the hybrid code from executing correctly. These are documented in this section.

Most MPI libraries used for testing the applications in this thesis profess support for hybrid MPI + OpenMP applications. However, there were often caveats not mentioned in documentation that meant special measures need to be taken to ensure correct operation of the code.

None of the code in the applications in this thesis uses anything other than a *master only* style of hybrid programming. Only the master thread of each set of threads carries out any message passing communication; all MPI library calls occur within the master thread. This *should* mean that no multi-threading support is required from the MPI implementation used for running the code. However, with some implementations it was found that unless multi-threading support was requested at initialisation time, shared memory threads were not allowed to be spawned within the parallel application, or if they were spawned, they were prevented from running on any cores other than the core the MPI process was running on. This may be due to the default `MPI_Init()` method call acting as `MPI_Init_thread(MPI_THREAD_SINGLE)`, so specifying that only one thread will run. With this in mind, it may be safer to always use `MPI_Init_thread`, even when no specific multi-threading support is required.

Intel MPI would frequently display behaviour where the shared memory threads would be spawned, but not allowed to migrate between cores on a processor, instead all being forced to run on the same core as the master thread/MPI process. Setting the environment variable `I_MPI_PIN_DOMAIN=off` solved this problem, despite documentation stating that this is the default for the environment variable anyway.

Behaviour was also seen at some times, similar to the case with Intel MPI, where it seemed that the application was only using one of the available cores on each of the nodes on which it was running. This was due to the `OMP_NUM_THREADS` environment variable not being propagated to all nodes running MPI processes. Most MPI runtimes (`mpirun/mpiexec`) provide a method to transmit environment variables to all MPI processes. Using this option in this case fixed the problem.

List of Figures

2.1 Domain Decomposition - a domain of cells is divided into sub-domains, each of which is assigned to an individual processor. 14

(a) Data Domain. 14

(b) Decomposed and Allocated. 14

2.2 Distributed Memory and Shared Memory. 16

(a) Distributed Memory - individual processors within nodes communicate through explicit messages over a communication network. 16

(b) Shared Memory - individual processors access the same shared memory space, allowing implicit communication. 16

2.3 Data Required from Neighbour Processes - P5 requires knowledge of the data belonging to the subdomains on the neighbouring processes in order to complete the work for its own subdomain. 24

2.4 Data Parallelism - Halo Data Communication. 26

(a) Up Message 26

(b) Up Message Received. 26

(c) Down Message. 26

(d) Down Message Received. 26

(e) Right Message. 26

(f) Right Message Received. 26

(g) Left Message. 26

(h) Left Message Received. 26

3.1 Merlin - achievable bandwidth using MPI_Sendrecv on a chain of 8 - 512 processes, each process sending data to the neighbour to the left and receiving data from the neighbour to the right. 57

3.2 Merlin - achievable bandwidth on a chain of 8-512 Processes, each process using MPI_Send, MPI_IRecv and MPI_Wait to send data to its neighbour to the left and receive data from its neighbour to the right. 57

3.3 Merlin - message overhead when sending a message of 0 bytes size on 8-512 Processes. As the number of processes increases the overheads increase. 58

3.4 OpenMP PRIVATE clause timing & overhead as array size changes. Both overhead and total time increase as the array size increases, until the largest size tested, where both times decrease. 59

3.5 Example graphical results presentation. Average recorded times shown as a line plotted with error bars of 1 standard error to the mean (1 SEM). The minimum recorded time is also presented as a separate marker. 61

3.6 MPI Processes and Shared Memory Thread combinations for Pure MPI, Hybrid (1 MPI) case and Hybrid (2 MPI) case 66

(a) Pure MPI - 1 MPI process per core, no OpenMP threads. 66

(b) Hybrid (1 MPI) - 1 MPI process per node, all other cores filled with OpenMP threads. 66

(c) Hybrid (2 MPI) - 2 MPI processes per node, all other cores filled with OpenMP threads. 66

4.1 Merlin memory bandwidth tests. Error bars of 5% shown. Very little performance difference is observed between the codes running on a fully populated node (ppn=8) and an under populated node (ppn=4). Overlapping error bars indicate no statistical significance in difference between timings on under populated and fully populated nodes. 76

4.2 Average performance timing for the small test case on Merlin using the Infiniband interconnect, shown with error bars of 1 SEM and minimum observed timing. MPI is consistently faster than both hybrid (1 MPI) and hybrid (2 MPI). 81

4.3 Average performance timing for the large test case on Merlin using the Infiniband interconnect, shown with error bars of 1 SEM and minimum observed timing. MPI is consistently faster than both hybrid (1 MPI) and hybrid (2 MPI). 81

4.4 Average performance timing for the small test case on CSEEM64T using the Infinipath interconnect, shown with error bars of 1 SEM and minimum observed timing. Hybrid (1 MPI) and hybrid (2 MPI) out performed by pure MPI at all core counts. 83

4.5 Average performance timing for the large test case on CSEEM64T using the Infinipath interconnect, shown with error bars of 1 SEM and minimum observed timing. Hybrid (1 MPI) and hybrid (2 MPI) out performed by pure MPI at all core counts. 83

4.6 Average 64 core timing for all tests on CSEEM64T using the Infinipath connection. Pure MPI provides best performance, while hybrid (1 MPI) performs better than hybrid (2 MPI). 84

4.7 Overall average timing, small test on CSEEM64T using GigE connection, presented with error bars of 1 SEM and minimum observed timing. Below 48 cores pure MPI provides best performance, above 48 cores both hybrid (1 MPI) and hybrid (2 MPI) provide better performance than pure MPI. 85

4.8 Overall average timing, medium test on CSEEM64T using GigE connection, presented with error bars of 1 SEM and minimum observed timing. Below 48 cores pure MPI provides best performance, above 48 cores both hybrid (1 MPI) and hybrid (2 MPI) provide better performance than pure MPI. 85

4.9 Overall average timing, large test on CSEEM64T using GigE connection, presented with error bars of 1 SEM and minimum observed timing. Below 48 cores pure MPI provides best performance, above 48 cores both hybrid (1 MPI) and hybrid (2 MPI) provide better performance than pure MPI. 86

4.10 Average 64 core timing for all tests on CSEEM64T using the GigE connection, presented with error bars of 1 SEM. Hybrid (2 MPI) performs better than hybrid (1 MPI), both perform better than pure MPI. 87

4.11 Average amount of data sent per process per time step for the small simulation. 5% error bars shown. As the number of cores increases, the amount of data sent per process per time step reduces by a larger amount in the hybrid code than in the pure MPI code. 90

4.12 Cumulative data sent per time step for the small simulation. 5% error bars shown. The pure MPI code sends more data per time step in total than the hybrid codes. 90

4.13 Movout routine timing on Merlin at 128 cores, presented with error bars of 1 SEM. Hybrid (1 MPI) code consistently slower than pure MPI code. 93

4.14 Movout routine timing on CSEEM64T at 96 Cores. Both hybrid cases slower than pure MPI over Infinipath, but faster than pure MPI over GigE. 94

4.15 Movout routine timing in large simulation on CSEEM64T over both Infinipath and GigE interconnects. Pure MPI consistently faster over Infinipath, but hybrid (1 MPI) faster than pure MPI over GigE above 16 cores. 96

4.16 Movout routine timing in large simulation on Merlin over Infiniband interconnect. Pure MPI code consistently faster than hybrid (1 MPI) and hybrid (2 MPI). 97

4.17 Sum routine timings for the large simulation on Merlin. Hybrid codes significantly outperform the pure MPI code due to reduced numbers of processes needed to communicate. 98

4.18 Absolute difference between average pure MPI timing and average hybrid timing for the forces routine on Merlin (line), presented with error bars representing the sum of errors on both forces timing measurements, and difference between both minimum observed timing for both hybrid and pure MPI code (markers) . Large difference between codes at small counts decreases rapidly as the number of cores increases. 100

4.19 Routine timing breakdown for the large simulation over both interconnects on CSEEM64T at 96 cores. 103

5.1 Speedup for Test 10 on both clusters for pure MPI, hybrid (1 MPI) and hybrid (2 MPI). Hybrid (1 MPI) and hybrid (2 MPI) achieve higher speedup than pure MPI at larger core counts for each cluster. 114

5.2 Speedup for Test 20 on both clusters for pure MPI, hybrid (1 MPI) and hybrid (2 MPI). Hybrid (1 MPI) and hybrid (2 MPI) achieve higher speedup than pure MPI at larger core counts for each cluster. 115

5.3 Average total time for Test 10 on Merlin, presented with minimum timing and error bars of 1 SEM. Pure MPI code performs better at lower core counts, hybrid (1 MPI) and hybrid (2 MPI) codes perform better than pure MPI above 256 cores. 116

5.4 Average total time for Test 20 on Merlin, presented with minimum timing and error bars of 1 SEM. Pure MPI code performs better at lower core counts, hybrid (1 MPI) and hybrid (2 MPI) codes perform better than pure MPI above 128 cores. 116

5.5 Average total time for Test 30 on Merlin, presented with minimum timing and error bars of 1 SEM. Pure MPI code performs better at lower core counts, hybrid (1 MPI) and hybrid (2 MPI) codes perform better than pure MPI above 64 cores. 117

5.6 Average total time for Test 10 on Stella, presented with minimum timing and error bars of 1 SEM. Pure MPI outperforms both hybrid codes at lower core counts, hybrid (1 MPI) and hybrid (2 MPI) perform better than pure MPI above 64 cores. 117

5.7 Average total time for Test 20 on Stella, presented with minimum timing and error bars of 1 SEM. Pure MPI outperforms both hybrid codes at lower core counts, hybrid (1 MPI) and hybrid (2 MPI) perform better than pure MPI above 64 cores. 118

5.8 Average total time for Test 30 on Stella, presented with minimum timing and error bars of 1 SEM. Pure MPI outperforms both hybrid codes at lower core counts, hybrid (1 MPI) and hybrid (2 MPI) perform better than pure MPI above 64 cores. 118

5.9 Communication profile for Test 10, showing average data transferred per process per time step and total data transferred per time step. 5% error bars shown. Hybrid codes have lower total data transfer per time step than the pure MPI code at all core counts, and lower average data transferred per process per time step above 64 cores. 122

5.10 Communication profile for Test 20, showing average data transferred per process per time step and total data transferred per process per time step. 5% error bars shown. Hybrid codes have lower total data transfer per time step than the pure MPI code but higher average data transferred per process per time step at all core counts. 122

5.11 Average total communication time for Test 30 on Merlin, presented with minimum timing and error bars of 1 SEM. Above 64 cores the hybrid codes spend less time carrying out communication than the pure MPI code. 123

5.12 Average total communication time for Test 30 on Stella, presented with minimum timing and error bars of 1 SEM. Hybrid (1 MPI) code consistently spends less time carrying out communication than the pure MPI code. 124

5.13 Communication time as a percentage of total time for Test 10 on both clusters. Communication makes up less of the total runtime in the hybrid codes on both clusters, with the gap between pure MPI and hybrid codes increasing as the number of cores increases. 125

5.14 Average synchronisation time for Test 10 on Stella, presented with minimum timing and error bars of 1 SEM. Hybrid codes consistently perform better than pure MPI over 10GigE connection. 126

5.15 Average synchronisation time for Test 10 on Merlin, presented with minimum timing and error bars of 1 SEM. Hybrid codes perform better at most core counts over Infiniband connection. 127

5.16 Average collective communication time for Test 20 on Merlin, presented with minimum timing and error bars of 1 SEM. Below 64 cores the pure MPI code performs better than both hybrid codes, while above 64 cores the hybrid codes perform better than pure MPI. 129

5.17 Average ollective communication time for Test 20 on Stella, presented with minimum timing and error bars of 1 SEM. Hybrid (1 MPI) code performs better than pure MPI above 16 cores, while hybrid (2 MPI) code performs better above 32 cores. 129

5.18 Average Point-to-Point communication time for Test 30 on Merlin, presented with minimum timing and error bars of 1 SEM. Pure MPI spends more time carrying out point-to-point communication than hybrid (1 MPI) code at all core counts, and than hybrid (2 MPI) code at all core counts other than 16 and 64. . 131

5.19 Average Point-to-Point communication time for Test 30 on Stella, presented with minimum timing and error bars of 1 SEM. Both hybrid codes spend less time carrying out point-to-point communication than the pure MPI code at all core counts. 132

5.20 two_body_forces loop timing for Test 20 on Merlin. Difference between average pure MPI and average hybrid timings presented as line with errors bars representing combined errors from both measurements. Difference between minimum observed timings presented as marker. Overheads shrink rapidly as the number of cores increases (and subdomain sizes per core decrease). 133

5.21 two_body_forces loop timing for Test 30 on Merlin. Difference between average pure MPI and average hybrid timings presented as line with errors bars representing combined errors from both measurements. Difference between minimum observed timings presented as marker. Differences between minimums shrink rapidly as the number of cores increases (and subdomain sizes per core decrease). 134

5.22 two_body_forces loop timing for Test 20 & 30 on Stella. Overheads shrink rapidly as the number of cores increases (and subdomain sizes per core decrease). 134

5.23 Average metal_ld_compute routine timing for Test 20 on Merlin, presented with minimum timing and error bars of 1 SEM. Hybrid code performs better than pure MPI code at high core counts, taking less time to run. Performance difference shrinks rapidly as number of cores increases. 136

5.24 Average metal_ld_compute routine timing for Test 30 on Merlin, presented with minimum timing and error bars of 1 SEM, presented with minimum timing and error bars of 1 SEM. Hybrid code performs better than pure MPI code at high core counts, taking less time to run. Performance difference shrinks rapidly as number of cores increases. 137

5.25 Average metal_ld_compute routine timing for Test 20 & 30 on Stella, presented with minimum timing and error bars of 1 SEM. Overall the pure MPI code performs better than the hybrid codes, except for in two cases. Performance difference shrinks as the number of cores increases. 137

6.1 Differences between halo sizes and communication for hybrid message passing + shared memory and pure MPI codes. 146

(a) Hybrid Code Decomposition & Communication. 146

(b) MPI Code Decomposition & Communication. 146

6.2 Predicted number of particles communicated per time step for DL_Poly Test 20 on 4-64 nodes of a two-socket quad-core cluster. 5% error bars shown. 148

6.3 Actual data communicated per time step for DL_Poly Test 20 on 4-32 nodes of a two-socket quad-core cluster. 5% error bars shown. 149

6.4 Predicted point-to-point communication time for DL_Poly Test 20 on 1 to 64 nodes of a dual-socket quad-core cluster, shown with max and min estimates based on parameter error. 153

6.5 Actual point-to-point communication time for DL_Poly Test 20 on 1 to 64 nodes of a dual-socket quad-core cluster. 154

6.6 Predicted collective communication time for DL_Poly Test 10 on 1 to 64 nodes of a dual-socket quad-core cluster, shown with min and max estimates based on parameter error. 157

6.7 Actual collective communication time for DL_Poly Test 10 on 1 to 64 nodes of a dual-socket quad-core cluster. 157

6.8 Comparison of predicted and actual collective communication time, 128-512 cores of a dual-socket quad-core cluster 159

6.9 Actual total communication time for DL_Poly Test 20 on 1 to 64 nodes of a dual-socket quad-core cluster. 162

6.10 Predicted communication time for DL_Poly Test 20 on 1 to 64 nodes of a dual-socket quad-core cluster. 164

6.11 Predicted and actual total communication time for DL_Poly Test 30 on 1 to 64 nodes of a dual-socket quad-core cluster. 164

(a) Predicted total comms time, DL_Poly Test 30 164

(b) Actual total comms time, DL_Poly Test 30 164

6.12 Predicted execution time for main work loop of DL_Poly Test 10 on 1 to 64 nodes of a dual-socket quad-core cluster. Maximum and minimum bounds very close to predicted values. 169

6.13 Actual execution time for main work loop of DL_Poly Test 10 on 1 to 64 nodes of a dual-socket quad-core cluster. 169

6.14 Predicted difference in execution time of work loop of DL_Poly Test 10 on 1 to 64 nodes of a dual-socket quad-core cluster. 170

6.15 Actual difference in execution time of work loop of DL_Poly Test 10 on 1 to 64 nodes of a dual-socket quad-core cluster. 171

6.16 Predicted total runtime of DL_Poly Test 20 on 1 to 16 nodes of a dual-socket quad-core cluster. 176

6.17 Actual total runtime of DL_Poly Test 20 on 1 to 16 nodes of a dual-socket quad-core cluster. 177

6.18 Predicted total runtime of DL_Poly Test 30 on 1 to 64 nodes of a dual-socket quad-core cluster. 178

6.19 Actual total runtime of DL_Poly Test 30 on 1 to 64 nodes of a dual-socket quad-core cluster. 179

6.20 Predicted performance on possible future hardware, 16 cores per node, doubled communication performance. 185

List of Tables

3.1 OpenMP Clause Timing & Overheads, with errors (+/-) as provided by benchmark software. 59

3.2 Nodes, total number of cores, total MPI processes and OpenMP threads per process (TPP), Merlin 67

3.3 Nodes, total number of cores, total MPI processes and OpenMP threads per process (TPP), CSEEM64T 67

3.4 Nodes, total number of cores, total MPI processes and OpenMP threads per process (TPP), Stella 67

4.1 Profile of MD application, 4 Nodes, Merlin. 71

4.2 Distribution of processor cores across dimensions for three-dimensional data decomposition 74

4.3 Example processor ID, physical ID and core ID numbering 75

4.4 Timing effects of cache sharing. Times in seconds. MPI code affected more by cache sharing than hybrid code, but effects on overall performance are small. . 77

4.5 Ratios between average amount of data sent per process for all three classes of code. Core counts where ratio is similar to number of MPI processes (so where total amount of data transferred will be similar) are highlighted in bold. 91

4.6 Absolute difference between minimum observed pure MPI timing and minimum observed hybrid timing for the forces routine on Merlin running the small simulation. Absolute differences given in seconds. 99

4.7 Absolute difference between minimum observed pure MPI timing and minimum observed hybrid timing for the forces routine on Merlin running the large simulation. Absolute differences given in seconds. 100

4.8 Minimum observed routine timing breakdown for the large simulation running over the Infiniband interconnect on Merlin. (Times in seconds). 102

4.9 Minimum observed routine timing breakdown for the large simulation over both interconnects at 96 cores on CSEEM64T. Times in seconds. 103

5.1 Profile of the DL_Poly application running Test 20 on 4 nodes of Merlin. . . . 109

5.2 Minimum observed total timing results for all three tests on both Stella and Merlin. Times given in seconds and represent the minimum observed total elapsed wall time as recorded by the master MPI process. 120

5.3 Absolute difference between pure MPI and hybrid timing for the main work loop in Test 30 running on Stella. Absolute differences given in seconds. 135

5.4 Absolute difference between pure MPI and hybrid timing for the main work loop in Test 30 running on Merlin. Absolute differences given in seconds. . . . 136

6.1 Message sizes in each direction for a three-dimensional decomposition of a domain of cells. 147

6.2 Parameter estimates for Point-to-Point performance model on Merlin, with errors of estimates, and the percentage error these errors represent. 152

6.3 Percentage errors for Point-to-Point communication performance model estimates to actual minimum observed performance times, Test 10, 20 and 30 on Merlin. Values in bold indicate core counts where the actual runtime falls between the bounds given by the maximum and minimum predicted runtimes. . 155

6.4 Parameter estimates for collective communication performance model, pure MPI data, Merlin with errors of estimates, and the percentage error these errors represent. 158

6.5 Comparison of predicted and actual collective communication time, 128-512 cores of a dual-socket quad-core cluster 159

6.6 Percentage errors for collective communication performance model estimates to actual minimum observed performance times, Test 10, 20 and 30 on Merlin. Collective communication has time of 0 seconds for Hybrid (1 MPI) code at 8 cores (only 1 MPI process is used), so no relative percentage error is calculable. Values in bold indicate core counts where the actual runtime falls between the bounds given by the maximum and minimum predicted runtimes. 160

6.7 Parameter estimates for total communication performance model, Tests 10, 20 and 30, pure MPI data, Merlin with errors of estimates, and the percentage error these errors represent. 163

6.8 Percentage errors for total communication performance model estimates to actual minimum observed performance times, Test 10, 20 and 30 on Merlin. Values in bold indicate core counts where the actual runtime falls between the bounds given by the maximum and minimum predicted runtimes. 165

6.9 Parameter estimates for work loop overheads, Merlin with errors of estimates, and the percentage error these errors represent. 170

6.10 Percentage errors for direct overheads performance model estimates to actual minimum observed performance times, Test 10, 20 and 30 on Merlin. Values in bold indicate core counts where the actual runtime falls between the bounds given by the maximum and minimum predicted runtimes. 172

6.11 Parameter estimates for overall performance, Stella with errors of estimates, and the percentage error these errors represent. 176

6.12 Parameter estimates for overall performance, Merlin with errors of estimates, and the percentage error these errors represent. 178

6.13 Percentage errors for overall performance model estimates to actual minimum observed performance times, Test 10, 20 and 30 on Stella. Values in bold indicate core counts where the actual runtime falls between the bounds given by the maximum and minimum predicted runtimes. 180

6.14 Percentage errors for overall performance model estimates to actual minimum observed performance times, Test 10, 20 and 30 on Merlin. Values in bold indicate core counts where the actual runtime falls between the bounds given by the maximum and minimum predicted runtimes. 181

Listings

3.1 Hybrid Job Script, DL_Poly, Merlin 64
4.1 OpenMP additions to forces main work Loop 72
5.1 OpenMP additions to two_body_forces 111

Bibliography

[1] L. Adhianto and B. Chapman. Performance Modelling of Communication and Computation in Hybrid MPI and OpenMP Applications. *Simulation Modelling Practice and Theory*, 15(4):481–491, 2007.

[2] A. Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, 1992.

[3] I. Ahmad, S. Ranka, and S. Khan. Using Game Theory for Scheduling Tasks on Multi-Core Processors for Simultaneous Optimization of Performance and Energy. In *Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–6. IEEE, April 2008.

[4] S. Alam, P. Agarwal, S. Hampton, and J. Vetter. Impact of Multicores on Large-Scale Molecular Dynamics Simulations. In *Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–7. IEEE, April 2008.

[5] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman. LogGP-Incorporating Long Messages into the LogP Model for Parallel Computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, July 1997.

[6] Y. Alexeev, B. Allan, R. Armstrong, D. Bernholdt, T. Dahlgren, D. Gannon, C. Janssen, J. Kenny, M. Krishnan, J. Kohl, G. Kumfert, L. McInnes, J. Nieplocha, S. Parker, C. Rasmussen, and T. Windus. Component-based Software for High-Performance Scientific Computing. *Journal of Physics: Conference Series*, 16:536–540, January 2005.

[7] B. Armstrong, H. Bae, R. Eigenmann, and F. Saied. HPC Benchmarking and Performance Evaluation with Realistic Applications. *Proceedings of the SPEC Benchmarking Workshop*, 2006.

[8] B. Armstrong, S. Kim, and R. Eigenmann. Quantifying Differences between OpenMP and MPI using a Large-Scale Application Suite. In *High Performance Computing*, volume 1940 of *Lecture Notes in Computer Science*, pages 482–493. Springer, 2000.

[9] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and Others. The Landscape of Parallel Computing Research: A View from Berkeley. *Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December*, 18(2006-183):19, 2006.

[10] 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*, 17(10):1329–1361, August 2005.

[11] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Fine-Grained Multithreading Support for Hybrid Threaded MPI Programming. *International Journal of High Performance Computing Applications*, 24(1):49–57, Feb. 2010.

[12] P. Balaji, W. Feng, D. Panda, et al. The Convergence of Ethernet and Ethernot: A 10-Gigabit Ethernet Perspective. Technical report, OSU-CUSRC-1/06-TR10, Ohio State University, 2006.

[13] A. Basumallik and R. Eigenmann. Towards Automatic Translation of OpenMP to MPI. In *Proceedings of the 19th annual International Conference on Supercomputing*, pages 189–198. ACM, 2005.

[14] H. Berendsen, D. van der Spoel, and R. van Drunen. Gromacs: A message-passing parallel molecular dynamics implementation. *Computer Physics Communications*, 91(1Ð3):43 – 56, 1995.

[15] J. Berthou. Comparing OpenMP, HPF, AND MPI Programming: A Study Case. *International Journal of High Performance Computing Applications*, 15(3):297–309, Aug. 2001.

[16] H. Boehm. Threads Cannot be Implemented as a Library. In *ACM SIGPLAN Notices*, pages 261–268. ACM New York, NY, USA, 2005.

[17] R. Brightwell. A Comparison of Three MPI Implementations for Red Storm. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3666 of *Lecture Notes in Computer Science*, pages 425–432. Springer, 2005.

[18] R. Brightwell. Exploiting Direct Access Shared Memory for MPI On Multi-Core Processors. *International Journal of High Performance Computing Applications*, 24(1):69–77, Jan. 2010.

[19] G. Bronevetsky, J. Gyllenhaal, and B. Supinski. CLOMP: Accurately Characterizing OpenMP Application Overheads. In *OpenMP in a New Era of Parallelism*, volume 5004 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 2008.

[20] F. Broquedis, F. Diakhaté, S. Thibault, O. Aumage, R. Namyst, and P. Wacrenier. Scheduling Dynamic OpenMP Applications over Multicore Architectures. In *OpenMP in a New Era of Parallelism*, volume 5004 of *Lecture Notes in Computer Science*, pages 170–180. Springer, 2008.

[21] H. Brunst and B. Mohr. Performance Analysis of Large-scale OpenMP and Hybrid MPI/OpenMP Applications with VampirNG. In *OpenMP Shared Memory Parallel Programming*, volume 4315 of *Lecture Notes in Computer Science*, pages 5–14. Springer, 2008.

[22] J. Bull. Measuring Synchronization and Scheduling Overheads in OpenMP. In *Proceedings of the 1st European Workshop on OpenMP*, 1999.

[23] J. Bull, J. Enright, and N. Ameer. A Microbenchmark Suite for Mixed-Mode OpenMP/MPI. In *Evolving OpenMP in an Age of Extreme Parallelism*, volume 5568 of *Lecture Notes in Computer Science*, pages 118–131. Springer, 2009.

[24] I. Bush, C. Noble, and R. Allan. Mixed OpenMP and MPI for Parallel Fortran Applications. *European Workshop on OpenMP (EWOMP2000)*, 2000.

[25] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The Impact of Multicore on Math Software. In *Proceedings of the 8th International Conference on Applied Parallel Computing: State of the Art in Scientific Computing*, volume 4699, pages 1–10. Springer, 2007.

[26] F. Cappello and D. Etiemble. MPI versus MPI+ OpenMP on the IBM SP for the NAS Benchmarks. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, page 12, 2000.

[27] F. Cappello and G. Krawezik. Performance Comparison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors. In *Proceedings of the 15th annual ACM symposium on Parallel Algorithms and Architectures*, pages 118–127. ACM New York, NY, USA, 2003.

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

[29] L. Chai, Q. Gao, and D. Panda. Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 471–478. IEEE, May 2007.

[30] B. Chapman, F. Bregier, A. Patil, and A. Prabhakar. Achieving Performance under OpenMP on ccNUMA and Software Distributed Shared Memory Systems. *Concurrency and Computation: Practice and Experience*, 14(8-9):713–739, July 2002.

[31] M. Chorley and D. Walker. Performance Analysis of a Hybrid MPI/OpenMP Application on Multi-core Clusters. *Journal of Computational Science*, 1(3), August 2010.

[32] M. Chorley, D. Walker, and 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(3):196–211, July 2009.

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

[34] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *ACM SIGPLAN Notices*, 28(7):1–12, July 1993.

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

[36] A. Debudaj-Grabysz and R. Rabenseifner. Nesting OpenMP in MPI to Implement a Hybrid Communication Method of Parallel Simulated Annealing on a Cluster of SMP Nodes. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3666 of *Lecture Notes in Computer Science*, pages 18–27. Springer, 2005.

[37] V. Dimakopoulos, P. Hadjidoukas, and G. Philos. A Microbenchmark Study of OpenMP Overheads Under Nested Parallelism. In *OpenMP in a New Era of Parallelism*, volume 5004 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2008.

[38] R. Dimitrov and A. Skjellum. Software Architecture and Performance Comparison of MPI/Pro and MPICH. In *Computational Science, ICCS 2003*, volume 2659 of *Lecture Notes in Computer Science*, pages 695–695. Springer, 2003.

[39] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur. Hybrid Parallel Programming with MPI and Unified Parallel C. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, pages 177–186, Beltinoro, Italy, 2010.

[40] S. Dong and G. Karniadakis. P-Refinement and P-Threads. *Computer Methods in Applied Mechanics and Engineering*, 192(19):2191–2201, 2003.

[41] J. Dongarra. Trends in High-Performance Computing: A Historical Overview and Examination of Future Developments. *IEEE Circuits Devices Magazine*, 22(1):22—27, 2006.

[42] J. Dongarra, D. Gannon, G. Fox, and K. Kennedy. The Impact of Multicore on Computational Science Software. *CTWatch Quarterly*, 3(1):1–10, 2007.

[43] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier. High-Performance Computing: Clusters, Constellations, MPPs, and Future Directions. *Computing in Science & Engineering*, 7(2):51–59, 2005.

[44] N. Drosinos and N. Koziris. Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, page 15, 2004.

[45] A. Duran, M. Gonzàlez, and J. Corbalán. Automatic Thread Distribution for Nested Parallelism in OpenMP. *Proceedings of the 19th annual International Conference on Supercomputing (ICS)*, pages 121–130, 2005.

[46] S. Fide and S. Jenks. Architecture Optimizations for Synchronization and Communication on Chip Multiprocessors. In *Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–8. IEEE, April 2008.

[47] E. Gabriel, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 353–377. Springer, 2004.

[48] W. Gropp and E. Lusk. Reproducible Measurements of MPI Performance Characteristics. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1697 of *Lecture Notes in Computer Science*, pages 681–681. Springer, 1999.

[49] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[50] W. Gropp and R. Thakur. Issues in Developing a Thread-Safe MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science*, pages 12–21. Springer, 2006.

[51] D. Grove and P. Coddington. Precise MPI Performance Measurement using MPIBench. In *Proceedings of HPC Asia*, pages 1–14, 2001.

[52] G. Hager, G. Jost, and 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.

[53] Y. He and C. Ding. An Evaluation of MPI and OpenMP Paradigms for Multi-Dimensional Data Remapping. In *OpenMP Shared Memory Parallel Programming*, volume 2716 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2003.

[54] P. Henon, P. Ramet, and J. Roman. On Using an Hybrid MPI Thread Programming for the Implementation of a Parallel Sparse Direct Solver on a Network of SMP Nodes. In *Parallel Processing and Applied Mathematics*, volume 3911 of *Lecture Notes in Computer Science*, pages 1050–1057. Springer, 2006.

[55] D. Henty. Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, page 10, 2000.

[56] O. Hernandez, F. Song, B. Chapman, J. Dongarra, B. Mohr, S. Moore, and F. Wolf. Performance Instrumentation and Compiler Optimizations for MPI/OpenMP Applications. In *OpenMP Shared Memory Parallel Programming*, volume 4315 of *Lecture Notes in Computer Science*, pages 267–278. Springer, 2008.

[57] T. Hilbrich, M. Müller, and B. Krammer. MPI Correctness Checking for OpenMP/MPI Applications. *International Journal of Parallel Programming*, 37(3):277–291, April 2009.

[58] R. Hockney. The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, 1994.

[59] R. Hockney and J. Eastwood. *Computer simulation using particles*. Taylor & Francis, 1988.

[60] J. Hoeflinger, P. Alavilli, T. Jackson, and B. Kuhn. Producing Scalable Performance with OpenMP: Experiments with two CFD applications. *Parallel Computing*, 27(4):391–413, 2001.

[61] M. Howison, E. Bethel, and H. Childs. MPI-Hybrid Parallelism for Volume Rendering on Large Multi-core Systems. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, page 10, 2010.

[62] S. Hunold and T. Rauber. Reducing the Overhead of Intra-Node Communication in Clusters of SMPs. In *Parallel and Distributed Processing and Applications*, volume 3758 of *Lecture Notes in Computer Science*, pages 58–65. Springer, 2005.

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

[64] Intel. Intel MPI 3.2 Release Notes. `http://software.intel.com/en-us/intel-mpi-library/`, 2009.

[65] Intel. Intel Trace Collector and Analyzer. `http://software.intel.com/en-us/intel-trace-analyzer/`, 2009.

[66] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. *NASA Ames Research Center, Technical Report NAS-99-011*, 1999.

[67] M. Jones, R. Yao, and C. Bhole. Hybrid MPI-OpenMP Programming for Parallel OSEM PET Reconstruction. *IEEE Transactions on Nuclear Science*, 53(5):2752–2758, 2006.

[68] G. Jost, H. Jin, D. an Mey, and F. Hatay. Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster. In *Proceedings of the 5th European Workshop on OpenMP (EWOMP03)*, 2003.

[69] C. Kartsaklis, I. Todorov, and W. Smith. DL_POLY 3: Hybrid CUDA/OpenMP porting of the non-bonded force-field for two-body systems. In *Symposium on Chemical Computations on GPGPUs*. 240th ACS National Meeting and Exposition, Boston, 2010.

[70] V. Kazempour, A. Fedorova, and P. Alagheband. Performance Implications of Cache Affinity on Multicore Processors. In *Euro-Par 2008 - Parallel Processing*, volume 5168 of *Lecture Notes in Computer Science*, pages 151–161. Springer, 2008.

[71] T. Kielmann, H. Bal, and K. Verstoep. Fast Measurement of LogP Parameters for Message Passing Platforms. *Parallel and Distributed Processing*, pages 1176–1183, 2000.

[72] M. Koop, R. Kumar, and D. Panda. Can Software Reliability Outperform Hardware Reliability on High Performance Interconnects? A Case Study with MPI over Infiniband. In *Proceedings of the 22nd annual International Conference on Supercomputing (ICS)*, pages 145–154. ACM Press, 2008.

[73] M. Koop, S. Sur, Q. Gao, and D. Panda. High Performance MPI Design using Unreliable Datagram for Ultra-Scale InfiniBand Clusters. In *Proceedings of the 21st Annual International Conference on Supercomputing*, pages 180–189. ACM, 2007.

[74] D. Kuck. Productivity in High Performance Computing. *International Journal of High Performance Computing Applications*, 18(4):489–504, 2004.

[75] B. Kuhn, P. Petersen, and E. O'Toole. OpenMP Versus Threading in C/C++. *Concurrency: Practice and Experience*, 12(12):1165–1176, October 2000.

[76] R. Kumar, A. Mamidala, and D. Panda. Scaling alltoall collective on multi-core systems. In *Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–8. IEEE, 2008.

[77] H. Kwak, B. Lee, A. Hurson, S. Yoon, and W. Hahn. Effects of Multithreading on Cache Performance. *IEEE Transactions on Computers*, 48(2):176–184, 1999.

[78] T. Le and J. Rejeb. A Detailed MPI Communication Model for Distributed Systems. *Future Generation Computer Systems*, 22(3):269–278, 2006.

[79] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. Panda. Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, pages 58–58, 2003.

[80] J. Liu, W. Jiang, P. Wyckoff, D. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS), 2004*, pages 16–25. IEEE, 2004.

[81] E. Lusk and A. Chan. Early Experiments with the OpenMP/MPI Hybrid Programming Model. In *OpenMP in a New Era of Parallelism*, volume 5004 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2008.

[82] D. Mallón, G. Taboada, C. Teijeiro, J. Touriño, B. Fraguela, A. Gómez, R. Doallo, and J. Mouriño. Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, page 184, 2009.

[83] B. Manaskasemsak, P. Uthayopas, and A. Rungsawang. A Mixed MPI-Thread Approach for Parallel Page Ranking Computation. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pages 1223–1233. Springer, 2006.

[84] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, 1994.

[85] F. Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, 1993.

[86] K. Nakajima. OpenMP/MPI Hybrid vs. Flat MPI on the Earth Simulator: Parallel Iterative Solvers for Finite Element Method. In *High Performance Computing*, volume 2858 of *Lecture Notes in Computer Science*, pages 486–499. Springer, 2003.

[87] I. Nielsen and C. Janssen. Multi-threading: a new Dimension to Massively Parallel Scientific Computation. *Computer Physics Communications*, 128(1):238–244, 2000.

[88] J. Nieplocha, R. Harrison, and R. Littlefield. Global Arrays: A Portable Shared-Memory Programming Model for Distributed Memory Computers. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 340–349. ACM, 1994.

[89] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, May 2006.

[90] NVIDIA Corporation. NVIDIA CUDA Programming Guide, 2010.

[91] H. Ong and P. Farrell. Performance Comparison of LAM/MPI, MPICH, and MVICH on a Linux Cluster connected by a Gigabit Ethernet Network. In *Proceedings of the 4th annual Linux Showcase & Conference*, 2000.

[92] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 2. `http://www.openmp.org`, 2005.

[93] M. Ott, T. Klug, J. Weidendorfer, and C. Trinitis. autopin–Automated Optimization of Thread-to-Core Pinning on Multicore Systems. In *Proceedings of 1st Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2008.

[94] C. Pheatt. Intel®threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[95] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. Fagg, E. Gabriel, and J. Dongarra. Performance analysis of MPI collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[96] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995.

[97] R. Rabenseifner. Hybrid Parallel Programming on HPC platforms. In *proceedings of the Fifth European Workshop on OpenMP, EWOMP*, pages 22–26, 2003.

[98] R. Rabenseifner. Hybrid Parallel Programming: Performance Problems and Chances. In *Proceedings of the 45th Cray User Group Conference*, pages 12–16, 2003.

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

[100] A. Rane and D. Stanzione. Experiences in Tuning Performance of Hybrid MPI/OpenMP Applications on Quad-Core Systems. In *Proceedings of 10th LCI Internationall Conference on High-Performance Clustered Computing*, 2009.

[101] R. Rosenberg, G. Norton, J. Novarini, W. Anderson, and M. Lanzagorta. Modelling Pulse Propagation and Scattering in a Dispersive Medium: Performance of MPI/OpenMP Hybrid Code. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 47. ACM/IEEE, 2006.

[102] T. Simon, S. Cable, and M. Mahmoodi. Application Scalability and Performance on Multicore Architectures. In *DoD High Performance Computing Modernization Program Users Group Conference, 2007*, pages 378–381. IEEE, June 2007.

[103] T. Simon and J. McGalliard. Observation and analysis of the multicore performance impact on scientific applications. *Concurrency and Computation: Practice and Experience*, 21(17):2213–2231, 2009.

[104] L. Smith and J. Bull. Development of Mixed Mode MPI/OpenMP Applications. *Scientific Programming*, 9(2-3):83–98, 2001.

[105] L. Smith and P. Kent. Development and Performance of a Mixed OpenMP/MPI Quantum Monte Carlo Code. *Concurrency: Practice and Experience*, 12(12):1121–1129, October 2000.

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

[107] A. Spiegel, D. an Mey, and C. Bischof. Hybrid Parallelization of CFD Applications with Dynamic Thread Balancing. *Applied Parallel Computing. State of the Art in Scientific Computing*, pages 433–441, 2006.

[108] E. Strohmaier, J. Dongarra, H. Meuer, and H. Simon. The Marketplace of High-Performance Computing. *Parallel Computing*, 25(13-14):1517–1544, December 1999.

[109] E. Strohmaier, J. Dongarra, H. Meuer, and H. Simon. Recent Trends in the Marketplace of High Performance Computing. *Parallel Computing*, 31(3):261–273, 2005.

[110] X. Sun and Y. Chen. Reevaluating Amdahl's Law in the Multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, February 2010.

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

[112] R. Thakur. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, Feb. 2005.

[113] R. Thakur and W. Gropp. Test Suite for Evaluating Performance of Multithreaded MPI Communication. *Parallel Computing*, 35(12):608–617, 2009.

[114] The UPC Consortium. The UPC Language Specification V1.2. `http://upc.gwu.edu/docs/upc_specs_1.2.pdf`, 2005.

[115] S. Vadhiyar, G. Fagg, and J. Dongarra. Automatically Tuned Collective Communications. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[116] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice Boltzmann Simulation Optimization on Leading Multicore Platforms. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS) 2008*, pages 1–14, 2008.

[117] F. Wolf and B. Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. *Journal of Systems Architecture*, 49(10-11):421–439, November 2003.

[118] Wolfram Research. Mathematica Version 8.0. Available from http://www.wolfram.com/mathematica/.

[119] C. Wu, C. Yang, K. Lai, and P. Chiu. Designing Parallel Loop Self-Scheduling Schemes by the Hybrid MPI and OpenMP Model for Grid Systems with Multi-Core Computational Nodes. *Journal of Supercomputing*, 59(1):42–60, 2011.