Mixed Mode Programming on HECToR

Anastasios Stathopoulos

August 22, 2010

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2010

Abstract

The majority of the modern HPC systems are clusters of SMP nodes. Mixed mode programming paradigm is considered to exploit the architecture of such systems and deliver better performance than the traditional programming model (message passing) which is commonly used on these systems. HECToR, the UK's national supercomputer, is a clustered SMP system and has recently been upgraded. This project investigates the performance of the mixed mode (MPI + OpenMP) programming model on both the old and the upgraded systems of HECToR.

Five mixed mode benchmark codes are used in this project which are run on both the Cray XT4 and Cray XT6 systems of HECToR. The available performance analysis tools are used for understanding the resulting performance of each benchmark. The investigation of mixed mode programming is performed by identifying how the replacement of the MPI processes with OpenMP threads affects the various parts of each benchmark code.

It was found that there are cases where the mixed mode versions of a program can achieve better performance than the equivalent pure MPI version by improving certain characteristics of the program, such as the load balance, which are the main bottlenecks of the performance of the program. Furthermore, in most of the runs, mixed mode programming reduced the congestion of the network on the XT6 system which is the Achilles' heel of the upgraded system of HECToR.

Contents

1 Introduction **1**

2 Background **3**
2.1 Clustered SMP systems . . . 3
2.2 Mixed mode programming . . . 5
2.2.1 MPI . . . 5
2.2.2 OpenMP . . . 7
2.2.3 Combining MPI and OpenMP . . . 8
2.2.4 Benefits of mixed mode programming . . . 10

3 HECToR and Benchmarks **12**
3.1 HECToR hardware . . . 12
3.1.1 Cray XT4 system . . . 12
3.1.2 Cray XT6 system . . . 13
3.2 HECToR software . . . 14
3.3 Benchmarks . . . 15
3.3.1 DEISA benchmarks . . . 15
3.3.2 NAS Parallel Benchmarks . . . 16
3.3.3 ASC Sequoia benchmarks . . . 18
3.3.4 Jacobi kernel benchmark . . . 18

4 Results and Analysis **20**
4.1 Experimental procedure . . . 20
4.2 CPMD . . . 22
4.2.1 CPMD on Cray XT4 system . . . 22
4.2.2 CPMD on Cray XT6 system . . . 28
4.2.3 Memory usage of CPMD . . . 33
4.3 BQCD . . . 34
4.3.1 BQCD on Cray XT4 system . . . 35
4.3.2 BQCD on Cray XT6 system . . . 40
4.3.3 Memory usage of BQCD . . . 44
4.4 SP-MZ . . . 45
4.4.1 SP-MZ on Cray XT4 system . . . 46
4.4.2 SP-MZ on Cray XT6 system . . . 50
4.4.3 Memory usage of SP-MZ . . . 52

4.5 IRS 53
4.5.1 IRS on Cray XT4 system 54
4.5.2 IRS on Cray XT6 system 59
4.5.3 Memory usage of IRS 62
4.6 Jacobi kernel 63
4.6.1 Jacobi kernel on Cray XT4 system 64
4.6.2 Jacobi kernel on Cray XT6 system 72
4.6.3 Memory usage of Jacobi kernel 79

5 Conclusions and Future Work **81**
5.1 Conclusions 81
5.2 Future Work 82

A Experimental Data **84**
A.1 CPMD 84
A.1.1 CPMD on Cray XT4 system 84
A.1.2 CPMD on Cray XT6 system 85
A.2 BQCD 85
A.2.1 BQCD on Cray XT4 system 85
A.2.2 BQCD on Cray XT6 system 86
A.3 SP-MZ 87
A.3.1 SP-MZ on Cray XT4 system 87
A.3.2 SP-MZ on Cray XT6 system 87
A.4 IRS 88
A.4.1 IRS on Cray XT4 system 88
A.4.2 IRS on Cray XT6 system 88
A.5 Jacobi kernel 89
A.5.1 Jacobi kernel on Cray XT4 system 89
A.5.2 Jacobi kernel on Cray XT6 system 91

List of Tables

3.1 Comparison of Cray XT4 and Cray XT6 systems 14

4.1 Combinations of OpenMP threads per MPI process on Cray XT6 system. 21
4.2 Statistics produced by CrayPAT for runs using 64 processors and 1 molecule problem size (time in seconds and percentage). 24
4.3 Statistics produced by CrayPAT for runs using 64 processors and 128 molecules problem size (time in seconds and percentage). 25
4.4 Statistics produced by CPMD for collective communications on runs using 64 processors and 128 molecules problem size. 25
4.5 Statistics produced by CrayPAT for runs using 64 processors and 256 molecules problem size (time in seconds and percentage). 26
4.6 The dimensions of the real space mesh of each problem size. 27
4.7 Statistics produced by CrayPAT for runs using 256 processors and 1 molecule problem size (time in seconds and percentage). 28
4.8 Comparison of statistics produced by CrayPAT for pure MPI version with 1 molecule problem size run on Cray XT4 using 64 processors and Cray XT6 using 96 processors (time in seconds and percentage). . . 30
4.9 Statistics produced by CrayPAT for runs using 96 processors and 1 molecule problem size (time in seconds and percentage). 31
4.10 Statistics produced by CrayPAT for runs using 96 processors and 128 molecules problem size (time in seconds and percentage). 31
4.11 Statistics produced by CrayPAT for runs using 384 processors and 1 molecule problem size (time in seconds and percentage). 32
4.12 Statistics produced by CrayPAT for runs using 64 processors and 24^3x48 lattice size (time in seconds and percentage). 35
4.13 Average sent message statistics per process for 24^3x48 lattice size, using 64 processors. 37
4.14 The dimensions of the process grid of 64, 256 and 1024 processors. . . 38
4.15 Average sent message statistics per process for 24^3x48 lattice size, using 256 processors. 38
4.16 Statistics produced by CrayPAT for runs using 256 processors and 48^3x96 lattice size (time in seconds and percentage). 40
4.17 Average sent message statistics per process for 48^3x96 lattice size, using 256 processors. 40

4.18 The dimensions of the processor grid of 192, 384, 768 and 2304 processors. 41
4.19 Statistics produced by CrayPAT for runs using 192 processors and 24^3x48 lattice size (time in seconds and percentage). 42
4.20 Average sent message statistics per process for 24^3x48 lattice size, using 192 processors. 42
4.21 Statistics produced by CrayPAT for runs using 384 processors and 24^3x48 lattice size (time in seconds and percentage). 43
4.22 Average sent message statistics per process for 24^3x48 lattice size, using 384 processors. 43
4.23 Statistics produced by CrayPAT for runs using 64 processors and problem class C (time in seconds and percentage). 47
4.24 Statistics produced by CrayPAT for runs using 256 processors and problem class D (time in seconds and percentage). 48
4.25 Statistics produced by Scalasca for runs using 64 processors and problem class C (time in seconds and percentage) compiled by the GNU compiler. 50
4.26 Statistics produced by Scalasca for runs using 256 processors and problem class D (time in seconds and percentage) compiled by the GNU compiler. 50
4.27 Statistics produced by CrayPAT for runs using 96 processors and problem class C (time in seconds and percentage). 51
4.28 Statistics produced by CrayPAT for runs using 384 processors and problem class D (time in seconds and percentage). 52
4.29 Statistics produced by CrayPAT for runs using 8 domains and same number of processors (time in seconds and percentage). 56
4.30 Statistics produced by CrayPAT for runs using 216 domains and same number of processors (time in seconds and percentage). 57
4.31 Statistics produced by CrayPAT for runs using 216 domains and 108 processors (time in seconds and percentage). 58
4.32 Statistics produced by CrayPAT for runs of IRS benchmark using 216 domains and 216 processors(time in seconds and percentage). 60
4.33 Average sent message statistics per process on 216 processors using 216 domains. 61
4.34 Statistics produced by CrayPAT for runs of IRS benchmark using 216 domains and 108 processors(time in seconds and percentage). 61
4.35 Hardware counter metrics for pure MPI and funnelled codes. 68
4.36 Optimal work distribution of X dimension on master thread for funnelled code 76

A.1 CPMD benchmark execution time (in seconds) using 64 cores. 84
A.2 CPMD benchmark execution time (in seconds) using 256 cores. 84
A.3 CPMD benchmark execution time (in seconds) using 96 cores. 85
A.4 CPMD benchmark execution time (in seconds) using 256 cores. 85
A.5 BQCD benchmark execution time (in seconds) on 24x24x24x48 lattice. 85

A.6 BQCD benchmark execution time (in seconds) on 48x48x48x96 lattice. 86
A.7 BQCD benchmark execution time (in seconds) on 24x24x24x48 lattice. 86
A.8 BQCD benchmark execution time (in seconds) on 48x48x48x96 lattice. 86
A.9 SP-MZ benchmark execution time (in seconds) compiled with PGI compiler. 87
A.10 SP-MZ benchmark execution time (in seconds) compiled with GNU compiler. 87
A.11 SP-MZ benchmark execution time (in seconds). 87
A.12 IRS benchmark, microseconds per zone-iteration. 88
A.13 IRS benchmark execution time (in seconds). 88
A.14 IRS benchmark, microseconds per zone-iteration. 88
A.15 IRS benchmark execution time (in seconds). 89
A.16 Master Only time(in seconds) on 128 processors 89
A.17 Master Only time(in seconds) on 256 processors 89
A.18 Funnelled time(in seconds) on 128 processors 89
A.19 Funnelled optimal time(in seconds) on 128 processors 90
A.20 Funnelled time(in seconds) on 256 processors 90
A.21 Multiple time(in seconds) on 128 processors 90
A.22 Multiple time(in seconds) on 256 processors 90
A.23 Master Only time(in seconds) on 192 processors 91
A.24 Master Only time(in seconds) on 384 processors 91
A.25 Funnelled time(in seconds) on 192 processors 91
A.26 Funnelled optimal time(in seconds) on 192 processors 92
A.27 Funnelled time(in seconds) on 384 processors 92
A.28 Multiple time(in seconds) on 192 processors 92
A.29 Multiple time(in seconds) on 384 processors 93

List of Figures

2.1 Schematic representation of a distributed memory system, reproduced from [9]. . . . 3
2.2 Schematic representation of a shared memory system, reproduced from [9]. . . . 4
2.3 Schematic representation of a clustered SMP system, reproduced from [9]. . . . 4
2.4 The flow of execution of an OpenMP code when run with six OpenMP threads, reproduced from [5]. . . . 8
2.5 Schematic representation of a hierarchical mixed mode progamming model for a 2D array, reproduced from [6]. . . . 9

3.1 Two-dimensional tiling of three-dimensional mesh, reproduced from [21]. 17
3.2 MPI Distribution, reproduced from [15]. . . . 19

4.1 CPMD execution time ratio of mixed mode versions to pure MPI for 1, 128 and 256 molecules, using 64 processors on Cray XT4. . . . 23
4.2 CPMD execution time ratio of mixed mode versions to pure MPI for 1, 128, 256, 384 and 512 molecules, using 256 processors on Cray XT4. . 27
4.3 CPMD execution time ratio of mixed mode versions to pure MPI for 1, 128 and 256 molecules, using 96 processors on Cray XT6. . . . 29
4.4 CPMD execution time ratio of mixed mode versions to pure MPI for 1, 128, 256, 384 and 512 molecules, using 384 processors on Cray XT6. . 32
4.5 Memory usage of CPMD program for 1 molecule problem size using 64 processors on the Cray XT4 system and 96 processors on the Cray XT6 system. . . . 34
4.6 BQCD execution time ratio of mixed mode versions to pure MPI for 64, 256 and 1024 processors on 24^3x48 lattice. . . . 36
4.7 Performance of two different processor grids of BQCD benchmark using 64 processors on 24^3x48 lattice. . . . 37
4.8 BQCD execution time ratio of mixed mode versions to pure MPI for 256 and 1024 processors on 48^3x96 lattice. . . . 39
4.9 BQCD execution time ratio of mixed mode versions to pure MPI for 96, 384, 768 and 2304 processors on 24^3x48 lattice. . . . 41
4.10 BQCD execution time ratio of mixed mode versions to pure MPI for 384, 768 and 2304 processors on 48^3x96 lattice. . . . 44

4.11 Memory usage of BQCD program for 24^3x48 lattice using 256 processors on the Cray XT4 system and 192 processors on Cray XT6 system. . 45
4.12 SP-MZ time ratio using the PGI compiler on the Cray XT4 system. . . . 47
4.13 Comparison of PGI and GNU compiler on 64 processors. 49
4.14 Comparison of PGI and GNU compilers on 256 processors. 49
4.15 SP-MZ time ratio on the Cray XT6 system. 51
4.16 Memory usage of SP-MZ benchmark for class D problem size using 256 processors on Cray XT4 system and 384 processors on Cray XT6 system. 53
4.17 IRS time ratio per zone-iteration of mixed mode versions to pure MPI. . 55
4.18 IRS execution time ratio of mixed mode versions to pure MPI. 56
4.19 IRS time ratio per zone-iteration of mixed mode versions to pure MPI when the number of domains is not equal to the number of processors. . 57
4.20 IRS execution time ratio of mixed mode versions to pure MPI when the number of domains is not equal to the number of processors. 58
4.21 IRS time ratio per zone-iteration of mixed mode versions to pure MPI. . 59
4.22 IRS execution time ratio of mixed mode versions to pure MPI. 60
4.23 IRS execution time on both Cray XT4 and Cray XT6 systems. 62
4.24 Memory high water mark per node of IRS benchmark on both systems. 63
4.25 Execution time in seconds of Pure MPI and Master Only versions using 128 processors. 66
4.26 Comparison of the execution time of Pure MPI and Master Only versions on 128 and 256 processors. 66
4.27 Execution time in seconds of Pure MPI and Funnelled versions using 128 processors. 68
4.28 Execution time in seconds of Pure MPI and optimised Funnelled versions using 128 processors. 69
4.29 Execution time in seconds of Pure MPI and Multiple versions using 128 processors. 71
4.30 Comparison of the execution time of Multiple code on 128 and 256 processors. 72
4.31 Execution time in seconds of Pure MPI and Master Only versions using 192 processors. 73
4.32 Execution time in seconds of Pure MPI and Funnelled versions using 192 processors. 75
4.33 Execution time in seconds of Pure MPI and optimised Funnelled versions using 192 processors. 76
4.34 Execution time in seconds of Pure MPI and Multiple versions using 192 processors. 77
4.35 Jacobi computation time of each thread of Multiple code on 192 processors. 78
4.36 Memory high water mark per node of master only code on both systems. 79

Acknowledgements

I would like to thank my supervisor, Dr. Mark Bull, for his guidance and his support throughout this project. I also want to thank SAAS, the University of Edinburgh and EPCC for providing the funds for this degree.

My thanks also go to Jason Beech-Brandt and Tom Spelce for their help in porting the BQCD code and the IRS code on HECToR,and Michal Piotrowski for providing his benchmark code.

Finally, I would to like thank my family and my friends for their endless support during the whole year of my studies.

Chapter 1

Introduction

Nowadays, the clock speed of single processor chips is not increasing significantly anymore. Physical limitations, such as heat dissipation, the speed of light, the size of the atoms, make the manufacturing of faster chips very difficult and expensive. However, manufacturers are still able to keep up with Moore's Law by developing multi-core chips where two or more cores are placed on the same silicon and they use the same memory bus.

The arrival of relatively low cost multi-core chips triggered the gradual replacement of MPP (Massively Parallel Processing) systems with clusters of SMP (Symmetric MultiProcessing) nodes. Today, almost all High Performance Computing (HPC) systems consist of nodes of multi-core processors. The memory of such systems is distributed across the nodes and shared by the cores of each node. These nodes are connected by some high speed interconnect mechanism.

Programming such systems has become a very challenging task due to the nature of their architecture. The most common programming model which is used for theses systems is message passing programming model (usually implemented by the MPI library). Message passing model is required for the communication between processes that run on different nodes, but it is also used for the communication of processes that run on the same node. If the MPI library is configured properly, communications on the same node are replaced by reads and writes on the shared memory. Furthermore, the shared memory programming model can be used for processes that run on the same node and share the same memory.

A combination of these two programming models can be used for extracting good performance from clusters of SMP nodes and is called mixed mode or hybrid (MPI + OpenMP) programming. In mixed mode programming, fewer MPI processes than the available cores on each node are run on the node, and the remaining cores are used for running OpenMP threads. This project investigates the performance of mixed mode programming paradigm on HECToR, the UK's national supercomputer.

HECToR is a cluster of SMP nodes. At present, it is under upgrade, moving from

Phase IIa to Phase IIb , and both the old and the upgraded systems are available. Phase IIa is a Cray XT4 system that consists of SMP nodes with single quad-core processor sockets. Phase IIb consists of 24-core nodes and each node is composed by two 12-core processor sockets. Both the clock speed and the memory per core are reduced but the total number of cores and memory per node are increased.

The availability of both systems provides a good opportunity to investigate how the performance of mixed mode programming is affected by the increase of the total cores per node along with changes in other characteristics of the system, such as the size of the shared memory of the cores.

A number of different, well established benchmark codes will be used to investigate the performance of mixed mode programming model on HECToR. The most effective way to evaluate this model is to compare the resulting performance of those mixed mode benchmarks with the performance of their pure MPI version. An in-depth analysis for each benchmark at the resulting performance will be made with the assistance of performance analysis tools that are available on HECToR, such as Craypat and Scalasca.

The structure of the report is presented below:

Chapter 2 provides the required background in order to understand the basic concepts that this project deals with. Clustered SMP systems are introduced in this chapter by describing the components that they consist of. The presentation of MPI and OpenMP standards lays the ground for introducing mixed mode programming which is described in depth along with the different mixed mode schemes and the reasons for the potential gain or loss in performance compared to message passing model.

Chapter 3 contains a detailed presentation of the system on which the performance of mixed mode programming is investigated, HECToR. Later in this chapter, there is a description of the benchmarks that are run on HECToR for the purposes of this project.

Chapter 4 starts with the description of the common components of the experimental procedure that was used applied to all the benchmark codes of the project. Then, the resulting performance of each code is presented and analysed in-depth where the causes of this performance are identified and explained.

Finally, the conclusions about the project and some suggestions for future work are included in Chapter 5.

Chapter 2

Background

The purpose of this chapter is to provide fundamental information about the hardware that this project is applied to. Furthermore, the mixed mode programming model is described along with its building elements. The different mixed mode programming styles are presented and the chapter ends with some reasons about why mixed mode programming model may result in good performance and is worth studying.

2.1 Clustered SMP systems

Mixed mode programming is a parallel programming model that is applied on clusters of SMP nodes in order to take advantage of their architecture and extract good performance from such systems. The architecture of clustered SMP systems is the combination of distributed memory and shared memory architectures. In a distributed memory system each processor has its own local memory and the processors are connected by some interconnect mechanism (Figure 2.1). The processors communicate with each other via explicit message passing.

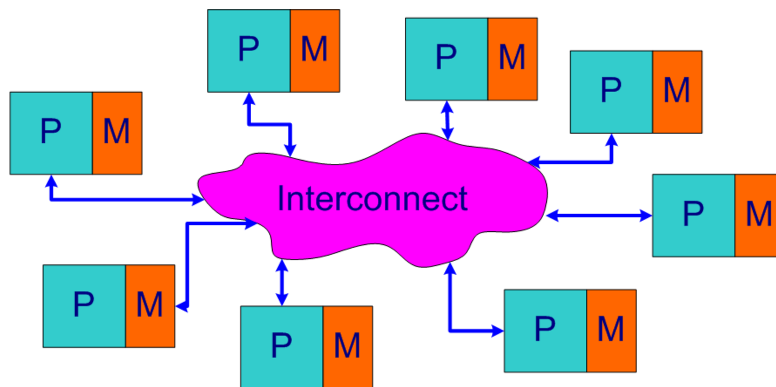

Figure 2.1: Schematic representation of a distributed memory system, reproduced from [9].

A shared memory architecture comprises processors that have access on a global memory (Figure 2.2). Processors access the global memory using a shared address bus and each processor has equal access to all parts of the memory. The communication between the processors is performed by reads/writes to the shared memory.

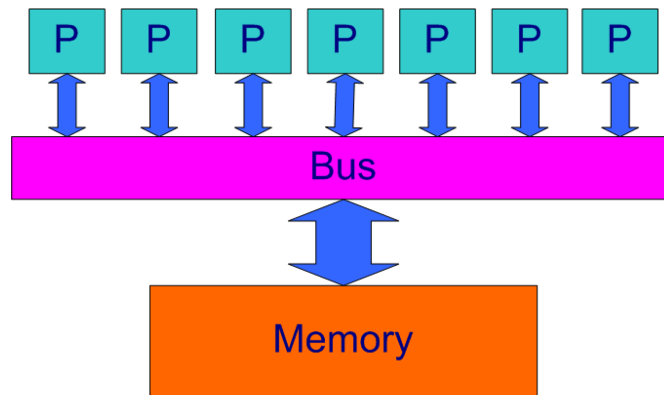

Figure 2.2: Schematic representation of a shared memory system, reproduced from [9].

A clustered SMP system is constructed as a distributed memory machine but with SMP nodes. Many SMP nodes are connected by a fast interconnect circuit, and each node consists of one or more multi-core processors which share the same memory. Cores of a certain node can only access the local memory of the node. Cores that lie on different nodes communicate through the interconnection mechanism using message passing (Figure 2.3).

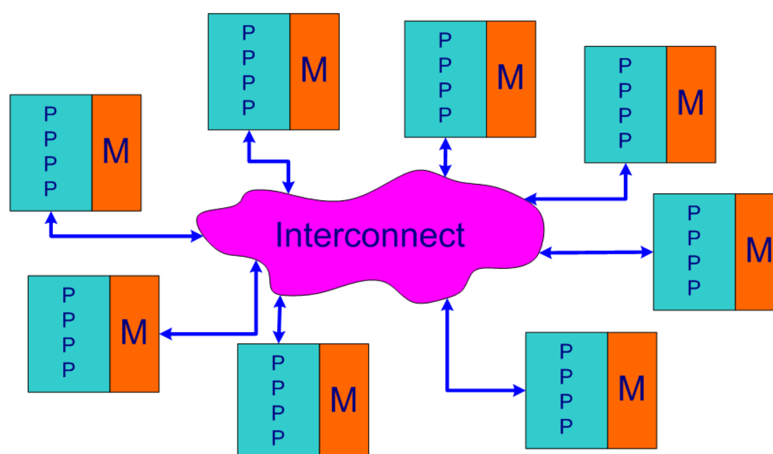

Figure 2.3: Schematic representation of a clustered SMP system, reproduced from [9].

This architecture combines the advantages of both distributed memory and shared memory architectures, but it also inherits some of their disadvantages. An SMP cluster can have almost any number of nodes. An existing system can be expanded by connecting

new SMP nodes on the network. But the scalability of the system relies on a good interconnect mechanism. When many processors try to communicate at the same time, interconnection mechanism can be the bottleneck and the reason for bad performance.

By using SMP nodes instead of single processor nodes, the inter-node communication of the standard distributed memory architecture is reduced and replaced by intra-node communication which is performed by reads/writes at the shared memory of the node. This helps to reduce the congestion in the interconnect mechanism. The performance of the intra-node communication now depends on the bandwidth and latency of the shared memory bus. Simultaneous accesses to the memory by many processors of a certain node may result in congestion in the shared memory bus.

Although this architecture combines the advantages of the architecture that consists of, it is more difficult to take advantage of the mixed architecture. Furthermore, its complexity makes the understanding of the resulting performance a harder task in comparison with the distributed memory architecture and shared memory architecture.

2.2 Mixed mode programming

As it is mentioned in the previous section, clustered SMP systems comprise two different architectures, distributed memory architecture and shared memory architecture. Message passing programming model is used for programming distributed memory systems. This model is implemented by libraries that follow the MPI (Message Passing Interface) standard. Shared memory systems are programmed using implementations of shared memory programming model, such as the OpenMP standard. Mixed mode programming is the combination of those two models which are described in detail in [2], [3] and [6].

2.2.1 MPI

MPI uses a SPMD (Single Program Multiple Data) model where a number of processes run the same set of instructions using different data. Each process has its own data set and communicates with other processes by sending and receiving messages. Message passing programming model achieves parallelism by using many processes working on the same data set which is distributed across these processes. Different execution paths can be followed by each process which is determined by the ID of the process. MPI is highly portable and can be used for both distributed memory systems and shared memory systems. Some vendor specific MPI libraries are able to distinguish communications that are performed on shared memory systems and replace the message passing with reads/writes in the shared memory.

MPI provides routines that perform synchronisation between processes, but they are rarely used and usually only for timing specific parts of the program. Processes are

synchronised when a communication is performed. Thus, the overheads that are introduced by the MPI library and result in poor performance come from the communication between the processes. These overheads are easily identified because they are related with the communication of the processes and can be reduced by minimising communications between them. Some vendors provide optimised communication routines for specific platforms. However, as MPI library is the assembly of parallel programming, the performance of the program fully relies on decisions of the programmer, such as the communication pattern and the data distribution.

MPI defines two types of communication depending on the number of processes that are involved, point-to-point communication and collective communication. Point-to-point communication is the simplest form of communication where one process (source) sends a message to another process (destination). This type of communication relies on matching sends and receives. The message contains the ids of the sender and receiver processes, the data type of the message, the size of the message, the message itself, a special tag, and the group of processes which the sender and receiver belong to (communicator).

Point-to-point communication can be synchronous or asynchronous. On synchronous point-to-point communication send and receive are blocking routines. For the sender, the communication is completed when the message has started being received. For the receiver, the communication is completed when the message is completely received. On asynchronous communications, the communication is completed as soon as the message leaves from the sender. Send and receive are implemented by non-blocking routines. Asynchronous point-to-point communication is often used for overlapping communication and computation when halo swapping is needed. In this case, non-blocking send and receive routines are issued and computation of non-halo data is performed until the communication is completed. The computation then continues to the halo regions of the data.

There are many cases where communication involves a group of processes. For these cases MPI defines collective communications which can implement more complicate communication patterns than point-to-point communication. Collective communications are implemented by blocking routines which are called from every process that participates to the communication (lies in the same communicator). A collective communication can be built by simpler point-to-point communications which produce the same communication pattern, but collective communication routines are usually implemented more efficiently.

There are many routines for different communication patterns. A process can broadcast some data to a group of processes, scatter the data across the processes, gather data from other processes. For example, master process reads data from a file, scatters the content of the file across all the other processes and after the computation, master process gathers the results from the other processes and writes a file. There are also routines for global reduction operations which are used for computing a result which involves data that is distributed across a group of processes. MPI also provides collective communication routines where all processes send data to all processes. A common use of this

routines is the transposition of a matrix which is involved in an FFT.

The time spent by a process waiting on blocking communication routines is often an indicator of load imbalance across the processes of the program. Processes with small work load tend to arrive first at blocking routines and wait for heavily loaded processes in order to perform the blocking communication. Performance analysis tools can identify the time that is spent on the synchronisation of the processes and can help to detect possible load imbalance.

2.2.2 OpenMP

OpenMP is the most commonly used standard for shared memory programming. Library routines and compiler directives are used for parallelising a program. Parallelising a program using OpenMP compiler directives requires minimum effort. However, shared memory programs can only be implemented efficiently on shared memory architecture systems. OpenMP parallelism is based on threads which access shared data and can also have some private variables. The communication of the threads is performed by reads and writes on the shared variables. Similarly to MPI, OpenMP uses a SPMD model where all threads execute the same program, but can follow different execution path that is determined by the unique id that each thread holds.

The program is initially executed by only a single thread (master thread). The program is executed in parallel on distinct sections of the program, called parallel regions which are the basic parallel construct in OpenMP. A parallel region is defined by the proper compiler directives. At the start of the first parallel region, master thread creates a team of threads that execute in parallel the statements which lie inside the parallel region. Before the execution of the program, the programmer sets an environmental variable defining the total number of threads that will be created during the execution of the program. At the end of the parallel region, master thread waits for all threads to complete their work and continues the execution of the next statements sequentially, until another parallel region occurs. During the serial execution of the program, all threads, except for the master thread, are idle (the program runs sequentially) wasting CPU time.

OpenMP is commonly used for parallelising loops inside a program. The parallelisation is achieved by distributing the iterations of the loop among the threads. Each thread is executing its own part of the loop independently from the other threads. The programmer can explicitly assign to threads which iterations of the loops will be executed by each thread, depending on the id of the thread. However, OpenMP provides easier ways of assigning iterations to OpenMP threads. In this case, the programmer uses the proper schedule clause in order to use the provided option that performs the desirable loop distribution.

Inside the parallel regions, threads can access private and shared variables. Private variables are owned and accessed by only one thread. As OpenMP programs follow the SPMD model, each thread has the same private variables and it keeps a different copy of the private variables to its local memory. Shared variables can be read/written by all

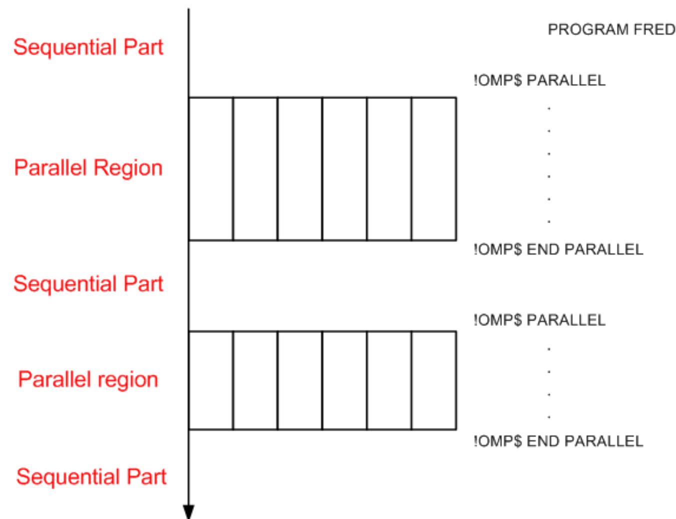

Figure 2.4: The flow of execution of an OpenMP code when run with six OpenMP threads, reproduced from [5].

threads. Additional synchronisation is needed in order to ensure that accesses to shared variables by different threads are performed in the correct order and by only one thread at the time.

OpenMP provides a variety of explicit synchronisation mechanisms. The difference of theses mechanisms has to do with the ease of use and the flexibility they offer along with the synchronisation overheads that they introduce in the program. There are cases where efficient mechanisms (e.g. atomic statements) cannot be used, so the additional overheads can be excessive. OpenMP threads are also implicitly synchronised at the start and the end of parallel regions. Synchronisation errors are not easily detected making debugging a hard task. Usually, small parallel regions are dominated by synchronisation overheads which are not compensated by the computation time that is spend on each thread.

2.2.3 Combining MPI and OpenMP

Now that the architectures that clustered SMP systems consist of are described along with the models that are used for programming these architectures, it becomes more clear that mixed mode programming matches the underlying architecture of clusters of SMP nodes. Message passing programming model is used for the communication between the nodes of the cluster and shared memory programming model is applied inside the SMP nodes. The notion behind this model is to minimize the intra-node communication, and replace the sends/receives of messages with faster reads/writes to the shared memory of the node. MPI is usually used for exchanging data between the processes of the program and OpenMP directives are used for parallelising loops that lie inside the MPI processes (Figure 2.5).

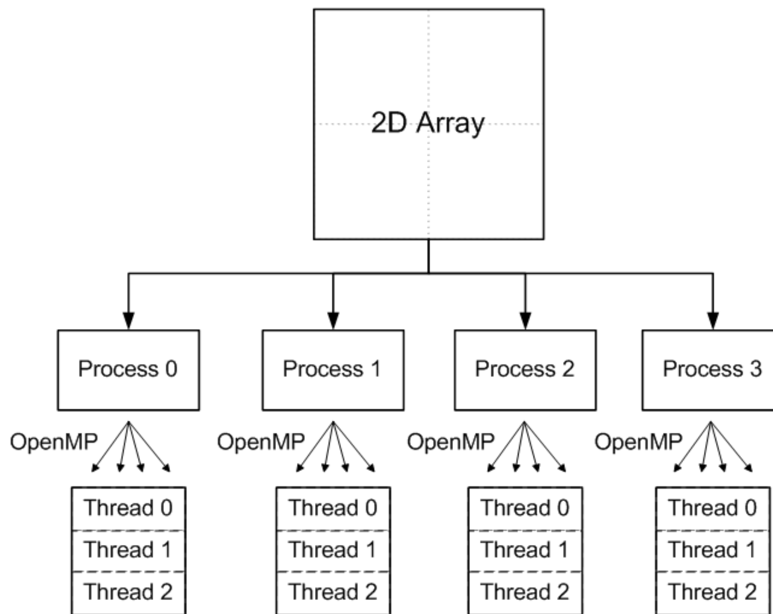

Figure 2.5: Schematic representation of a hierarchical mixed mode progamming model for a 2D array, reproduced from [6].

According to Rabenseifner ([7], [8]), depending on when and by which threads the communications between the processes are performed, the various mixed mode programming schemes can be classified in the following three categories:

- **Master Only**: In this mixed mode scheme, MPI communications are performed outside the parallel region, i.e. by the master thread. Thus, on master only scheme, during the communication of the MPI processes the OpenMP threads are idle and CPU cycles are wasted.
- **Funnelled**: In this mixed mode scheme, MPI is called inside the OpenMP parallel region, but with the restriction that only one thread at the time can perform the communication which is usually funnelled to the master thread. Using this programming scheme it is possible to overlap communication and computation. While the master thread performs the communication between the processes, e.g. halo swapping, the rest of the threads are performing computations that do not need any halo data. In this way, all threads are kept busy and CPU cycles are not wasted. The difficult part of this scheme is to distribute the work evenly across the threads, as the thread that calls the MPI communication routines has more work to do than the others.
- **Multiple**: In this mixed mode scheme, MPI communication is performed inside the parallel region and by more than one thread. Any thread can call MPI routines at any part of the program. The flexibility of this scheme provides the potentials of extracting good performance using mixed mode programming, but it is the most complicated of all three schemes and requires the greatest effort to create an efficient program. Like funnelled, this scheme can also be used for overlapping

computation and communication Both funnelled and multiple schemes rely on the existence of MPI libraries that implement the specific thread safety levels in order to work.

2.2.4 Benefits of mixed mode programming

Nowadays, the majority of the programs that run on clusters of SMP nodes are implemented using MPI. OpenMP cannot be used across the nodes, thus pure OpenMP implementations are restricted to run only on a single node. Converting a pure MPI/pure OpenMP implementation to a mixed mode implementation, or developing a mixed mode program from the scratch requires some effort. There are several cases that developing mixed mode programs is worth the additional effort because they extract better performance than the equivalent pure MPI programs, and these cases are described in [6] and [4].

- **Replicated data**: This strategy is used in MPI codes in order to allow some computations to be performed independently by the processes of the program without exchanging any data and increase the efficiency of the program by reducing the communication between the processes. When replicated data strategy is applied, each process keeps a copy of the data that is very frequently used. In pure MPI, every process of a node would keep a copy of this data, whereas on mixed mode implementations only a copy per SMP node is needed. In this way, larger problems can be studied using replicated data strategy. This becomes more beneficial now that the number of cores per node is increasing and the available memory per core is starting to decrease.
- **Poorly scaling MPI codes**: Mixed mode programming can be beneficial for codes whose scalability is poor as the number of MPI processes is increasing. Load imbalance is one cause of poor scalability and can be cured by reducing the number of MPI processes and replace them with OpenMP threads. In this way, there is a coarser distribution of the load among the processes and the load imbalance is reduced while the level of parallelism is kept the same. There are also cases (fine grain parallelism problems, 1-D domain decomposition) where OpenMP codes scale better than the equivalent MPI codes, but MPI is still needed for inter-node communication. So reducing MPI processes and replacing them with OpenMP threads can improve the performance of the program.
- **Restricted MPI process applications:** Modern HPC systems provide thousands of processors for running applications. However, there are applications which only run with specific numbers of processes, or cannot run with more than a certain number of processes. Furthermore, some MPI implementations cannot handle great number of processes efficiently. In order to achieve higher levels of parallelism and utilise more efficiently such systems, mixed mode programming uses OpenMP threads inside the MPI processes to perform further distribution of the work across more processors.

- **Poorly optimised intra-node MPI**: Some MPI implementations are not tuned for SMP clusters which results in inefficient intra-node communication. MPI should identify that the requested communication involves processes that lie on the same node and replace the sends/receives of messages with reads/writes on the shared memory. Inefficient access to the memory by MPI or even the inability of MPI to distinguish intra-node communication are possible causes of bad performance of pure MPI. Thus, replacing inefficient intra-node communication with fast accesses to the shared memory managed by OpenMP increases the performance of the program significantly.

Although mixed-mode programming model seems very appealing, some previous studies have showed that this model has also some weaknesses. Rabenseifner [8] found that pure MPI codes can achieve higher inter-node bandwidth on some platforms than the equivalent mixed mode ones when only a subset of the threads are participating in the inter-node communication (master only and funnelled schemes).

Master only scheme requires the least developing/maintaining effort of all three schemes and it is the only scheme that is very portable because it does not rely on a thread-safe MPI implementation. However, when master only scheme is used, the OpenMP overheads that may be introduced in order to synchronize the threads before and after the communication are high. As it is mentioned before, in order to produce a very efficient mixed mode program, ease of implementation, maintainability and portability have to be sacrificed, and it is not always guaranteed that the desired performance will be achieved.

The purpose of this project is to investigate mixed mode programming paradigm on HECToR using several benchmark codes. It will be examined whether the cases where mixed mode programming produces better performance than pure MPI are applied to the benchmark codes that will be used, or the bad performance of mixed mode codes is a result of some of the reasons that are described above.

Chapter 3

HECToR and Benchmarks

The purpose of this project is to investigate mixed mode programming on HECToR. This chapter contains a detailed description of HECToR. Moreover, the benchmarks that will be run on HECToR are presented in this chapter.

3.1 HECToR hardware

HECToR (High End Computing Terascale Resource) is the UK' s national supercomputing service. Currently, it consists of two different machines, the Phase IIa machine and the Phase IIb machine. The Phase IIa part of HECToR is a Cray XT5h system which has two components, a Cray XT4 component and a Cray XT2 component (which is assembled from vector processors). The Phase IIb machine is a Cray XT6 system. The benchmark codes will be run on the Cray XT4 and XT6 systems, thus the description of HECToR is focused only on those two systems and is based on the information provided from [10] and [11].

3.1.1 Cray XT4 system

HECToR is a clustered SMP machine. The Cray XT4 system consists of 2832 single socket nodes and each node has an AMD 2.3 GHz quad-core Opteron processor. So, there are 11,832 cores available on the Cray XT4 system which offer 114 TFlops of theoretical peak performance. Each processor has 8 GB of 800MHz, DDR2 main memory available which is shared between the cores of the processor (giving 2 GB per core). All the cores of each processor use the same memory bus to access the memory with a peak bandwidth of 12.8 GB/s. The total main memory of the XT4 system is 24.4 TB.

The nodes are connected with each other using the high bandwidth Cray SeaStar2+ interconnect chip. SeaStar2+ chips are arranged on a 3 dimensional torus with direct links to each of the six neighbouring chips which can achieve 9.6 GB/s peak bi-directional

bandwidth. Each processor has its own SeaStar2+ chip connected to its HyperTransport system with a link that can offer a peak bandwidth of 6.4 GB/s.

Each core of the processor has dedicated level 1 and level 2 caches. There is also a level 3 cache which is shared across all the cores. Data and instructions are stored in two different 64KB, 2-way associative L1 caches. L2 cache is a 16-way associative, 512KB memory which is used for both data and instructions. 64 byte cache lines (8 double precision words) are used from both L1 and L2 caches. The L2 cache is used as victim cache for L1 cache. New data from main memory is directly copied to L1 cache. When L1 cache is full, then the least recently data is move to L2 cache. The victim cache of L2 cache is a shared L3 cache with capacity of 2MB. When L2 cache is full, data is moved to L3 cache.

Before the upgrade of HECToR from Phase IIa to Phase IIb and the reduction of the size of the Cray XT4 system to the half, HECToR was ranked as the 20th fastest computer in the world at the list of the top 500 supercomputers [12] which is published twice a year with 174.08 TFlops maximum performance and a peak performance of 208.44 TFlops.

3.1.2 Cray XT6 system

This system is the new compute engine (Phase IIb) of HECToR' s facility. It became available to all users on the 14th of June 2010. This upgrade is the reason for reducing the Cray XT5h system from 60 to 33 cabinets. The Cray XT6 system is contained in 20 cabinets and consists of 1856 compute nodes. Each node comprises two AMD 2.1 GHz 12-core processors and giving a total of 44,544 cores. The theoretical peak performance of this system reaches the 373 Tflops.

According to the brochure of Cray about the XT6 system [13], each node shares 32 GB of DDR3, SDRAM main memory, and compared with the XT4 system, the available memory per core is now reduced to 1.33 GB. The memory per node is divided into four NUMA regions. The total memory of the Cray XT6 system is 58TB. The memory bandwidth per compute node reaches a peak of 85.3 GB/s and the memory bus is shared between the cores of each processor (12 cores are fighting for the same memory bus).

The arrangement of the nodes and the interconnection mechanism is the same as in the Cray XT4 system; the nodes are arranged on a 3-dimensional torus and the interconnection chip is the Cray SeaStar2+. This may causes some performance problems because six times more cores are trying to access the interconnect network, something that will be fixed on the next upgrade of HECToR by replacing the interconnection chip.

The cache layout of each core of the processor is almost the same with the cache memory of the processors of the Cray XT4 system. According to [13], the main difference in the cache of the 12-core processors is the increase in size of the Level 3 cache to 12 MB.

On June 2010, Top500 supercomputer sites released the most recent list of the top 500 fastest supercomputers in the world [14]. HECToR XT6 system was ranked as

the 16th fastest supercomputer with 274.70 TFlops maximum performance and a peak performance of 366.74 TFlops.

	Cray XT4	**Cray XT6**
Processor	AMD 2.3 GHz quad-core Opteron	AMD 2.1 GHz 12-core Opteron
Processors per node	1	2
Cores per node	4	24
Total cores	11,832	44,544
Memory per node	8GB	32GB
Memory per core	2GB	1.33GB
Total memory	24.4 TB	58TB
Bandwidth to local memory	12.8 GB/s	85.3 GB/s
Cache Memory		
Level 1	64KB data, 64KB instruction dedicated per core	64KB data, 64KB instruction dedicated per core
Level 2	512KB data and instruction dedicated per core	512KB data and instruction dedicated per core
Level 3	2MB shared to 4 cores	12 MB shared to 24 cores
Interconnect		
Chip	Cray Seastar2+	Cray Seastar2+
Arrangement	3D torus	3D torus

Table 3.1: Comparison of Cray XT4 and Cray XT6 systems

3.2 HECToR software

Both Cray XT4 and Cray XT6 systems use Cray Linux Environment. This environment works on a Linux-based operating system. SuSE Linux is used for the service nodes and Compute Node Linux (CNL) is used for the compute nodes. CNL is a very lightweight operating system and it is designed to minimise the OS effects in order to increase the parallel performance of the system.

CLE is a suite of high performance software. This software is managed using the module environment. This environment is also used for managing several application packages that are installed on HECToR. The software that is provided by CLE is the same on both systems, however the version of some programs may differ across these systems. There is a variety of compilers and different versions of theses compilers available on

HECToR. PGI, Pathscale, Cray and GNU compilers can be used by the programmer on both machines. Furthermore, CLE comes with some performance analysis tools, such as Cray Performance Analysis Tool (CrayPAT) and Apprentice2, that will be used for the investigation of mixed mode programming on HECToR.

Moreover, HECToR makes available a number of numerical libraries. Some of these libraries are needed for compiling and running the benchmark codes that are used for investigating mixed mode programming on HECToR. The software that is used for running and investigating the performance of each benchmark is cited at the description of the experimental procedure of every benchmark code.

3.3 Benchmarks

Almost all the mixed mode MPI/OpenMP codes that are used on this project for investigating mixed mode programming on HECToR are selected from benchmark suites which are supplied by well-established organisations. The description of these benchmarks is categorised based on the organisation that provides these codes. At the end of this section, the code that implements the Jacobi relaxation algorithm is described which was created in the scope of Michal Piotrowski's dissertation [15].

3.3.1 DEISA benchmarks

DEISA (Distributed European Infrastructure for Supercomputing Application) is a consortium of leading national supercomputing centres [16]. DEISA provide a suite of benchmarks from various scientific sectors to be used for quantifying the performance of modern HPC systems. Although the codes that are used in this project are not created by DEISA, they are packed in a framework where compilation, execution and analysis can be configured by XML files, making easier the running of the codes using different parameters (e.g. number of processors, number of threads per process, problem size).

CPMD

The CPMD (Car-Parrinello Molecular Dynamic) code comes from the materials science sector and it is created and licensed under the CPMD consortium [17]. "The CPMD code is a plane wave/pseudopotential implementation of Density Functional Theory, particularly designed for ab initio molecular dynamics" [16]. Using DEISA framework, the benchmark performs 10 steps of steepest descent energy minimization of a water box. Various box sizes and number of water molecules are available for testing different computation loads on HPC systems.

The CPMD is code is written in Fortran. The compiler that will be used for the compilation of the benchmark must support “Cray Pointer” style dynamic memory manage-

ment. Furthermore, the code must be linked with a library that contains BLAS/LAPACK linear algebra subroutines. Linking with an FFT library is optional as CPMD code comes with a portable and quite competitive FFT implementation. In order to be fairly portable, the CPMD code contains many optional parts, which can be used by defining the proper preprocessor flags. The default version of CPMD benchmark is serial. In order the parallel version to be created, the proper preprocessor flag must be defined. The building environment that was used for running this code on HECToR is presented in the description of the experimental procedure of this benchmark.

BQCD

Berlin Quantum ChromoDynamics (BQCD) is a program that is used for performing lattice QCD simulations with dynamic Wilson fermions. According to [18] and [19], the algorithm that is employed in this program to perform QCD simulations is the Hybrid Monte Carlo (HMC). An iterative solver of large system of linear equations with even-odd pre-conditioning is the main kernel of HMC programs. In BQCD, standard conjugate gradient solver is used in this kernel and most of the execution time is spent on this part of the program (about the 80% or 95% depending on the parameters of the program).

The simulations are performed in a 4-dimensional lattice with periodic boundaries and as it is mentioned in [16], the parallelization is achieved by performing regular grid decomposition in the highest three dimensions. DEISA framework provides two lattice sizes to run the program under various computation loads; a 24^3x48 lattice and a 48^3x96 lattice. Furthermore, the programmer can explicitly set the number of MPI processes that will be used on each lattice dimension.

The program is written in Fortran. According to [19], the main reason for selecting this programming language (except for the obvious reason that the developer was a Fortran programmer) is the ability of Fortran to handle complex arithmetic (complex arithmetic support was added in C after the start of BQCD implementation). BQCD uses a C preprocessor (which was employed from the beginning) for conditional compilation and macro preprocessing. BQCD comes with various preprocessor flags for different machine architectures and compilers.

3.3.2 NAS Parallel Benchmarks

NASA advanced supercomputing division [20] has developed a benchmark suite that is used for the performance evaluation of modern high performance computing systems. Three mixed mode simulated CFD application benchmarks are contained in the version 3.3 of multi-zone NAS Parallel Benchmarks (NPB). “The application benchmarks Lower-Upper Symmetric Gauss-Seidel (LU), Scalar Penta-diagonal (SP), and Block Tri-diagonal (BT) solve discretized version of the unsteady, compressible Navier-Stokes equations in three spatial dimensions”[21]. From those benchmarks, only SP is used in

this project. This decision was based on some of the characteristics of the other two benchmarks. In BT, the mesh is partitioned in uneven zones which are split across the MPI processes, and this unusual division was the reason that this benchmark code was not included in the project. Furthermore, LU was not used by this project due to the fact that this benchmark cannot run using more than sixteen MPI processes which limits the investigation process.

Scalar Penta-diagonal benchmark

This benchmark operates on a 3-dimensional structured mesh with periodic boundary conditions in the two horizontal dimensions (*x* and *y*). This mesh is divided into zones with approximately the same size which form a 2-dimensional tiling (Figure 3.1). These zones are distributed across the MPI processes at the start of the program. The boundary values of the zones are exchange at every time step.

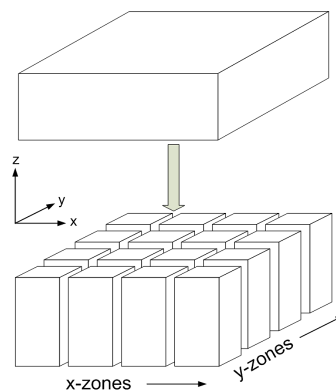

Figure 3.1: Two-dimensional tiling of three-dimensional mesh, reproduced from [21].

SP, like all NPBs, categorizes the size of the mesh into eight classes; S, W and A through F with memory requirements from 1MB up to 5 TB, providing the capability of evaluating the computational performance of modern supercomputers under a variety of work loads. The class of the mesh that will be used by the program is defined during the compilation time. The number of MPI processes of the program are also defined at the build of the program. These values are passed as argument to the makefile that builds the program.

The program is implemented in Fortran. However some C source files exist in the benchmark suite which are common for all the multi-zone NAS parallel benchmarks and their use is limited in setting the parameters of the program (e.g. mesh size, number of zones, number of processes) based on the values that are passed during the building process.

3.3.3 ASC Sequoia benchmarks

The Advanced Simulation and Computing department of Lawrence Livermore National Laboratory in California provides a collection of benchmarks which was initially developed for measuring the performance of Sequoia, one of their computer resources [22]. Among these benchmarks, only some of them are mixed-mode codes. The most complete mixed-mode codes are the UMT and IRS benchmark codes. They are both single physics package codes. However, UMT relies on a package called pyMPI which integrates MPI into the Python interpreter [23]. Patrick Miller in [24] reports that pyMPI is more heavy weight than traditional C or Fortran MPI, and in general Python scripts are slower than C and Fortran. For these reasons, it was decided not to include UMT benchmark in this project.

IRS

IRS stands for Implicit Radiation Solver. This code solves the radiation diffusion equation on a 3-dimensional structured mesh. Preconditioned conjugate gradient (PCCG) method is used in IRS for inverting a matrix equation, and the matrix pre-conditioners can be one of the following; no pre-conditioner, diagonal scaling and two- step Jacobi. The size of the mesh is 10x10x10 cm^3 which is fixed and it is divided into domains. These domains are distributed across the processors. Although the size of the mesh is fixed, the size of the problem can become larger by refining the computational mesh. IRS comes with a test deck where the suggested resolution is 25x25x25 cells per domain. The number of processors that can be used by the program should be integers cubed (1, 8, 27, 64, 125, 216, ...) and this number should be greater than the number of the physical domains.

The number of processors, the number of domains and the number of zones per domain side can be set by the programmer and are passed as command-line arguments to the program. The main benchmarking results of the program is the time per zone-iteration, which is measured in microseconds, and the figure of merit (FOM) which is the speed per cell-iteration in one domain.

IRS is implemented in C. The build of the program is performed by the combination of Perl scripts and the make tool. MPI is used for exchanging data that lie on the surfaces of the domains. OpenMP is used for parallelising main loops of the code. In fact, only three OpenMP directives (two *parallel for* directives and a *critical* directive) are used in the code which are contained in different header files. These header files are included when these directives are needed.

3.3.4 Jacobi kernel benchmark

This benchmark code was designed and implemented as part of [15]. The benchmark is based on the iterative Jacobi relaxation method. The algorithm that is used in the

program is a modified version of the reverse edge-detection algorithm on 3 dimensions. The implementation of this algorithm contains point-to-point and collective communications which are monitored in order to investigate how these communications are affected by mixed mode programming in comparison with the pure MPI implementation of the benchmark.

This benchmark is written in C, and four different versions of the same program are implemented; a pure MPI version and three mixed mode versions, one for each mixed mode scheme (master only, funnelled and multiple). All versions of the algorithm use 3-dimensional static arrays. The algorithm requires the array that holds the egdes to be distributed across the processes. Thus, each process keeps an array of fixed size and the total size of the problem is increased with the number of the MPI processes. The size of the dimensions of the edge array is specified by the distribution of the processes across the three dimensions (Figure 3.2).

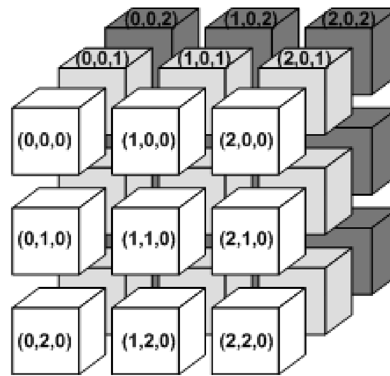

Figure 3.2: MPI Distribution, reproduced from [15].

In Figure 3.2, each cube represents the a 3-dimensional sub-array that is part of the edge array which is distributed across the processes. In the mixed mode versions of the algorithm, this sub-array is further distributed across OpenMP threads in 1-dimensional decomposition manner.

The build of every version of the algorithm is performed using the supplied make file (one file per version). All the versions of the benchmark have to be run with specific command-line arguments. These parameters are: the number of processes, the desired tolerance, the maximum number of iterations and the distribution of the MPI processes across the three dimensions.

Chapter 4

Results and Analysis

The benchmark codes that are described in the previous chapter were ported and run on both systems of HECToR. The resulting performance is described and analysed separately for each benchmark and each system of HECToR.

4.1 Experimental procedure

The performance of mixed mode programming is evaluated by comparing the results of mixed mode codes with the results of their pure MPI equivalents. The notion behind this procedure is to keep the total number of tasks (MPI processes and OpenMP tasks) constant in order to have the same level of parallelism in all codes (pure MPI and mixed mode) of the benchmark and try various combinations of OpenMP threads per MPI process in order to reduce the number of MPI processes per node and replace them with OpenMP threads. The total number of tasks is equal to the number of processors that are used for running each benchmark code. In fact, both pure MPI and mixed mode programs are produced by the same mixed mode program which is run using one OpenMP thread per MPI process for the pure MPI version.

On the Cray XT4 system where each node has four cores, only three combinations of threads per process can be used; one thread per process which represents the pure MPI version of the program, two threads per process and four threads per process. On the Cray XT6 system, the presence of 24 cores per node offers the opportunity of experimenting with more combinations of threads per process which are presented in Table 4.1.

However, a node in Phase IIb system is logically divided into four NUMA regions on which the MPI processes are placed by the scheduler. By default, the scheduler places the first six processes on the first NUMA region, the next six on the second NUMA region and so on. When the number of processes are less that the cores of the node then a slight imbalance is introduced between the tasks of the first regions and the subsequent ones. So, on the Cray XT6, *aprun* command can be called with

processes/node	threads/process
24	1
12	2
8	3
6	4
4	6
3	8
2	12
1	24

Table 4.1: Combinations of OpenMP threads per MPI process on Cray XT6 system.

an extra parameter which defines how many MPI processes should be place on every NUMA region and ensures an even distribution of MPI processes across the NUMA regions. Several runs of various benchmark codes using the same number of processes and threads were performed with and without this parameter (-S). The results of theses runs showed that -S ensures similar performance between successive runs. So, when the number of MPI processes of a run can be divided equally by the number of NUMA regions, application launcher *aprun* is called with -S parameter.

On the Cray XT4 system the module *xtpe-barcelona* is always loaded before the built of any benchmark code. The equivalent module on the Cray XT6 system is *xtpe-mc12*. These modules set the proper compiler options in order to pass information to the compiler about the target processor. The compiler uses the specification of the processor to perform better optimisations and produce a more target specific executable.

Furthermore, all benchmarks were run using the parallel job runner *aprun*. On both systems, the total number of MPI processes, the number of MPI processes per node and the number of threads per process are passed to the job runner. On the Cray XT6 system, the task affinity on the NUMA regions was sometimes specified to the aprun launcher.

Two performance analysis tools are used for understanding the performance of mixed mode programming, CrayPAT and Scalasca. Unfortunately, the time that this report was written Scalasca was available only on the Cray XT4 system.

Moreover, the change of the memory usage per node is monitored for every benchmark code on both systems of HECToR. CrayPAT measures the average memory "high water mark" per process. So, the memory usage per node is the multiplication of this value with the number of processes per node. The size of the distributed data per node is the same for all the versions of a program for specific problem size and processor count. Thus, the memory usage per node is only affected by the additional memory needed by each process to perform various tasks, such as packing and unpacking the messages for halo swapping.

4.2 CPMD

The benchmark was built and run on both systems using the provided framework from DEISA and the process was configured by setting the desired parameters in the corresponding XML files. ACML library was linked to the program in order to provide the required BLAS/LAPACK linear algebra routines.This library (4.2.0 version on XT4 and 4.3.0 XT6) became available to the program by loading the corresponding module in HECToR (*acml*) and passing the corresponding option to the linker. PGI compiler (version 9.0.4 on XT4 and version 10.3.0 on XT6) was used for building the benchmark and the following compiler options were used on both Cray XT4 and Cray XT6 systems:

- **-r8**: This option forces the compiler to treat REAL variables as double precision numbers.
- **-pc=64**: This option sets the precision of the floating point stack to double precision.
- **-Msignextend**: This option makes the compiler to extend the sign bit when converting longer to shorter integers.
- **-Bstatic**: This option is passed to the linker in order to perform static binding.
- **-fast**: This an optimisation flag. This option implies that more optimisation flags are used which are optimal for the target platform. The optimisation concerns the serial execution of the program. The optimisation involves the generation of scalar SSE code with xmm registers, O2 optimisation level, vector pipelining and alignment of long objects on cache-line boundaries.
- **-mp**: This option enables the recognition of OpenMP directives by the compiler. On Cray XT4 this option is followed by the *nonuma* parameter which advices the compiler not to use libraries to give affinity between threads and processors because this is useful on NUMA architectures which is not the case on this system.

Due to the size of the CPMD code (about 250,000 lines), it is very difficult to understand the functionality of the code at a very low level. For this reason, the use of performance analysis tools is the only way to understand the resulting performance. However, Scalasca never worked with PGI compiler, so only CrayPat was used.

4.2.1 CPMD on Cray XT4 system

The first runs of the CPMD code on this system of HECToR were performed using 64 cores (16 nodes). The real space grid (problem size) is determined by the number of the water molecules and the cut off for the plane wave basis in Rydberg. This size varies between 20 (small system, low cutoff) and 300 (large system, high cutoff) and is printed in the output file The cut off value was the same for every number of water molecules

and the used value was 70. The problem size was changed by altering the number of the water molecules that were used in the program.

The investigation of mixed mode programming on 64 processors started with one molecule of water. The real space mesh size of this run is 108x108x108. Figure 4.1 shows that the mixed mode versions perform better than the pure MPI version of the CPMD code, for this problem size and processor count. In fact, as the MPI processes are being reduced and replaced with OpenMP threads, the execution time is also reducing. The best performance is achieved when four threads per process are used where the execution time of the program is reduced by almost 60%.

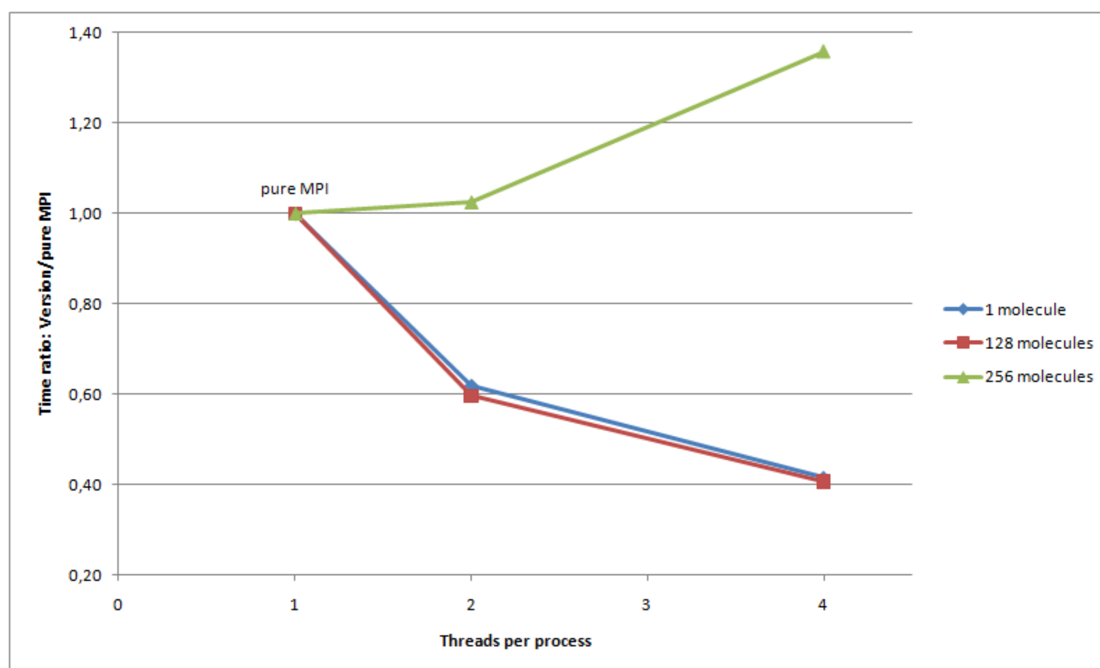

Figure 4.1: CPMD execution time ratio of mixed mode versions to pure MPI for 1, 128 and 256 molecules, using 64 processors on Cray XT4.

The output of the CPMD program contains very useful statistics about the CPU time and memory usage which are an indication of where the execution time is spent on. The output of the pure MPI version shows that about the 37% of the execution time is spent on all-to-all communications. However, the size of the data involved in all-to-all communications is about 176 KB and the time needed for transferring this data results in a very low bandwidth of about 85 MB/s. The manual of the CPMD code[17] mentions that the retrieval of these statistics is highly platform dependant and the output may be unreliable. So, CrayPAT is recruited in order to confirm these figures.

The results of CrayPAT are produced from an instrumented executable which means that these results contain some profiling overheads. The profile of the program shows that a great portion of the execution time is indeed spent in all-to-all communications and according to the report of CrayPAT, this portion is about the 34% of the total time. In total, the execution of MPI functions takes about the 52% of the execution time.

CrayPAT can detect load imbalance in a program by measuring the time that faster processes are waiting outside a collective communication for the slower ones to arrive. When a program is instrumented to trace MPI functions, this type of load imbalance is automatically measured by timing a barrier that is called before entering the collective communication and it is introduced to the program during the instrumentation. This time is presented in the profile as MPI_SYNC. For all-to-all communications, this synchronisation time is the 12% of the execution time of the pure MPI program and in total the 21% is spent in synchronisation before entering into collective routines (Table 4.2).

With a careful look to the profile produced by CrayPAT, the load imbalance is apparent in all functions of the program. The process with the lowest load of all executes the most time consuming function of the program (fftstp_) three times faster that the process with the heaviest load. This confirms the claim of the manual that "the most important bottleneck in the distributed memory parallelization of CPMD is the load-balancing problem in the FFT"[17].

The load balance is improved as the MPI processes are replaced by OpenMP threads and when the program is run using four threads per process it is almost eliminated. With four threads per process, the portion of the time spent on synchronisation before entering collective communications (which is indication of load imbalance) is reduced from 21% to 6.8% of the execution time. Furthermore, the variation of execution time of traced functions among processes now ranges from 1.0% to 7.0% whereas in the case of pure MPI version the range of this figure is 18% to almost 30%.

Function group	**pure MPI**		**2 threads/process**		**4 threads/process**	
USER	14.17	43%	13.19	48%	14.24	60%
MPI	10.49	32%	7.46	27%	5.88	25%
MPI_SYNC	6.86	21%	5.17	19%	1.62	6.8%
Total Time	33.00		27.53		23.65	

Table 4.2: Statistics produced by CrayPAT for runs using 64 processors and 1 molecule problem size (time in seconds and percentage).

It is shown that mixed mode programming cures the inherent load imbalance of this program for this problem size. By reducing the processes, a coarser distribution of the real space mesh is achieved which reduces the load imbalance and results in a noticeable reduce of the execution time of the most time consuming functions. Furthermore, as collective communications seem to be the bottleneck of the performance of the application, mixed mode programming reduces the number of processes that participate in these communications.

For 128 molecules problem size and above, DEISA scripts provide the option to enable a technique which is implemented in the CPMD code and helps the program to achieve additional speed up beyond a few hundreds of processors. This technique is based on the use of groups of tasks and the parallelization is performed not only across data but also across tasks. The programmer can define the number of task groups that the program should use in the DEISA scripts. After experimenting with various numbers of

task groups, the program performed better with four task groups. Furthermore, because of the significant increase of the problem size, the execution of the program is now limited to 10 iterations in order to reduce the execution time and save HPC resources.

When the CPMD benchmark is run for 128 molecules of water on 64 processors, mixed mode programming provides similar performance improvement (Figure 4.1). The size of the real space mesh now is 100x200x200 (increased by 70%). The load imbalance of the pure MPI version is reduced as a larger data set is divided into the same number of processes (coarser load distribution) and now only the 5.5% of the execution time is spent on MPI_SYNC (the percentage for one molecule problem size was 21%, Table 4.3). However, the output of mixed mode version of CPMD program using two threads per process shows that the execution time of almost all subroutines is reduced along with the time spent on collective communications and the overall performance of the program is improved by 40%.

Function group	**pure MPI**		**2 threads/process**		**4 threads/process**	
USER	180.86	78%	184.85	75%	224.99	83%
MPI	37.33	16%	44.05	18%	33.21	12%
MPI_SYNC	12.75	5.5%	15.60	6.3%	9.17	3.4%
Total Time	232.34		246.80		270.80	

Table 4.3: Statistics produced by CrayPAT for runs using 64 processors and 128 molecules problem size (time in seconds and percentage).

The best performance for this problem size is achieved again by the mixed mode version that uses four threads per process where the total execution time is reduced by almost 60%. The statistics contained in the output of the program show that the time spent in collective communications is further reduced. Moreover, MPI_SYNC time that was measured by CrayPAT shows that load imbalance is reduced in comparison with the pure MPI version. However, this difference in MPI_SYNC time between pure MPI and mixed mode version is significantly smaller than running the CPMD program using one molecule of water (Table 4.4).

Version	**Collective communications**	
pure MPI	22.88 seconds	11%
2 threads/process	18.38 seconds	15%
4 threads/process	13.75 seconds	17%

Table 4.4: Statistics produced by CPMD for collective communications on runs using 64 processors and 128 molecules problem size.

For one and 128 molecules problem size, both mixed mode versions of the program achieved the same improvement in performance. But this in not the case when the program is run for 256 molecules of water. The real space mesh of this problem size has dimensions 200x200x200 which is 50% larger than the real space mesh of 128 molecules. Experimenting with this problem size is very demanding in HPC resources

as the execution of the program for 10 iterations takes more than 15 minutes. This is the first time that the pure MPI version outperformed both mixed mode versions of the program. Pure MPI is about 3% faster than mixed mode version using two threads per process and about 27% faster than mixed mode version using four threads per process (Figure 4.1).

In order to understand the resulting performance, the pure MPI version and the mixed mode version with four threads per process were profiled. The produced profiles show that the portion of the execution time spent on MPI routines is reduced when mixed mode programming is applied but the actual time spent is increased (Table 4.5). Furthermore, there is less load imbalance in the mixed mode version than in pure MPI version which is appeared in the reduction of both MPI_SYNC time and imbalance times of each profiled function.

Function group	**pure MPI**		**4 threads/process**	
USER	508.25	81%	686.92	84%
MPI	70.96	11%	82.96	10%
MPI_SYNC	41.33	6.6%	39.89	4.9%
Total Time	622.66		814.06	

Table 4.5: Statistics produced by CrayPAT for runs using 64 processors and 256 molecules problem size (time in seconds and percentage).

This experiment revealed the weaknesses of mixed mode implementation of this program. With a careful look to the profiles of the various versions of the program for this problem size, it can be noticed that the execution time of some subroutines is significantly increased as the MPI processes are replaced with OpenMP threads. The reason is that overheads by the parts of the code which are executed sequentially are introduced to the execution of the program. It seems that the parts of the code that are executed in parallel by OpenMP threads are not adequate to replace the parallelism that is achieved by MPI processes. Thus, the work distribution among the threads is uneven with the master thread having always more work than the rest of the threads and in the case of four threads per process for 256 molecules problem size this work imbalance is severe.

These overheads are increased as the load per process is increased, and for 256 molecules, the reduce of load imbalance does not compensate any more the increase of these overheads. That is why pure MPI outperforms both mixed mode versions. Moreover, parallelising more parts of the code using OpenMP threads may have resulted in better performance for this problem size, but it may have affected the performance of the program for smaller problem sizes, because OpenMP overheads would have owned a larger portion of the execution time.

The experiments are continued using 256 processors, which involve four times more processors than the previous runs in order to investigate how the increase of processors affects the resulting performance of the program. The program was run for all the problem sizes that are provided by DEISA framework; 1, 128, 256, 384 and 512 molecules. The real space mesh size of the first three sizes is already mentioned in

the previous experiments. The real space mesh of 384 molecules problem size has dimensions 200x200x300 and the real space mesh of 512 molecules problem size has dimensions 200x200x400 (Table 4.6).

Problem size	Real space mesh
1 molecule	108x108x108
128 molecules	100x200x200
256 molecules	200x200x200
384 molecules	200x200x300
512 molecules	200x200x400

Table 4.6: The dimensions of the real space mesh of each problem size.

The resulting performance of the runs using different problem sizes is depicted in Figure 4.2. Mixed mode version using two threads per process still achieves better performance that the pure MPI version in almost every problem size, showing similar behaviour with the runs of the program using 64 processors. However, the main difference in the performance of the mixed mode versions using 64 processors and using 256 processors is that now mixed mode version using four threads per process always performs worse than mixed mode version using two threads per process.

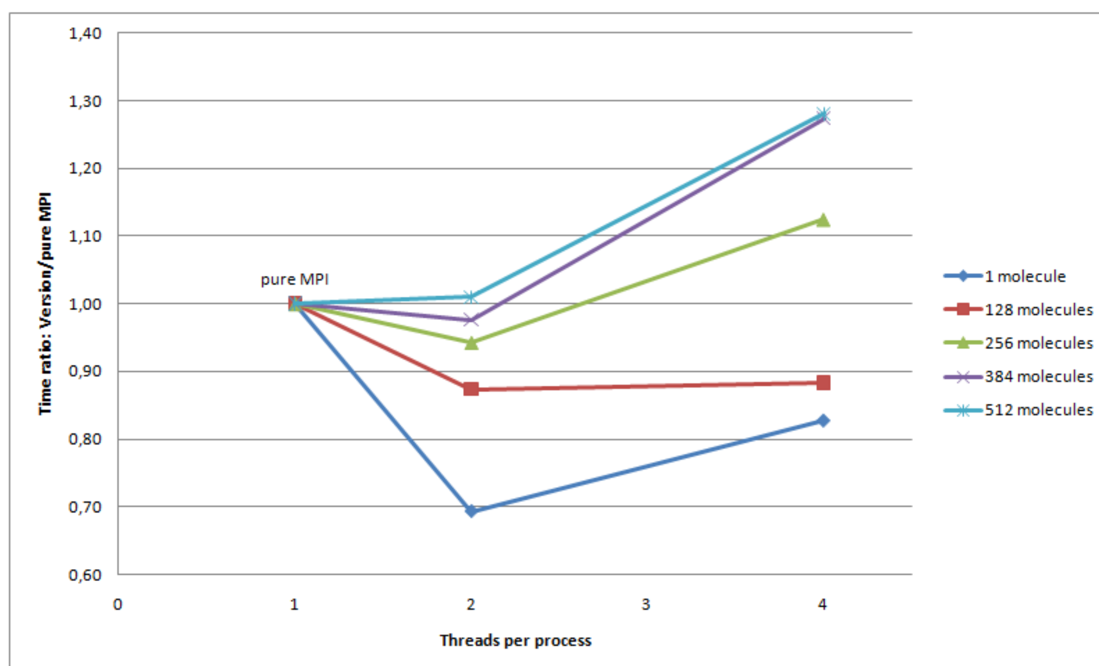

Figure 4.2: CPMD execution time ratio of mixed mode versions to pure MPI for 1, 128, 256, 384 and 512 molecules, using 256 processors on Cray XT4.

The profile of the execution of the program for 1 molecule problem size shows that the load imbalance across the MPI processes is severe. The 72% of the execution time of

pure MPI version is spent in MPI routines and the two thirds of this time comes from the load imbalance of the program (table 4.7). Similarly to the runs on 64 processors, the load imbalance is reduced by the use of mixed mode programming. However, in comparison with the runs using 64 processors, now the overheads that are introduced by the serial parts of the program are more significant and they are increasing as the number of threads and the work load are also being increased. This is the same behaviour that became visible for 64 processors only when the problem size was significantly increased, but in this case it is present even for one molecule problem size.

Function group	pure MPI		2 threads/process		4 threads/process	
USER	6.13	21%	5.22	26%	6.74	39%
MPI	8.24	29%	7.94	40%	6.11	35%
MPI_SYNC	12.45	43%	4.95	25%	2.95	17%
Total Time	28.58		19.77		17.42	

Table 4.7: Statistics produced by CrayPAT for runs using 256 processors and 1 molecule problem size (time in seconds and percentage).

When the problem size is increased (128 molecules), mixed mode programming is still beneficial for the performance of the program. Both mixed mode versions perform better than the equivalent pure MPI program. For problem size of 256 molecules and above, the overheads introduced by the sequential execution of parts of the code start to severely affect the performance of mixed mode versions. For 512 molecules, pure MPI again outperforms both mixed mode versions. The reason is that MPI processes are fully loaded and the imbalance across the processes is insignificant. Thus, the reduction of the number of MPI processes degrades the performance of the program. Furthermore, load balanced processes are replaced by load imbalanced threads and pure MPI program achieves better performance than its mixed mode equivalents.

4.2.2 CPMD on Cray XT6 system

The experimental procedure that was used for investigating mixed mode programming on Phase IIb machine of HECToR is similar with the one followed on the Cray XT4 system. At first, the CPMD code was run for 1, 128 and 256 molecules of water using 96 processors. Then, the experiments were performed using 384 processors (again four times more processors) for all the available problem sizes. As it is already mentioned, there are more mixed mode versions of the program on the Cray XT6 system and they are presented in Table 4.1.

At first, the runs for one molecule problem size were investigated (Figure 4.3). For this problem size, the mixed mode version using six threads per process achieves the best performance which is 80% faster than the equivalent pure MPI version. The profile produced by CrayPAT for the pure MPI version shows that the 92% of the time is spent on MPI subroutines and only the 7.2% is spent on USER subroutines. This was

partially expected due to the inherent load imbalance of the program for small problem sizes when a large number of processors is used. However, the MPI_SYNC time, which is an indication of load imbalance, was only the 11% of the execution time. The rest of the MPI time is spent on all-to-all communications.

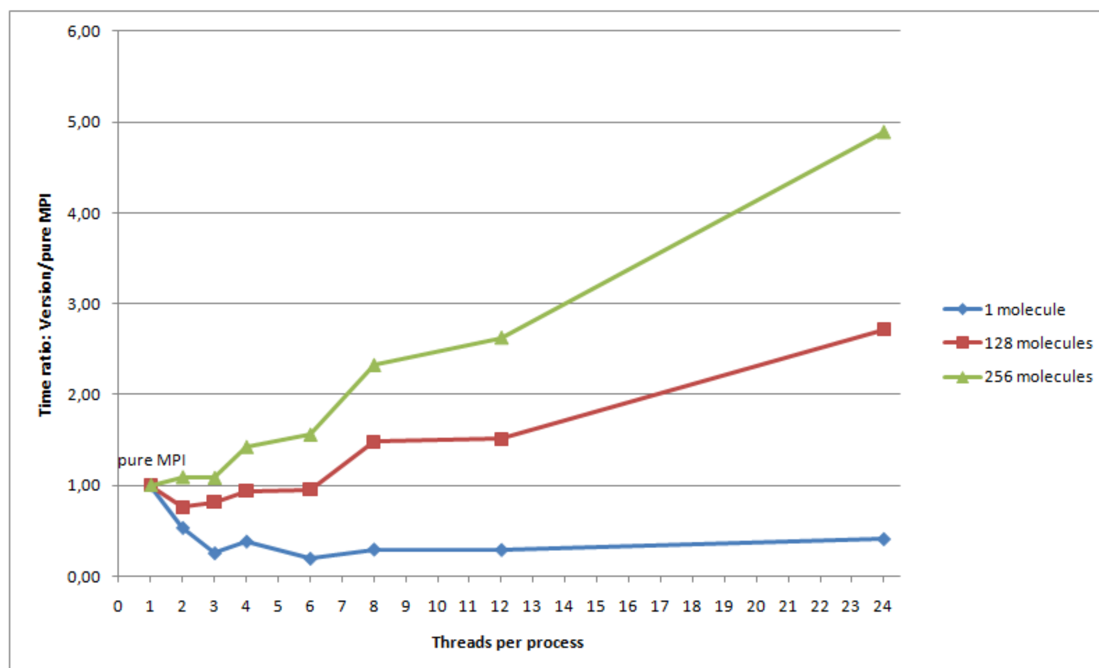

Figure 4.3: CPMD execution time ratio of mixed mode versions to pure MPI for 1, 128 and 256 molecules, using 96 processors on Cray XT6.

The comparison of this performance with the corresponding performance of the XT4 system is inevitable (Table 4.8).The same run using 33% less processors on the Cray XT4 system was more than four times faster (the execution time was reduced by 72 seconds) than the same run on the Cray XT6 system. The profiles of both runs show that the main cause of this bad performance is the time taken to perform all-to-all communications. For almost the same total message size, all-to-all communications last about 10 times more than they last on the Cray XT4 system although the number of processors is increased only by 33%. However, the time spent on the user functions is reduced to the half, as it was expected from the increase in the number of processors.

An explanation for the aggravating performance mainly lies on the interconnect chip of HECToR. The runs on 96 processors involve only four nodes which means that the communication pattern of the pure MPI version mainly involves intra-node communications. Furthermore, as it is already mentioned in the description of the hardware of HECToR, the same interconnect chip is used on both systems, but on the Cray XT6 system more cores are contenting for accessing it and this totally affects the performance of inter-node communications. In all-to-all communications, 24 processors are trying to access the network through the same chip simultaneously in order to send and receive data from processors of different nodes (or even from the same node, if the MPI imple-

Function group	pure MPI (XT4)		pure MPI (XT6)	
USER	14.17	43%	7.44	7.2%
MPI	10.49	32%	83.92	81%
MPI_SYNC	6.86	21%	11.33	11%
Total Time	32.99		103.44	

Table 4.8: Comparison of statistics produced by CrayPAT for pure MPI version with 1 molecule problem size run on Cray XT4 using 64 processors and Cray XT6 using 96 processors (time in seconds and percentage).

mentation does not replace intra-node communications with reads/writes to the shared memory). This contention is printed in the performance of all-to-all communications.

Although this feature of the Cray XT6 system influences all versions of the CPMD program, the performance is significantly improved when the number of the processes is reduced and more threads per process are used. The time spent on the routines of the program is slightly increased but the time spent on MPI routines is substantially reduced and becomes less than a quarter of the equivalent time of pure MPI version when 3 threads per process are used. Again, this time is mostly spent on all-to-all communications and MPI_SYNC time is also reduced. The work is almost evenly shared across the threads of a process when these threads execute user functions.

The execution time of the CPMD program is increased when the program is run using four threads per process. The same behaviour can be noticed for runs with different problem sizes and processor counts. There is a peculiarity about this particular run which may be the reason for the bad performance. There are six processes per node when four threads per process are used. These processes are not equally distributed to the four NUMA regions and the corresponding scheduler option cannot be used. So, the scheduler places all the processes on the first NUMA region (on the same six-core chip) and the threads of each process are placed on cores of different chip. The degradation of the performance is appeared when threads access shared data which may lie on different NUMA region where the access time is increased. This peculiarity applies for all the number of processes which cannot distributed evenly across the four NUMA regions (1, 2 and 3 processes per node).

The best performance is achieved by the six thread mixed mode version. The time spent on MPI routines is the least from all versions of the program for this problem size. The same is also applied to the load imbalance across the MPI processes. However, as the number of threads per process is increased, the work distribution becomes more unbalanced. The work is concentrated to the master threads. Furthermore, the overheads that are introduced by OpenMP start to aggravate the performance of the program (Table 4.9).

Every time the number of threads per process is increased, the load imbalance across the MPI processes is slightly reduced, but the work distribution across threads becomes more unbalanced resulting in poor performance compared with the rest mixed mode

Function group	pure MPI		3 threads/process		6 threads/process	
USER	7.44	7.2%	9.91	33%	11.86	48%
MPI	83.92	81%	15.39	52%	8.28	33%
MPI_SYNC	11.33	11%	2.62	8.9%	2.57	10%
OMP	- - -		1.46	5.0%	1.76	7.2%
Total Time	103.44		29.39		24.48	

Table 4.9: Statistics produced by CrayPAT for runs using 96 processors and 1 molecule problem size (time in seconds and percentage).

versions. In addition, the OpenMP overheads are increasing significantly. For example, the profile of the mixed mode version that uses 24 threads per process shows that the OpenMP overheads are more than the overheads produced by the load imbalance across the MPI processes. The time spent on the synchronisation of the threads is 7.28 seconds (14.2% of the execution time) and the MPI_SYNC time is 3.00 seconds (5.9% of the execution time).

Even for larger problem size (128 molecules), the main problem of pure MPI version (and of all mixed mode versions, but in smaller scale) is again the great amount of time which is spent on MPI routines and especially on all-to-all communication. In fact, none of the runs achieved the same performance with the equivalent runs on Cray XT4 using less processors. Furthermore, mixed mode programming improves significantly the performance of the program only when two threads are used which helps to reduce the MPI time from 133.16 seconds almost to the half (Table 4.10). When more threads per process are used, OpenMP overheads and the overheads introduced by the sequential execution of parts of the code aggravate the performance of the program and for 256 molecules, none of the mixed mode versions of the program perform better than the pure MPI equivalent.

Function group	pure MPI		2 threads/proc		3 threads/proc		6 threads/proc	
USER	91.78	37%	98.97	52%	111.36	61%	153.96	66%
MPI	133.17	54%	71.32	38%	56.76	31%	64.32	27%
MPI_SYNC	20.80	8.4%	16.73	8.8%	12.11	6.6%	12.99	5.5%
OMP	- - -		2.08	1.1%	2.10	1.2%	3.00	1.3%
Total Time	246.52		189.11		182.35		234.28	

Table 4.10: Statistics produced by CrayPAT for runs using 96 processors and 128 molecules problem size (time in seconds and percentage).

The investigation of mixed mode programming using the CPMD code continues with runs on 384 processors for all the problem sizes that are provided by DEISA (1, 128, 256, 384 and 512 molecules). The computational power is increased four times, but the performance of the program does not follow this trend for all the problem sizes (similarly to the runs on Cray XT4).

As it is shown in Figure 4.4, the best performance for one molecule problem size is

achieved by the mixed mode version using 12 threads per process (15.89 seconds) which is about three times better than the performance of the equivalent pure MPI version. At it was expected, the load balance is slightly worse for this number of processors and is always improving as the number of threads per process is increased (until 12 threads per process). The overheads for the serial parts of the code are kept at low levels, as the problem size is small and so is the work of each process. However, as the number of threads is increased, the OpenMP overheads introduced in the program are also increased and become greater than the overheads introduced by the load imbalance across the MPI processes.

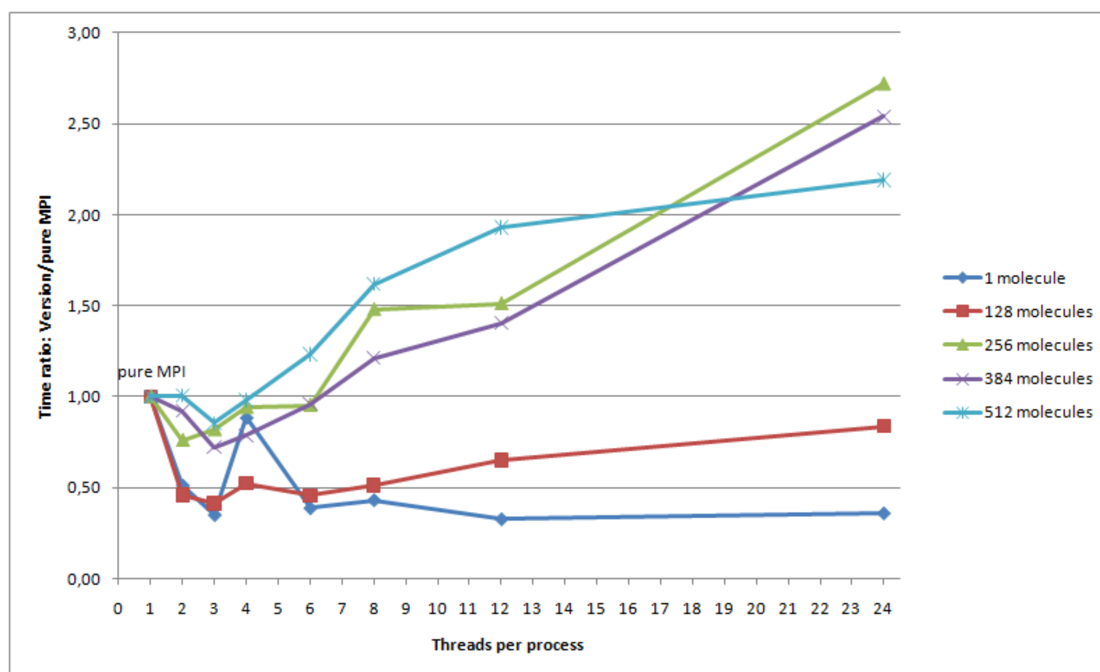

Figure 4.4: CPMD execution time ratio of mixed mode versions to pure MPI for 1, 128, 256, 384 and 512 molecules, using 384 processors on Cray XT6.

Function group	**pure MPI**		**3 threads/proc**		**6 threads/proc**		**12 threads/proc**	
USER	4.34	8.0%	3.90	18%	5.06	20%	6.02	34%
MPI	36.67	67%	12.47	56%	14.75	59%	7.67	43%
MPI_SYNC	12.69	23%	4.41	20%	3.78	15%	2.06	11%
OMP	- - -		1.29	5.9%	1.46	5.8%	2.15	12%
Total Time	54.55		22.09		25.07		17.91	

Table 4.11: Statistics produced by CrayPAT for runs using 384 processors and 1 molecule problem size (time in seconds and percentage).

When the problem size increases, the performance pattern is similar with the one produced on 96 processors. However, even with 512 molecules of water, the processes are not as loaded as when the program is run for 256 molecules using 96 processors, and

this is the reason why pure MPI version never outperforms the equivalent mixed mode versions. Furthermore, as it was also shown by the runs on the Cray XT4 system, as the number of processors and the problem size are increasing, the work imbalance across the threads is also increasing, making the mixed mode version using three threads per process the most balanced combination for delivering good performance.

After investigating and analysing the performance of the CPMD benchmark code on both systems of HECToR, the reasons for the resulting performance became apparent. Mixed mode programming improved the performance of the program by reducing the load imbalance across the MPI processes. However, the processes cannot entirely be replaced by threads, because there is not an extensive use of OpenMP parallelism. Especially for large problem sizes and small or moderate number of processors where the load imbalance is insignificant, the processes are fully loaded and the lack of shared memory parallelism causes the overload of the master thread and the uneven work distribution across the threads of a process. Furthermore, mixed mode programming reduced the memory needs of the program on both systems of HECToR.

Moreover, mixed mode programming improved the performance of all-to-all communications on Phase IIb machine of HECToR, as the decrease in the number of MPI processes per node reduced the contention of the processes for accessing the interconnect chip. This problem is the main reason that the achieved performance on Phase IIa machine was almost always better than the corresponding performance on Phase IIb machine even though the runs on the first machine were performed using fewer processors.

The gains in performance from mixed mode programming were higher when the CPMD program was run for small problem sizes on large processor counts. However, the profiles of the pure MPI version of these runs show that the most of the execution time was spent on the MPI communications of the program, which is an indication that the number of the used processors is excessive for this problem size and the scalability of the program has been exhausted. So, the gains of these runs are of little relevance because, for real simulations, the runs would be performed on smaller processor counts for these problem sizes.

4.2.3 Memory usage of CPMD

Except for the performance, mixed mode programming also affected the memory usage of the program on both systems. The measured memory "high water mark" of each process shows that the memory used process is increased as more threads per process are used. However, the memory per node and the total memory used during the execution of the program is significantly reduced especially on the Cray XT6 system (Figure 4.5). This is very important for cases where the needed memory per process is smaller than the available memory per core.

Basic data and many matrices are replicated on all process. So, the reduce of the number of processes also reduces the additional memory needed for the replication of data.

When only one process per node is used, the memory usage per node on the Cray XT6 system is about 4.7 times lower than the memory usage of the pure MPI version of the program. The major improvement is noticed for small thread counts. For more than six threads per process, the gains in memory from mixed mode programming are low.

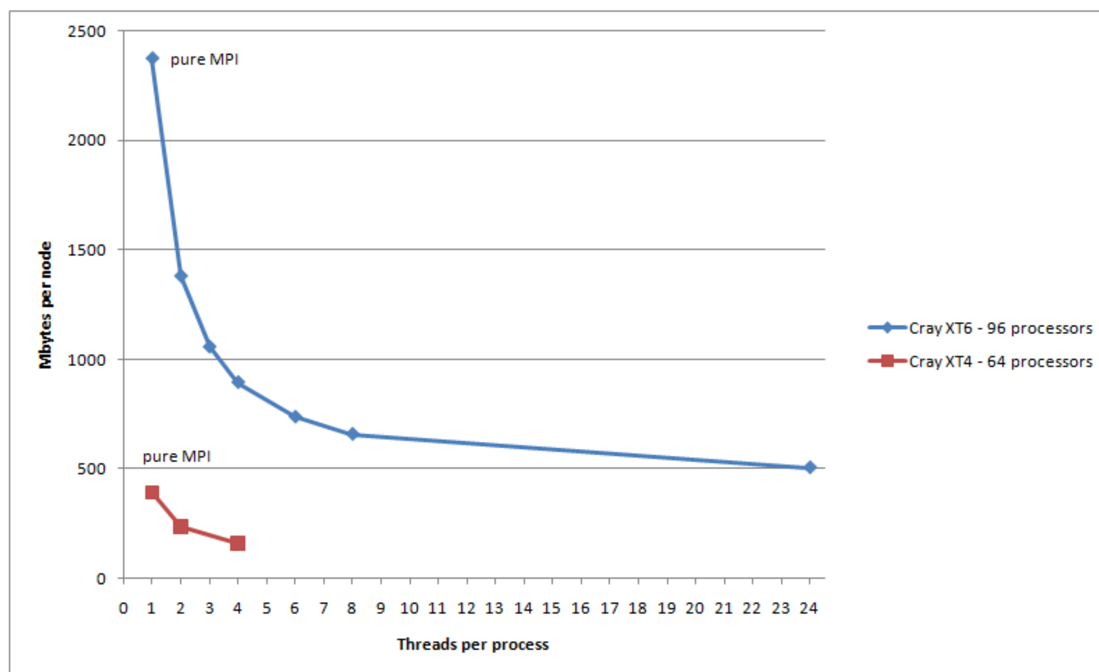

Figure 4.5: Memory usage of CPMD program for 1 molecule problem size using 64 processors on the Cray XT4 system and 96 processors on the Cray XT6 system.

4.3 BQCD

The build of this code does not need any prerequisite library (except for the MPI and OpenMP libraries). The automated build environment provided by DEISA was used in both the Cray XT4 and Cray XT6 systems. Actually, DEISA does not provide a pre-configured build environment for the Cray XT6 system. However, the great similarity between these two systems made the configuration for the Cray XT6 system a rather easy task. GNU compiler (version 4.4.2 on the Cray XT4 system and version 4.4.3 on the Cray XT6 system) was used for the build of the code because it is recommended in the manual[19] and it is the most efficient of all the compilers that produce a working executable. The following compiler flags were used on both the Cray XT4 and Cray XT6 systems:

- **-fno-range-check**: This flag disables range checking on results of simplification of constant expressions during compilation.
- **-nodef**: This flag forces the compiler not to predefine any system-specific or

GCC-specific macros.

- **-nostdinc**: This flag prompts the compiler not to search the standard system directories for header files and only the directories that are specified with -I options (and the directory of the current file, if appropriate) are searched.
- **-cpp**: This option enables preprocessing.
- **-O3**: This flags turns on all the supported optimisations of the compiler. It turns on a variety of compiler optimisation flags and their goal is to improve the performance of the serially executed program.
- **-fopenmp**: This option enables the OpenMP extensions and links the OpenMP library to the executable.

In order to keep the execution time of the program in short, the steps of Hybrid Monte Carlo algorithm are set to five and the maximum iterations of the solver are set to 10 for both problem sizes.

4.3.1 BQCD on Cray XT4 system

The runs of this benchmark code were performed using two different problem sizes. The problem size is determined by the dimensions of the lattice on which the QCD simulation is performed. The small lattice has dimensions 24x24x24x48 and 64, 256 and 1024 processors were used for running this benchmark with this lattice size. Figure 4.6 shows that the differences in the performance of the various versions of the program are minor. The execution of the program was profiled in order to understand how mixed mode programming affects certain characteristics of the program.

The profile of the pure MPI program which was run on 64 processors shows that most of the execution time is spent on program routines (Table 4.12). The time measured on MPI routines is mainly spent on point-to-point communications (both synchronous and asynchronous) and a small portion of this time is spent on all-to-all routine. Most of the messages that are exchanged have size between 256B and 64KB. However, the larger portion of the data is exchanged in messages of size between 64KB and 1MB. The MPI_SYNC time that was measured by CrayPAT is negligible, so there is no sign of noticeable load imbalance.

Function group	**pure MPI**		**2 threads/process**		**4 threads/process**	
USER	98.33	90%	102.88	92%	104.35	92%
MPI	9.37	8.6%	6.43	5.8%	7.07	6.2%
MPI_SYNC	1.56	1.4%	2.00	1.8%	2.44	2.1%
Total Time	109.29		111.37		113.91	

Table 4.12: Statistics produced by CrayPAT for runs using 64 processors and 24^3x48 lattice size (time in seconds and percentage).

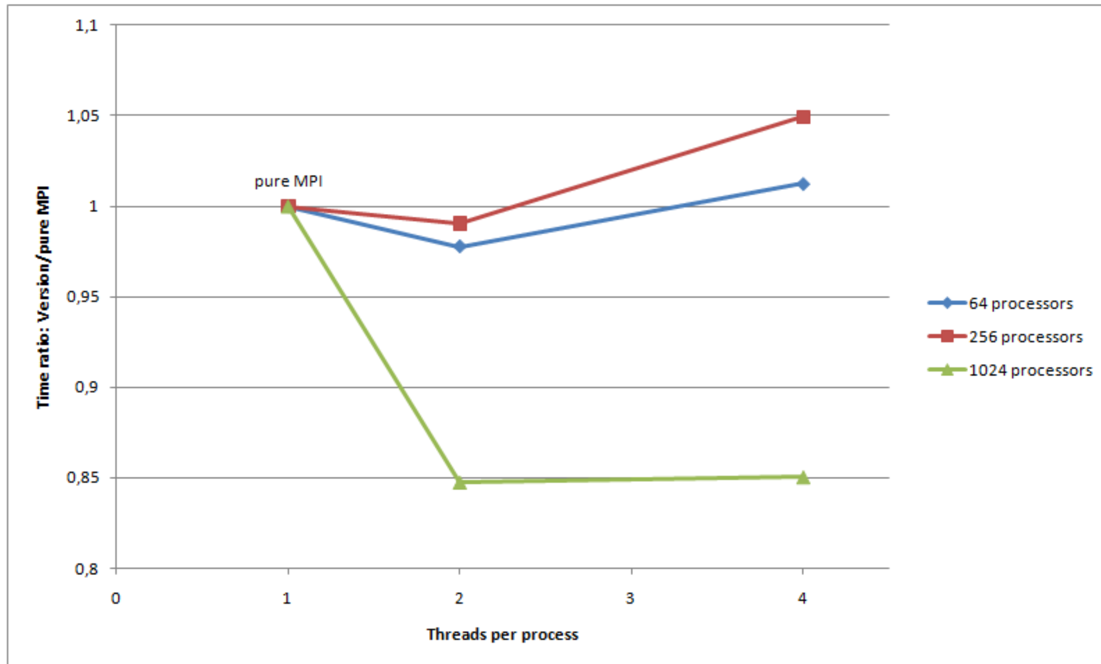

Figure 4.6: BQCD execution time ratio of mixed mode versions to pure MPI for 64, 256 and 1024 processors on 24^3x48 lattice.

After reducing the MPI processes to 32 and using two threads per process, the performance of the program was slightly improved. The main reason for this improvement is the reduce by one third of the time spent on MPI routines. Unfortunately, when CrayPAT is used with GNU compiler, it cannot produce an instrumented executable that measures accurately the overheads produced by the OpenMP library or by the lack of OpenMP parallelism.

The worst performance was achieved by the mixed mode version using four threads per process. Although, the MPI time is less than the equivalent time of the pure MPI version (but more than the MPI time of two threaded version), the execution time of program routines is the greatest of the three versions. The overheads introduced to the execution of the program routines are greater than the gains from the reduced communication time causing the bad performance of the program. In addition, the synchronisation time outside of all-to-all routines is increased by a second from the equivalent time of the pure MPI version. The next step of understanding the performance of the program is to identify why mixed mode programming improves point-to-point communications.

From the statistics about the sent messages of each program, it is shown that every time the number of MPI processes is halved, the average number of messages that each process sends is also halved. However, the total data that each process sends/receives is increased (Table 4.13). Many small messages are replaced by fewer but larger messages. It seems that in this way higher communication bandwidth is achieved which beneficial for the performance of the program. Among the three tested versions, mixed mode version using two threads per process is appeared to be the most balanced because

although more data per process is transferred, the MPI communications are completed faster than in the case of the pure MPI version, and the total message size per process is quite smaller than in the case of four threads mixed mode version.

Threads/process	Msg Count	Total Msg Size	Average Msg Size	Total Msg Size/node
pure MPI	50,305	1.17GB	24.48KB	4.68GB
2 threads	21,144	1.48GB	73.30KB	2.96GB
4 threads	9,544	2.14GB	235.20KB	2.14GB

Table 4.13: Average sent message statistics per process for 24^3x48 lattice size, using 64 processors.

The change of the performance of the program is not only a result of the replacement of the MPI processes by OpenMP threads. Along with the number of MPI processes, the decomposition of the lattice across the processes is changed. There is not only one decomposition for a specific number of processes. Almost every possible decomposition was tried for the pure MPI and mixed mode versions using 64 processors in order to find the more efficient one. In Figure 4.7, there are two different decompositions for each version and the more efficient is used for investigating mixed mode programming. This figure shows how the decomposition of the lattice affects the performance of the program.

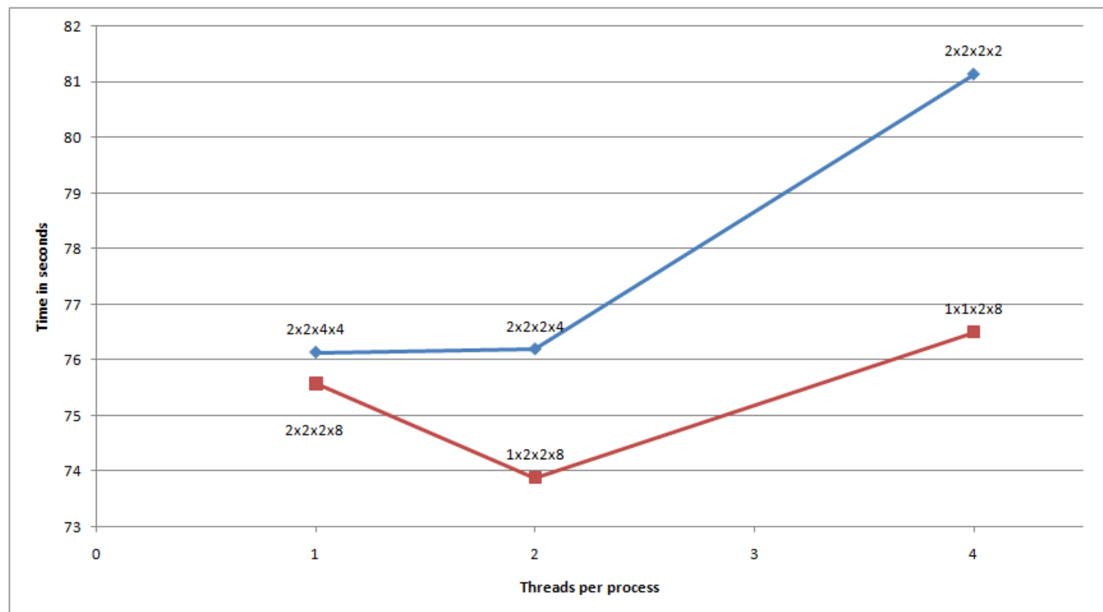

Figure 4.7: Performance of two different processor grids of BQCD benchmark using 64 processors on 24^3x48 lattice.

Investigating the decomposition that achieves the best performance is not in the scope of this project. However, the reduction of MPI processes of mixed mode versions affects the number of processes of each dimension of the lattice. The performance of mixed

mode programming is highly influenced by the decomposition of the lattice across the processes and especially by the dimension of the lattice in which the processes are reduced.

Experimenting with various lattice decompositions revealed that the difference in performance relies on the time spent on MPI routines. By changing the shape of the process grid, the communication pattern is also changed. The parameters that are affected are the number of the transferred messages, the size of the messages and the total size of the exchanged data. The performance of the program depends on how efficiently the MPI library and the underlying hardware can implement the communication pattern of a particular decomposition. In the case of 64 processors that is previously described, it seems that the communication pattern of 2 threaded mixed mode version is better suited on the Cray XT4 system than the other two. The process grids that are used for this program are presented in Table 4.14.

Threads/process	64 processors	256 processors	1024 processors
pure MPI	2x2x2x8	1x4x4x16	1x8x8x16
2 threads	1x2x2x8	1x2x4x16	1x4x8x16
4 threads	1x1x2x8	1x2x2x16	1x4x4x16

Table 4.14: The dimensions of the process grid of 64, 256 and 1024 processors.

When the program is run on 256 processors, its performance follows similar trend with the case of 64 processors. However, mixed mode programming affects differently the communication pattern of the program. Using the processor grids presented in Table 4.14, mixed mode programming does not change the number of the transferred messages but only the size of these messages and therefore the total size of the messages (Table 4.15). The profiles produced by the instrumented executable show that the time spent on the MPI routines is reducing as more threads per process are used but the time spent on program routines is increased.

Threads/process	Msg Count	Total Msg Size	Average Msg Size	Total Msg Size/node
pure MPI	21,106	406.78MB	19.73KB	1.59GB
2 threads	21,105	690.26MB	33.49KB	1.34GB
4 threads	21,117	1.12GB	55.75KB	1.12GB

Table 4.15: Average sent message statistics per process for 24^3x48 lattice size, using 256 processors.

All three versions achieve almost the same performance. There is a trade off between the MPI time and the overheads introduced by the OpenMP library and the serial fraction of the program as different lattice decompositions and more threads per process are used. The most balanced version that achieved the best performance is again the 2 threaded mixed mode version, and this is also applied when the benchmark is run using 1024 processors.

Compared with the previous cases, the mixed mode versions of the program perform considerably better than the pure MPI equivalent. The reason is that the time spent on program routines is not increased as much as in the previous cases when the MPI processes are reduced, but the communication time is significantly reduced for both the mixed mode versions. The increased USER time is the reason that four threaded mixed mode version is barely slower than two threaded version. The change of the process grid has the same effects as in the case of 256 processors. The message count is almost the same for every version but the size of the messages and the total message size are increased when fewer processes are used.

The message statistics per process of all the previous cases showed that the size of the total data that is sent from each process is increased. However, the total data that is transferred through the network is reduced. The increase of the average message size per process and the reduction of total transferred data was beneficial to the performance of the program for 24^3x48 lattice size, but this is not the case for the larger lattice (Figure 4.8).

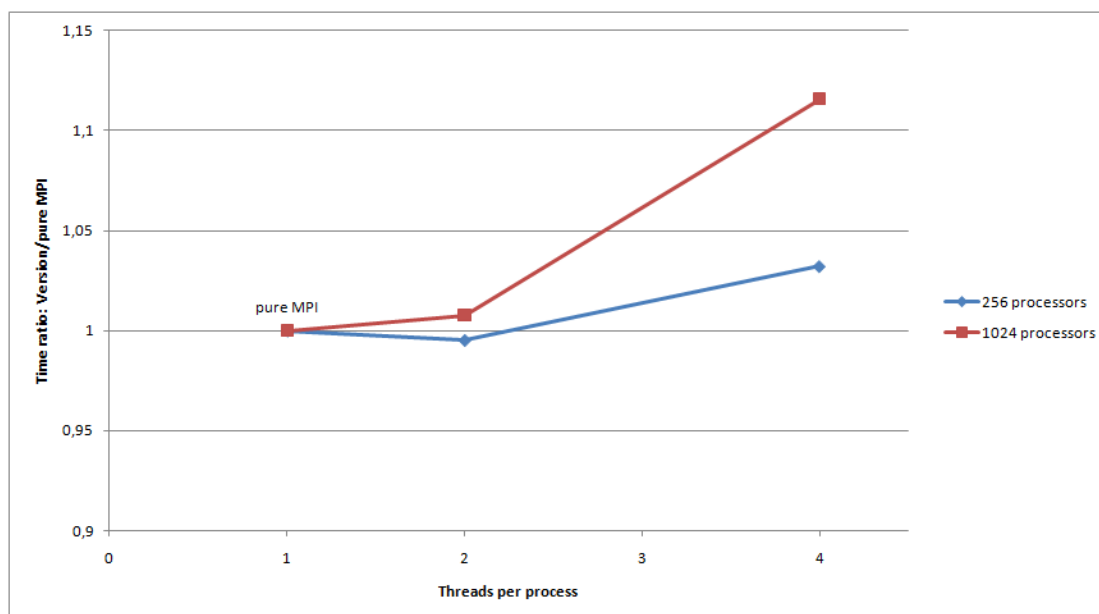

Figure 4.8: BQCD execution time ratio of mixed mode versions to pure MPI for 256 and 1024 processors on 48^3x96 lattice.

The runs of the benchmark for 48^3x96 lattice size on 64 processors could not be performed because the required memory per process exceeded the available memory per node. The profiles of the runs using 256 processors (Table 4.16) show that MPI time is increased when fewer processes are used. Moreover, the time spent on the program routines is increased. For very heavy load per process, the overheads by the sequential parts of the program are more effective in the performance of the program and this may be a reason for the increase of the USER time.

Comparing the time spent on MPI routines of the three versions of the program, it is

Function group	pure MPI		2 threads/process		4 threads/process	
USER	397.98	93%	401.76	89%	418.87	91%
MPI	24.61	5.7%	31.30	7.0%	32.54	7.0%
MPI_SYNC	5.88	1.4%	16.60	3.7%	10.59	2.3%
Total Time	428.52		449.74		462.05	

Table 4.16: Statistics produced by CrayPAT for runs using 256 processors and 48^3x96 lattice size (time in seconds and percentage).

shown that for this problem size, the increase of the total sent message size per process (Table 4.17) aggravates the performance of the program. The interconnect chip seems to handle better the communications between processes when smaller messages are sent. This is the reason why mixed mode versions do not achieve the same performance as the equivalent pure MPI version for this problem size.

Threads/process	Msg Count	Total Msg Size	Average Msg Size	Total Msg Size/node
pure MPI	21,157	2.96GB	146.64KB	11.84GB
2 threads	21,153	5.10GB	253.02KB	10.20GB
4 threads	21,198	8.63GB	426.78KB	8.63GB

Table 4.17: Average sent message statistics per process for 48^3x96 lattice size, using 256 processors.

4.3.2 BQCD on Cray XT6 system

BQCD code was also run on Phase IIb machine of HECToR. The experimental procedure is similar with the one followed on the Cray XT4 system. The runs are grouped based on the lattice size of the program. For 24^3x48 lattice size, the benchmark was run using 192, 384, 768 and 2304 processors, but for 48^3x96 lattice, the benchmark was only run on the last three processor counts due to insufficient memory per node. Table 4.18 presents the lattice decomposition across MPI processes of each version of the program on every processor count that was used for investigating mixed mode on the Cray XT6 system.

Figure 4.9 presents the time ratio of the mixed mode versions to the equivalent pure MPI for the runs with 24^3x48 lattice size. For every processor count the performance achieved by all the mixed mode versions is better than the performance of the pure MPI version. Furthermore, as the number of the processors is increasing, mixed mode version becomes more beneficial for the performance of the program. However, the number of threads per process that delivers the best performance is different for each processor count and is increased as more processors are used.

When the code is compiled with GNU compiler, CrayPAT cannot produce an instrumented executable which can measure the thread times and the overheads introduced by OpenMP library effectively. So, the profiles of CrayPAT show how the execution

Threads/process	192 processors	384 processors	768 processors	2304 processors
pure MPI	2x4x4x6	4x4x4x6	4x4x4x12	4x4x12x12
2 threads	2x2x4x6	2x4x4x6	4x4x4x6	4x4x6x12
3 threads	2x2x4x4	2x4x4x4	4x4x4x4	4x4x4x12
4 threads	2x2x3x4	2x2x4x6	2x4x4x6	4x4x6x6
6 threads	2x2x2x4	2x2x4x4	2x4x4x4	4x4x4x6
8 threads	2x2x2x3	2x2x3x4	2x2x4x6	2x4x6x6
12 threads	1x2x2x4	2x2x2x4	2x2x4x4	2x4x4x6
24 threads	1x2x2x2	2x2x2x2	2x2x2x4	2x2x4x6

Table 4.18: The dimensions of the processor grid of 192, 384, 768 and 2304 processors.

time of the traced functions varies among the different versions of the program, but the exact reason of this variation cannot be determined.

The profile of the pure MPI version on 192 processors using the small lattice shows that more than the half of the execution time is spent on program routines (Table 4.19). The rest of the time is spent on MPI communications. Every time the number of processes is reduced, the MPI time is reduced and this is the main reason that mixed mode versions achieve better performance than the equivalent pure MPI version. However, the performance is not always improved when more threads per process are used. In almost every number of processors that was used, the performance was not improved as the number of threads per process was increased from six to eight and from 12 to 24, but it

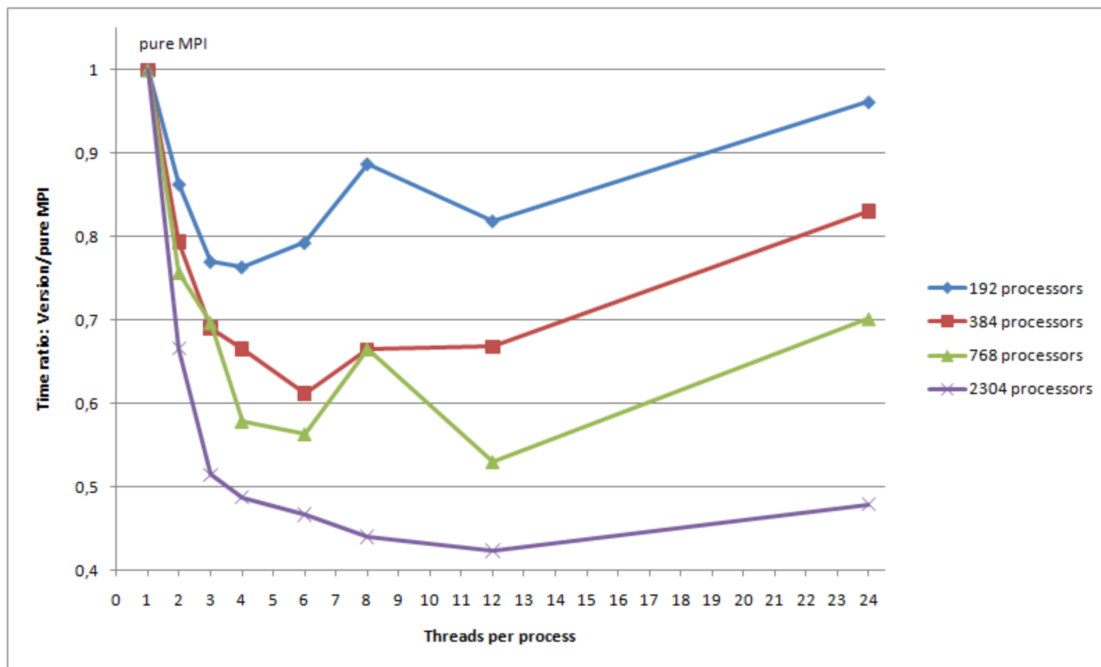

Figure 4.9: BQCD execution time ratio of mixed mode versions to pure MPI for 96, 384, 768 and 2304 processors on 24^3x48 lattice.

seems that the reason for the degradation of the performance is not the same for these two cases.

Threads/process	USER		MPI		MPI_SYNC		Total Time
pure MPI	32.49	59%	21.21	39%	1.02	1.9%	54.76
3 threads	35.02	77%	9.26	20%	1.03	2.3%	45.35
6 threads	35.71	77%	9.29	20%	1.27	2.8%	46.32
8 threads	37.54	67%	16.61	30%	1.43	2.6%	55.64
12 threads	40.67	80%	8.77	17%	1.43	2.8%	50.92
24 threads	45.40	77%	11.81	20%	1.27	2.2%	58.55

Table 4.19: Statistics produced by CrayPAT for runs using 192 processors and 24^3x48 lattice size (time in seconds and percentage).

When eight threads per process are used instead of six, the MPI time is considerably increased. The increase of the total message count and the total message size per process is not as significant as in other cases (Table 4.20) in order to affect in such extend the communication time of the program. The factor that could influence the communication pattern of the program is the shape of the process grid. Increasing or reducing the processes in each dimension of the process grid has different effects in the performance. The performance is improved only in the case of 2304 processors where the processes that are replaced by OpenMP threads are removed from the 1st dimension of the process grid. Further investigation is needed to understand how changing the number of processes in each dimension affects the efficiency of the communications of the program, but this is not in the scope of the project.

Threads/process	Msg Count	Total Msg Size	Average Msg Size	Total Msg Size/node
pure MPI	50,288	0.51GB	10.64KB	12.25GB
2 threads	50,286	0.86GB	17.94KB	10.32GB
3 threads	50,309	1.17GB	24.48KB	9.40GB
6 threads	50,368	1.91GB	39.84KB	7.64GB
8 threads	50,410	2.40GB	50.06KB	7.20GB
12 threads	21,199	2.19GB	108.24KB	4.38GB
24 threads	21,308	3.61GB	177.60KB	3.61GB

Table 4.20: Average sent message statistics per process for 24^3x48 lattice size, using 192 processors.

However, the runs that were performed for the investigation of mixed mode programming revealed that when the number of processes of a dimension is reduced from any count to one, the average number of messages that are send by each process is reduced. This happens because when only one process lies on a dimension of the process grid there is no need for communication inside this dimension and the average message count per process is reduced.

The reason for the poor performance when the number of threads per process is increased from 12 to 24 is not only the increase of the MPI time of the program but also

the increased execution time of the program routines which is also applied on greater processors counts (Table 4.21). The USER time always increases when more threads per process are used. However, in this case this increase is substantial. The most sensible cause are the overheads which are introduced by the OpenMP library and the sequential execution of some parts of the code due to lack of OpenMP parallelism but there is not the capability of identifying and quantifying theses overheads when using GNU compiler.

Threads/process	**USER**		**MPI**		**MPI_SYNC**		**Total Time**
pure MPI	14.62	44%	18.00	54%	0.63	1.9%	33.28
2 threads	16.99	56%	12.36	41%	1.09	3.6%	30.48
4 threads	18.08	64%	9.18	32%	0.92	3.3%	28.23
6 threads	18.24	70%	7.17	27%	0.76	2.9%	26.20
12 threads	19.97	69%	7.84	27%	0.92	3.2%	28.78
24 threads	24.53	70%	9.79	28%	0.77	2.2%	35.15

Table 4.21: Statistics produced by CrayPAT for runs using 384 processors and 24^3x48 lattice size (time in seconds and percentage).

Figure 4.9 shows that the mixed mode version that achieves the best performance is different for each processor count. However, for 192, 384 and 768 processors, the process grid of the most efficient mixed mode version has almost the same shape producing very similar sent message statistics. For these three cases, the total message count per process is 50,309 and the total message size is 1,261,458,449 bytes producing an average message size of 24.48KB (Tables 4.20 and 4.22), which shows that this average message size is the optimal for achieving good communication performance on the network of the Cray XT6 system. It also verifies the previous statement that the shape of the process grid is determinant of the performance of the program.

Threads/process	**Msg Count**	**Total Msg Size**	**Average Msg Size**	**Total Msg Size/node**
pure MPI	50,288	0.33GB	6.99KB	7.29GB
2 threads	50,288	0.51GB	10.64KB	6.12GB
4 threads	50,286	0.86GB	17.94KB	5.16GB
6 threads	50,309	1.17GB	24.48KB	4.68GB
12 threads	50,368	1.91GB	39.84KB	3.82GB
24 threads	50,494	3.39GB	70.46KB	3.39GB

Table 4.22: Average sent message statistics per process for 24^3x48 lattice size, using 384 processors.

When the same experiments are performed for the 48^3x96 lattice, the size of the data that travels through the network is increased significantly. Figure 4.10 shows that the trend in the performance of mixed mode versions compared with the equivalent pure MPI is the same for all three processors counts. It seems that for this amount of data, the message size does not affect considerably the performance of the communications

and the MPI time is mainly determined by the lattice decomposition across the MPI processes.

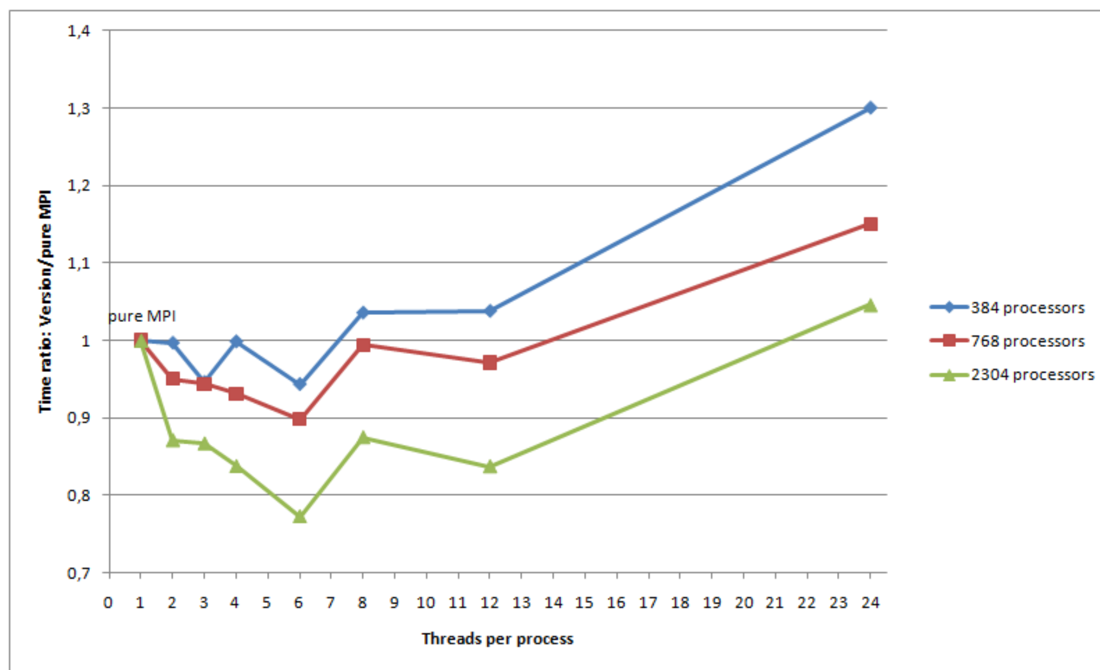

Figure 4.10: BQCD execution time ratio of mixed mode versions to pure MPI for 384, 768 and 2304 processors on 48^3x96 lattice.

The runs of the BQCD code on both systems of HECToR showed that, in some cases, mixed mode programming increased the performance of the program by improving the communication between the MPI processes, especially on the Cray XT6 system where there are many processors contenting for the same interconnect chip. However, as the performance is also affected by the decomposition of the lattice over the process, it is not always possible to distinguish whether this improvement comes from the change of the process grid or because of using mixed mode programming to run the program, especially on the Cray XT4 system where the improvement is minimal.

4.3.3 Memory usage of BQCD

The average memory "high water mark" per process was increasing as the number of processes were increasing. However, this increase was not proportional to the reduction of the number of processes per node. Thus, the memory usage per node was reduced as more threads per process were used (Figure 4.11). Mixed mode programming did not achieve the same reduction of the memory usage per node as in the case of the CPMD code where there was replication of data on all processes. However, the reduction of the memory usage per node is not negligible on both systems.

On the Cray XT4 system, the memory usage per node of the four threaded mixed mode

version was about 1.8 times lower than the equivalent memory usage of the pure MPI version. The reduction of the number of processes per node was more effective on the Cray XT6 system because of the more processes per node of the pure MPI version. The memory “high water mark” per node of the mixed mode version usign 24 threads per process is about 2.4 times lower than the memory usage of the pure MPI version on the Cray XT6 system. However, the main improvement was noticed for thread counts up to six. Above this number of threads, the memory usage per node was not reduced significantly.

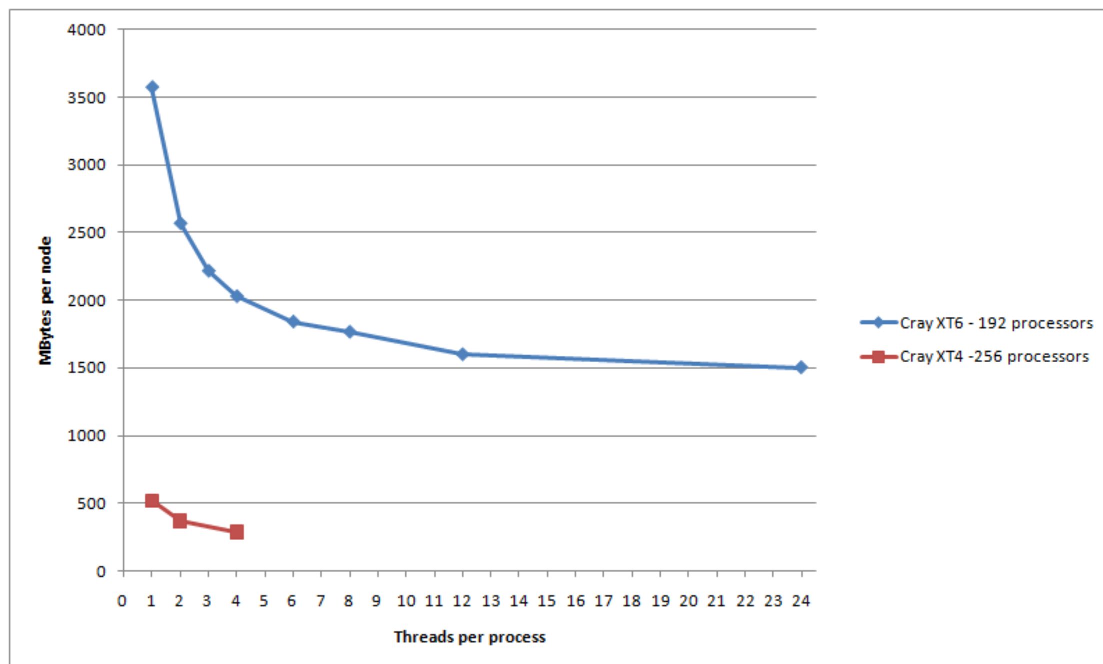

Figure 4.11: Memory usage of BQCD program for 24^3x48 lattice using 256 processors on the Cray XT4 system and 192 processors on Cray XT6 system.

4.4 SP-MZ

The process count of a run has to be declared at the compile time of the program. So, a different executable is used for each mixed mode version. In order to reduce the execution time of the program, the number of iterations per run is limited to 100. Thus, the computation of the program does not converge at the predefined values of each problem size.

The same problem sizes were used for both systems of HECToR. The first experiments were performed on relatively small processor counts for problem class C. This problem class involves 16x16x1 number of zones which form a grid with dimensions 480x320x28. The needed memory for this class is about 0.8 GB. The experimental procedure continues on problem class D. This class involves 32x32x1 number of zones

which are four times more than the number of zones of class C. So, the used processor count is also quadruplicated on each system of HECToR. Although each processor has the same number of zones as in the case of class C, the size of each zone is about four time larger on class D, so each processor has more work to do. The aggregate grid of this class is 1632x1216x34 which requires about 12.8 GB of memory.

The latest version of the PGI compiler was used for the build of this code (9.0.4 and 10.3.0 on Phase IIa and Phase IIb systems respectively). CrayPAT cannot measure the overheads that are introduced by the OpenMP library when the code is compiled with the GNU compiler. However, on the Cray XT4 system another profiling tool can be used with the GNU compiler which is called Scalasca. From all the benchmarks that are used for this project, only SP-MZ was successfully profiled with Scalasca. So, this benchmark code provided the opportunity to compare on the Cray XT4 system the performance of the program when it is built by two different compilers.

The compiler flags that were used for the PGI compiler are the following:

- **-fastsse**: This flag is equivalent with the -fast option that was used on the CPMD benchmark.
- **-r8**: This option forces the compiler to treat REAL variables as double precision numbers.
- **-mp[=nonuma, for the Cray XT4 system]**: Enables the recognition of OpenMP directives and links the OpenMP library to the program.

When the code is compiled on the Cray XT4 system with the GNU compiler (version 4.4.2) the following compiler flags are used:

- **-O3**: Enables a variety compiler optimisations in order to improve the performance of the program.
- **-fopenmp**: This flags links the OpenMP library to the code and enables the OpenMP directives recognition.

4.4.1 SP-MZ on Cray XT4 system

For problem class C, the program is run on 64 processors whereas, for problem class D the number of processors is increased to 256. At first, the code is build using the PGI compiler. The performance of the program is presented in Figure 4.12. On 64 processors, the mixed mode versions of the benchmark achieve about 25% better performance than the equivalent pure MPI version. However, this improvement is not applied on 256 processor count where the mixed mode version using two threads per process is slightly slower than pure MPI, and the mixed mode version of four threads per process is only 7% faster than pure MPI.

The output of the pure MPI version on 64 processors and its profile show that the work is evenly distributed across the processes. About the one third of the execution time

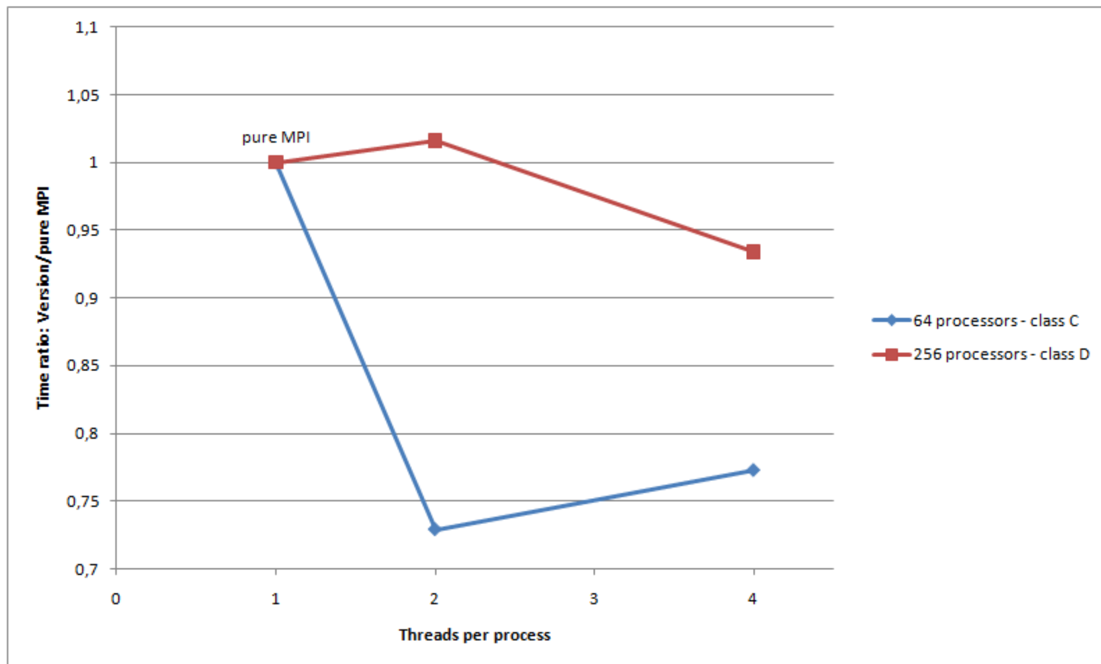

Figure 4.12: SP-MZ time ratio using the PGI compiler on the Cray XT4 system.

is spent on MPI communications and the rest of the time is spent on program routines (Table 4.23). The average message count per process is 466 and each message has size about 72 KB. Moreover, the improvement in the performance of the program when it is run using two threads per process, comes from the significant reduction of both the MPI time and the time spent on program routines.

Threads/process	**USER**		**MPI**		**OMP**		**Total Time**
pure MPI	5.34	70%	2.23	29%	- - -	- - -	7.66
2 threads	4.15	85%	0.63	13%	0.05	1.2%	4.86
4 threads	4.79	92%	0.26	5.1%	0.10	2.0%	5.19

Table 4.23: Statistics produced by CrayPAT for runs using 64 processors and problem class C (time in seconds and percentage).

The USER time of the mixed mode version using two threads per process is 23% lower than the equivalent time of the pure MPI version. The total USER time per thread shows that the work is almost evenly distributed across the threads and the difference in the work load between master thread and thread one is not more than 3%. Furthermore, all the most time consuming functions are parallelised using OpenMP threads. So, the portion of the code that is executed sequentially is very small.

By replacing the MPI processes with OpenMP threads, the resulting MPI time is more than three times lower than the equivalent time of the pure MPI version. The average message count per process is reduced to 315 and the average message size is increased to 112KB. However, the intensive OpenMP parallelisation introduces some overheads

in the execution time of the program which are insignificant.

When the program is run using four threads per process, the execution time is slightly higher than the execution time of the mixed mode version using two threads per process. This is caused by the increase of the OpenMP overheads introduced to the program and the higher USER time which cannot be covered by the faster MPI communications. However, this mixed mode version outperforms the pure MPI version of the program.

The performance of the mixed mode versions of the program compared with the pure MPI version is not the the same when both the problem size and the processor count are increased. Now that each process has about four times more work to do, the overheads introduced by the sequential parts of the code are higher, so the time spent on program routines is almost equal for both mixed mode versions and higher than the equivalent time of the pure MPI version. Thus, the MPI time determines whether the execution time of the mixed mode versions is higher or lower than execution time of pure MPI (Table 4.24).

Threads/process	**USER**		**MPI**		**Total Time**
pure MPI	25.90	87%	3.68	12%	29.72
2 threads	27.08	91%	2.38	8.0%	29.76
4 threads	26.82	96%	0.80	2.9%	27.92

Table 4.24: Statistics produced by CrayPAT for runs using 256 processors and problem class D (time in seconds and percentage).

The code was also compiled with the GNU compiler and it was run using the same processor counts and problem sizes. All versions of the program that were created by the GNU compiler were slower than the executables created by the PGI compiler. Figures 4.13 and 4.14 present the performance of the mixed mode versions of both compilers compared with the corresponding pure MPI version of each compiler on 64 processors for problem class C and on 256 processors for problem class D respectively. The performance of the program compiled by the GNU compiler does not improve as much as the program compiled by the PGI compiler when mixed mode programming is applied for 64 processor counts. Furthermore, on 256 processors, the pure MPI version always outperforms the mixed mode versions of the program.

Scalasca does not provide the same information with CrayPAT in order to directly compare the profiles of the various versions. In the profiles of Scalasca, the aggregated values of each process/thread are reported. The average of these values per process/thread is used for comparing PGI and GNU compilers.

Tables 4.25 and 4.26 contain information about the execution time of the program produced by Scalasca in a form that is comparable with the output of CrayPAT. The time spent on program functions of the GNU compiler is higher than the same time of the PGI compiler.

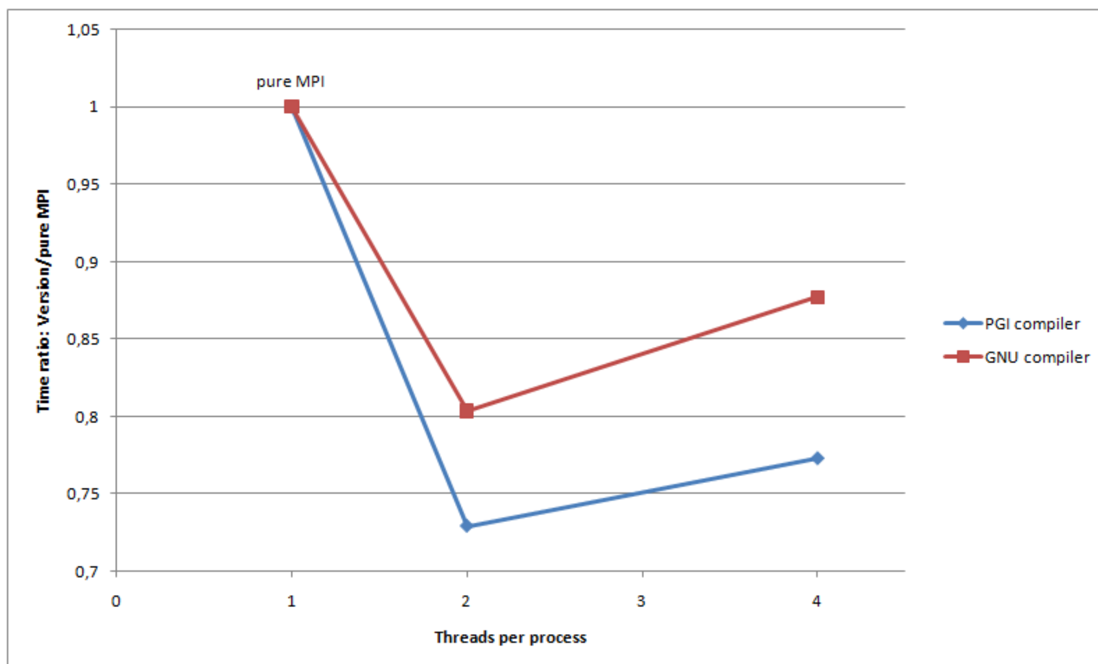

Figure 4.13: Comparison of PGI and GNU compiler on 64 processors.

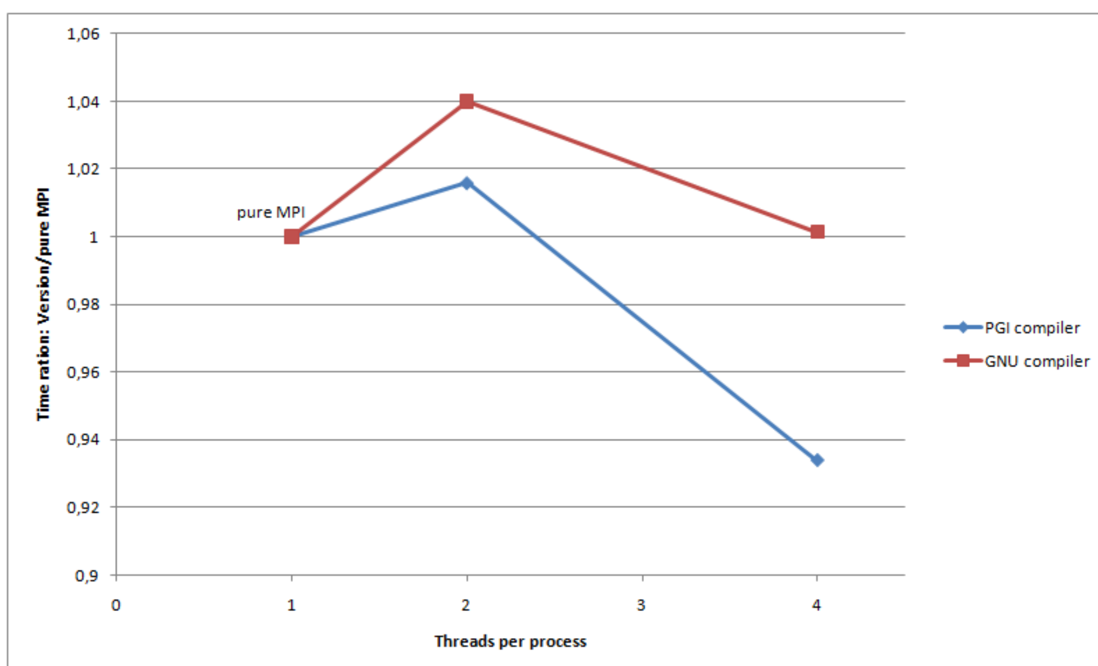

Figure 4.14: Comparison of PGI and GNU compilers on 256 processors.

The improvement of these times when mixed mode programming is applied is similar for both compilers. The same improvement is also noticed in the MPI communications. The time spent on MPI communications and the time spent on program routines

of mixed mode codes of these two compilers follow the same trend . However, the reported time that is spent on the OpenMP library is significantly higher on the GNU compiler. The reported OpenMP times by Scalasca and CrayPAT may not represent the real difference in the OpenMP overheads introduced by these two compilers but it justifies the difference in the performance of the mixed mode versions of GNU and PGI compilers.

Threads/process	USER	MPI	OMP	Total Time
pure MPI	6.16	0.92	- - -	7.09
2 threads	5.07	0.55	1.42	7.05
4 threads	5.29	0.29	3.90	9.49

Table 4.25: Statistics produced by Scalasca for runs using 64 processors and problem class C (time in seconds and percentage) compiled by the GNU compiler.

Threads/process	USER	MPI	OMP	Total Time
pure MPI	28.52	2.78	- - -	31.31
2 threads	29.23	2.08	3.01	34.32
4 threads	28.3	0.59	11.45	40.34

Table 4.26: Statistics produced by Scalasca for runs using 256 processors and problem class D (time in seconds and percentage) compiled by the GNU compiler.

4.4.2 SP-MZ on Cray XT6 system

Problem classes C and D were also used for Phase IIb machine of HECToR. However, the number of zones of both classes (256 and 1024) cannot be divided by any processor count of this system where all the processors of the allocated nodes are active during the execution of the program. This benchmark code was run on 96 and 386 for problem classes C and D respectively. Thus, there is a small load imbalance across the process of the runs on the Cray XT6 system.

Figure 4.15 shows that most of the mixed mode versions outperform the equivalent pure MPI version of the program on both problem classes. The most efficient mixed mode version of each problem achieves almost the same improvement in performance. However, the number of threads that is used on these runs is different and a careful look at the profiles of the program should provide an explanation about that.

On 96 processors, the performance of the program is improved only when the number of threads per process are increased from one to two and from two to three. At the first increase of the number of threads, both the execution time of program routines and the time spent on MPI communications are reduced resulting in a lower total execution time than pure MPI version (Table 4.27). However, for 3 threads per process, the USER time becomes equal to the equivalent time of pure MPI version but the MPI communications

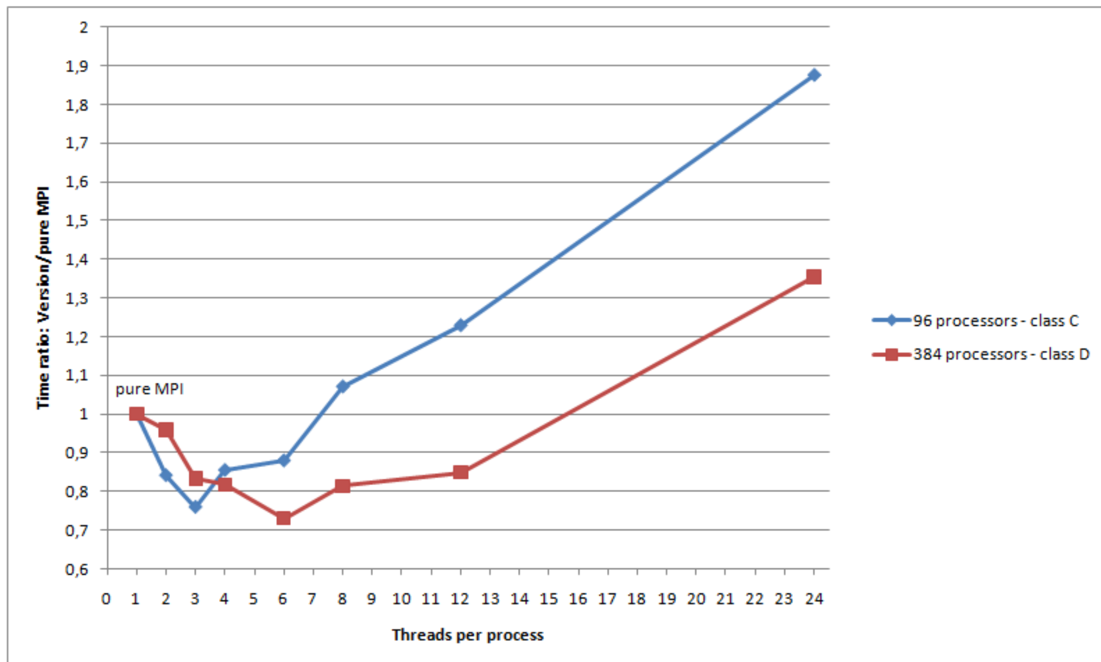

Figure 4.15: SP-MZ time ratio on the Cray XT6 system.

are performed three times faster than in the case of pure MPI and the overall performance is improved by almost 25%.The overheads which are introduced to the program from the load imbalance across the processes and the OpenMP library occupy a small portion of the execution time and they do not affect considerably the total execution time.

Threads/process	USER		MPI		MPI_SYNC		OMP		Total Time
pure MPI	2.94	71%	1.10	26%	0.06	1.6%	- - -		4.12
2 threads	2.66	75%	0.75	21%	0.11	3.3%	- - -		3.56
3 threads	2.92	85%	0.36	10%	0.06	1.9%	0.07	2.2%	3.43
6 threads	3.49	88%	0.28	7.2%	0.06	1.5%	0.10	2.7%	3.94
8 threads	4.04	86%	0.46	9.8%	0.04	1.0%	0.16	3.5%	4.72
24 threads	7.24	87%	0.28	3.4%	- - -		0.71	8.5%	8.31

Table 4.27: Statistics produced by CrayPAT for runs using 96 processors and problem class C (time in seconds and percentage).

For more than three threads per process, the improvement of the MPI communications cannot compensate the increase of both the execution time of the program routines and the overheads of the OpenMP library. Thus, the total execution time increases along with the number of threads per process, and for more than six threads per process this time is higher than the execution time of the pure MPI version.

On 384 processors, only the mixed mode version using 24 processors is slower than the pure MPI version. Mixed more programming is more beneficial for the performance

of the program for the larger problem size than it is for problem class C, which is the opposite case of the runs on the Cray XT4 system. This was not expected because as it is already seen in previous benchmarks, the overheads introduced by the execution of the sequential parts of the code have greater impact on the performance of the program when the work load per process is increased. Contrariwise, the profiles of the runs on 384 processors show that execution time of the program routines is reducing as the number of threads per process is increasing (Table 4.28). This improvement ends when the number of threads is 6, where the best performance of the program is achieved.

Threads/process	USER		MPI		MPI_SYNC		OMP		Total Time
pure MPI	15.55	76%	4.65	23%	0.32	1.6%	- - -		20.55
2 threads	16.49	77%	4.36	20%	0.34	1.6%	- - -		21.33
3 threads	15.82	85%	2.09	11%	0.18	1.0%	- - -		18.26
6 threads	15.23	92%	0.98	6.0%	- - -		0.16	1.0%	16.48
8 threads	15.98	86%	1.90	10%	- - -		0.60	3.3%	18.60
24 threads	27.30	91%	1.43	4.8%	- - -		1.28	4.3%	30.12

Table 4.28: Statistics produced by CrayPAT for runs using 384 processors and problem class D (time in seconds and percentage).

The MPI communications are getting faster every time the number of MPI processes is reduced, except for the cases where the number of processes per node is less than three. The average number of messages per process is not as large as in other cases where the contention over the interconnect network was high. The faster MPI communications mainly come from the increase of average message size of each process, which helps to achieve better bandwidth and hide the latency of the network. Furthermore, the OpenMP overheads start to affect the performance of the program when the number of threads per process is more than eight.

The runs of this benchmark code on both systems of HECToR showed that the OpenMP parallelism that is applied to the code is adequate to replace the MPI processes with threads and achieve similar or better performance than the pure MPI version of the program. Furthermore, the intensive use of parallel regions in the code does not introduce much overheads by the OpenMP library to the execution of the program, especially when the PGI compiler is used for the build of the code.

4.4.3 Memory usage of SP-MZ

Similarly to the previous benchmarks, mixed mode programming reduced the memory requirements per node of the program (Figure 4.16). On the Cray XT4 system, the memory usage per node of the pure MPI version of the program is about 1.6 times higher than the memory high water mark of the four threaded mixed mode version. This difference in memory usage was higher on the XT6 system, where the memory usage per node of the pure MPI version was about 2.4 times higher than the memory usage of the 24 threaded mixed mode version.

The memory that is required for the problem class D is about 12.8 GB. On the Cray XT4 system, the number of nodes allocated for 256 processors is 64. The size of the problem that is assigned to each node is about 205MB. However, when the program was run using four threads per process, the memory high water mark is 325MB. So, about 120MB of additional memory was needed by the process of each node during the execution of the program. On the XT6 system, the allocated nodes for 384 processors are 16. So, each node would be assigned about 820 MB of the problem size. The memory "high water mark" that was recorded by CrayPAT for 24 threads per process is 1190 MB (370MB of additional memory).

This difference in the additional memory per process between the two systems of HECToR was expected. The additional memory needed by each process during the execution of the program is increased when the problem size per process is increased because the matrices that are used to perform various tasks with the problem data, such as packing and unpacking halo data into buffers, are also increased.

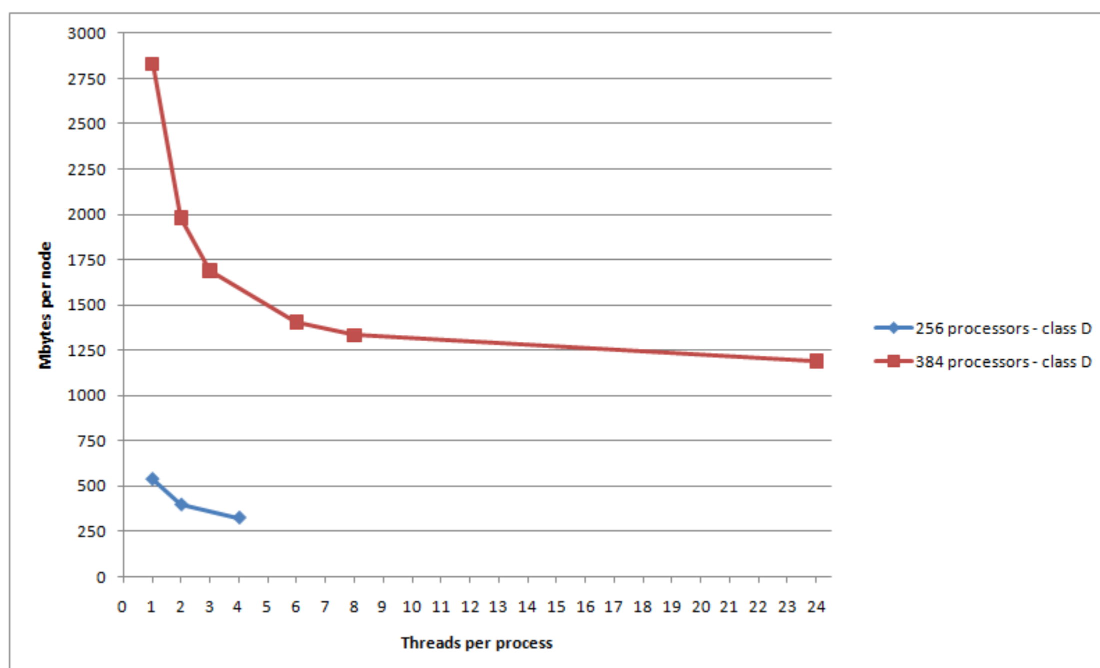

Figure 4.16: Memory usage of SP-MZ benchmark for class D problem size using 256 processors on Cray XT4 system and 384 processors on Cray XT6 system.

4.5 IRS

The build of this benchmark code involves a set of Perl scripts which are contained in the provided archive. So, the path to these scripts has to be added to the environmental variable PATH. PGI compiler was used for the build of this benchmark code on both

systems (version 9.0.4 on Phase IIa system and version 10.3.0 on Phase IIb system). The following compiler flags were set in the Makefile of the codes:

- **-fastsse**: This flag is equivalent with the -fast option that was used on the CPMD benchmark.
- **-mp[=nonuma, for the Cray XT4 system]**: Exactly the same options with the CPMD benchmark about the OpenMP library are applied for the build of this benchmark.
- **-c**: This flag instructs the compiler to skip the linking phase and stop after compiling and assembling the code. The output is saved into an object file.
- **-Bstatic**: The linker performs static binding.

As it is mentioned in the description of the benchmark, the physical problem is a 10x10x10 cm^3 mesh with variable resolution, which is divided into domains. In order to achieve even load distribution, the domains have to be divided equally across the processors. However, there is the restriction in the number of domains of a run where this number can only be integers cubed (8, 27, 64, 214...). Thus, the processor counts that can be used on both systems have to divide equally the number of domains and they have to be equally divided by the number of processors per node in order not to have idle processors inside the allocated nodes on both machines. So, this restriction highly limits the number of the possible experiments on HECToR, especially on the Cray XT6 system.

The benchmark code measures the time needed per zone iteration in microseconds. When the number of processors is equal to the number of domains, this time should be constant for different processor counts for perfect scalability. The same should happened when the processes are replaced with OpenMP threads. However, the total execution time is also measured which is very useful for better understanding the performance of the program.

4.5.1 IRS on Cray XT4 system

The domains counts that are used on the Cray XT4 system are 8, 64 and 216 because these numbers are divided by four (the number of processors per node) which ensures the even distribution of the mesh across the nodes and by extension across the processors. Furthermore, by using this number of domains, the HPC resource usage is kept low. Two different types of experiments were performed with this benchmark in order to investigate mixed mode programming. At the first set of experiments, the number of domains is equal to the number of the used processors. Then, for 64 and 216 domains, the processor counts are reduced in order to observe the behaviour of the program when each processor is overloaded.

Figure 4.17 shows the performance of mixed mode versions of the program in relation to the performance of the program when it is run using one thread per process (pure MPI

versions) for 8, 64 and 216 domain/processor counts. It is not clear which functions of the code are involved in this timing, thus the profiles of CrayPAT help only to understand which parts of the program are affected by the change in the number of processes and threads.

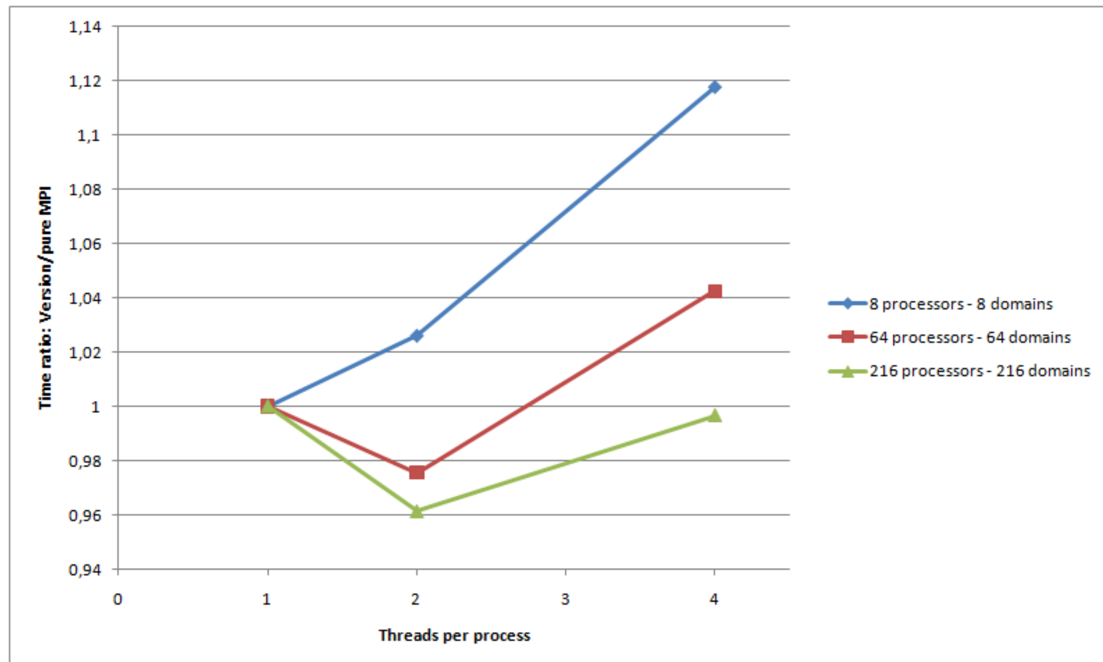

Figure 4.17: IRS time ratio per zone-iteration of mixed mode versions to pure MPI.

For eight processors, mixed mode programming increases the time that is needed per zone-iteration each time the number of threads per process is increased. However, as the number of domains/processors is increasing, mixed mode programming is becoming more beneficial to the performance of the program. Especially for 216 domains, all mixed mode versions outperform the pure MPI version on this part of the program. However, this timing is misleading about the total performance of the program.

In every case, the total execution time of the program is increased (Figure 4.18). The main reason is that OpenMP parallelism is not applied to all the functions of the program. There are only two functions on which most of the execution time is spent, the *main* function and function *rmatmult3*. OpenMP parallelism is only applied to *rmatmult3* and the time spent in this function remains almost the same when MPI processes are replaced with OpenMP threads. In contrast, the execution time of the *main* function is significantly increased when less processes are involved in the run of the program. So, due to the fact that a major portion of the execution time is spent on main function, the overall performance of four threaded mixed mode version of the program may be up to 40% worse than the equivalent pure MPI version.

It is clear now, that the overheads introduced to the program by the lack of OpenMP parallelism are significant. Nevertheless, at the part of the program which is timed (the

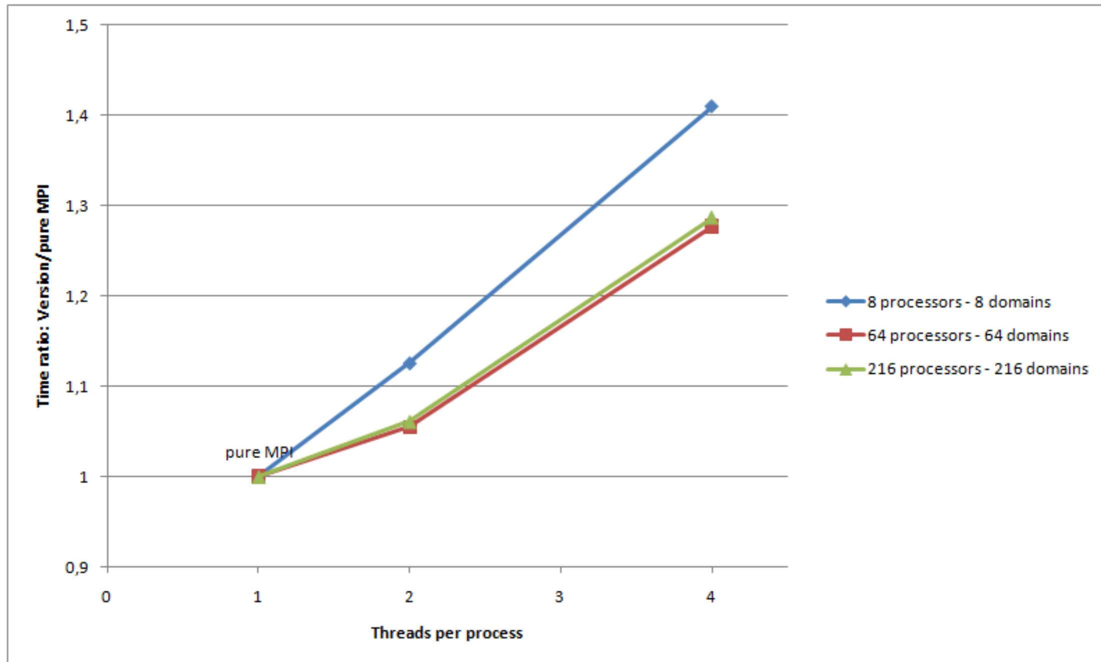

Figure 4.18: IRS execution time ratio of mixed mode versions to pure MPI.

time per zone-iteration) these overheads are not substantial and sometimes there are compensated by the faster MPI communications that are performed when mixed mode programming is applied. This improvement on MPI communications is observed to the mixed mode runs of all processor counts.

The mixed mode versions spend less time in both point-to-point and collective communications. For eight domains, the gains from the reduction of the MPI time are insignificant (Table 4.29). So, the overheads due to the lack of OpenMP parallelism dominate the execution time of the program and even the time per zone-iteration is increased as the number of processes is reduced. The reduction of the MPI time is achieved by performing better utilization of the network. When the number of MPI processes is reduced, fewer but larger messages are sent achieving higher bandwidth and hiding the latency of the network. In the case of eight domains, even when threads threads per process are used, the message size average message size per process is about 13.7KB.

Threads/process	**USER**		**MPI**		**MPI_SYNC**		**Total Time**
pure MPI	4.15	93%	0.18	4.2%	0.13	3.0%	4.47
2 threads	4.81	95%	0.15	2.9%	0.12	2.4%	5.08
4 threads	6.33	96%	0.12	1.8%	0.13	2.0%	6.58

Table 4.29: Statistics produced by CrayPAT for runs using 8 domains and same number of processors (time in seconds and percentage).

Mixed mode programming becomes more beneficial for the measured kernel of the program as the number of domains is increased where the average message count is

increased by more than six times and the average message size of the pure MPI versions is kept the same (about 2.5KB). For 216 processors (Table 4.30), the improvement in the performance of the MPI communication is always greater than the overheads introduced by the lack of OpenMP parallelism, which are increased when less processes are used and they are barely less than the gains of the better MPI communications.

Threads/process	**USER**		**MPI**		**MPI_SYNC**		**Total Time**
pure MPI	10.83	69%	1.85	12%	2.92	19%	15.60
2 threads	13.05	79%	1.40	8.5%	2.09	13%	16.54
4 threads	17.34	87%	0.99	5.0%	1.51	7.6%	19.85

Table 4.30: Statistics produced by CrayPAT for runs using 216 domains and same number of processors (time in seconds and percentage).

In the second set of runs, each process is assigned more than one domain in order to get the processes overloaded. As it is shown in other benchmarks (e.g. CPMD), when the load of each process is increased, the overheads introduced by the execution of the serial parts of the code dominate the execution time. Figure 4.19 presents this behaviour that is observed in the timed kernel of the program.

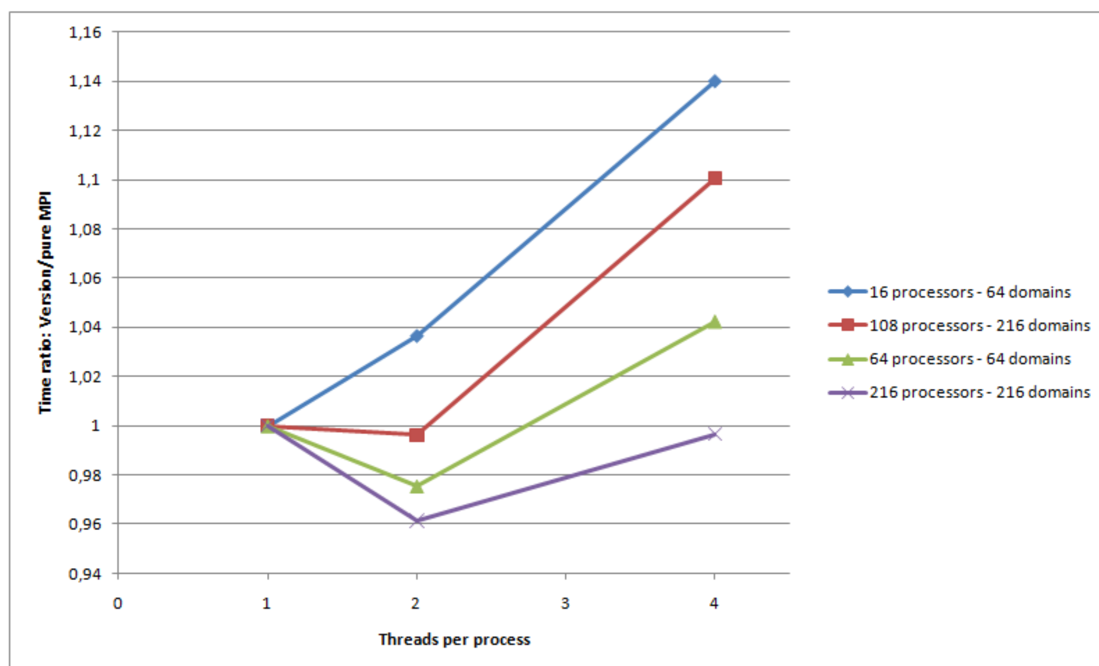

Figure 4.19: IRS time ratio per zone-iteration of mixed mode versions to pure MPI when the number of domains is not equal to the number of processors.

There are two different sets of runs. In the first run, each of the 16 processors is assigned three domains and in the second run, each of the 108 processors works on two domains. In both cases, due to the fact that the processes have more work to do, the portion of the time that is spent on MPI communication is very small now and most of the

execution time is spent on the program routines (Table 4.31). So, the possible gains in performance by the faster MPI communications when fewer processes are used cannot compensate the introduced overheads. So, the only mixed mode version whose kernel achieved similar performance with the equivalent pure MPI is the two threaded version on 108 processors.

Threads/process	USER		MPI		MPI_SYNC		Total Time
pure MPI	22.19	86%	1.78	7.0%	1.73	6.7%	25.71
2 threads	25.62	90%	1.40	8.5%	1.64	5.8%	28.49
4 threads	33.55	93%	1.22	4.3%	1.40	3.9%	35.93

Table 4.31: Statistics produced by CrayPAT for runs using 216 domains and 108 processors (time in seconds and percentage).

The increase of the total execution time of the program when two or more threads per process are used is greater for the overloaded processors than in the case where each processors has only one domain (Figure 4.20). The profiles of the program show a considerable increase in the execution time of *main* function, whose portion of the execution time becomes greater when less MPI processes are used.

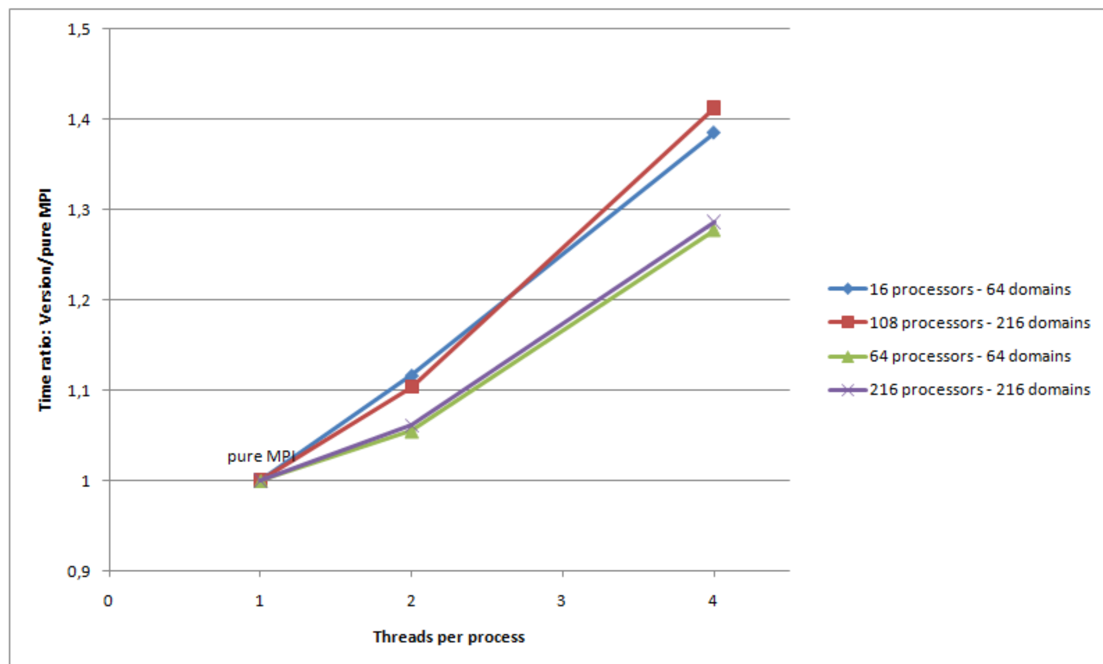

Figure 4.20: IRS execution time ratio of mixed mode versions to pure MPI when the number of domains is not equal to the number of processors.

4.5.2 IRS on Cray XT6 system

The choices of domain and processor counts are very limited. The only integer cubed that can be divided with 24 (the number of processors per node) is the six, so the total number of domains is 216. The program is run on 216 processors, with one domain per processor, and 108 processors where the processors are more loaded (two domains per processor).

The performance of the timed kernel of the program is presented in Figure 4.21. For one domain per process, the reduce of the time per zone-iteration is significant when the mixed mode version of the benchmark is run. The best performance is achieved when three and six threads per process are used where the iteration per zone is about 40% faster than pure MPI version. However, mixed mode programming is not so beneficial when the program is run using two domains per processor.

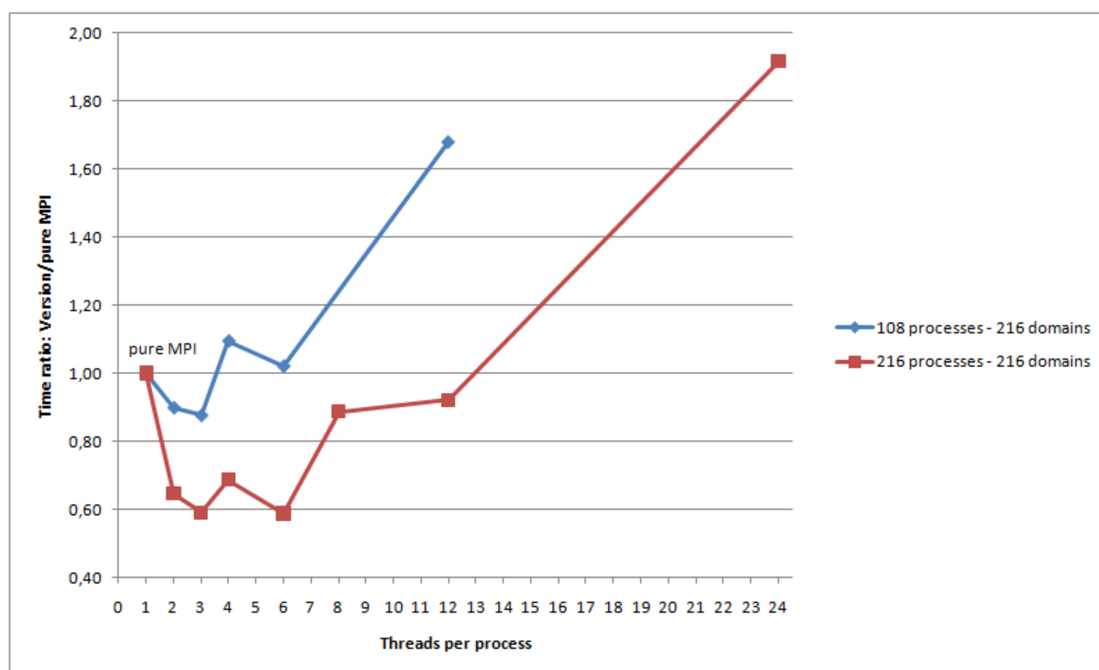

Figure 4.21: IRS time ratio per zone-iteration of mixed mode versions to pure MPI.

As it is shown by the runs of the Cray XT4 system, this timing does not reflect the real performance of the application. Figure 4.22 presents the overall performance of the mixed mode versions of the program in relation to the performance of the pure MPI version. In contrast with the runs on the Cray XT4 system, the execution time of the program, when each processor gets one domain, is reduced for two to six threads per process, whereas in the case of two domains per processor, the reduction of the MPI processes always increased the total execution time.

There are two factor that affect the overall performance of this benchmark when the MPI processes are replaced with OpenMP threads; the overheads which are introduced

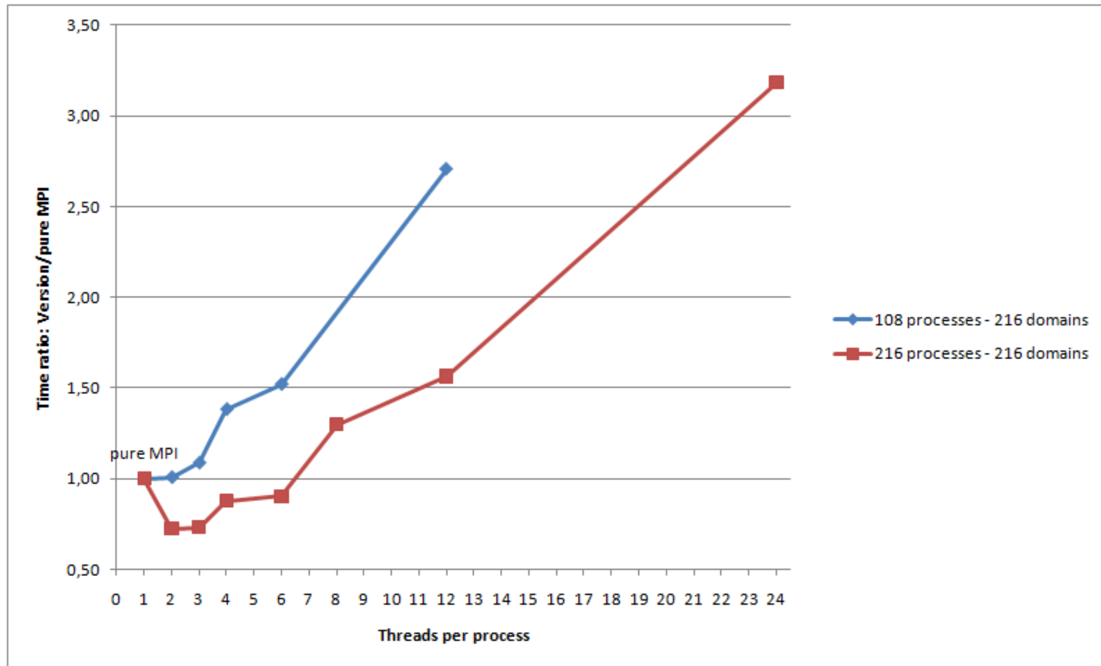

Figure 4.22: IRS execution time ratio of mixed mode versions to pure MPI.

to the program due to lack of OpenMP parallelism and the improvement of the MPI communications. In order a mixed mode version to achieve better performance than the equivalent pure MPI version, the improvement of the MPI communications has to be greater than the overheads that are introduced to the program.

When the number of domains is equal to the number of processors, the reduce of the MPI time (MPI and MPI_SYNC) is massive for two, three and six threads per process where this time is about two, three and five times lower than the equivalent time of the pure MPI version (Table 4.32). The increase of the time spent on program routines due to the lack of parallelism is significantly lower than the improvement of the MPI communications and the performance of the program is improved.

Threads/process	**USER**		**MPI**		**MPI_SYNC**		**Total Time**
pure MPI	10.12	40%	11.75	46%	3.64	14%	25.51
2 threads	12.65	69%	4.44	23%	1.84	9.7%	18.93
3 threads	14.82	76%	3.18	16%	1.47	7.6%	19.48
6 threads	21.13	87%	1.67	6.9%	1.39	5.8%	24.20
12 threads	38.30	93%	1.23	3.0%	1.73	4.2%	41.27
24 threads	79.45	96%	1.58	1.9%	2.03	2.4%	83.07

Table 4.32: Statistics produced by CrayPAT for runs of IRS benchmark using 216 domains and 216 processors(time in seconds and percentage).

As it is already observed on the previous benchmark codes, especially on BQCD, the exchange of a great amount of small messages between the processes is not the com-

munication pattern that can be implemented efficiently on the Cray XT6 system. On the pure MPI version, there are too many processes contenting for network and it seems that SeaStar2+ interconnect chip cannot handle very well this work load. So, the solution for this problem is the reduce of the number of message sent through the network and the reduce of the number of MPI processes trying to access the network.

Both of these actions are performed when the mixed mode versions of the program are run. The message count is considerably reduced along with the processes per node (Table 4.33). For 24 threads per process, the average message count is four times lower that the average message count of pure MPI version, and the average message size is about 45 times larger than the equivalent size of the pure MPI version. This is the reason that the MPI time is reduced from 11.75 seconds to 1.58 seconds.

Threads/process	Msg Count	Total Msg Size	Average Msg Size	Total Msg Size/node
pure MPI	44,192	89.43MB	2.07KB	2.09GB
2 threads	38,632	145.10MB	3.84KB	1.70GB
3 threads	33,143	200.82MB	6.20KB	1.56GB
6 threads	16,750	368.05MB	22.50KB	1.44GB
12 threads	14,688	522.32MB	36.41KB	1.02GB
24 threads	10,335	902.08MB	89.38KB	0.88GB

Table 4.33: Average sent message statistics per process on 216 processors using 216 domains.

For two domains per processor, the portion of the execution time that is spent on MPI routines is lower at the pure MPI version. So, even though the communication is improved in the mixed mode versions, the overheads of the serial parts of the code are always greater and the performance is always reduced when the number of threads per process is increased (Table 4.34).

Threads/process	USER		MPI		MPI_SYNC		Total Time
pure MPI	20.09	68%	6.57	22%	2.78	9.4%	29.47
2 threads	24.98	84%	2.89	9.7%	1.88	6.3%	29.76
3 threads	29.18	90%	1.73	5.4%	1.42	4.4%	32.33
6 threads	41.66	92%	1.66	3.5%	2.22	4.9%	45.49
12 threads	76.27	95%	1.61	2.0%	2.23	2.8%	80.12

Table 4.34: Statistics produced by CrayPAT for runs of IRS benchmark using 216 domains and 108 processors(time in seconds and percentage).

Now that exactly the same runs are performed on both machines of HECToR, it is a good opportunity to compare the achieved performance of the benchmark (Figure 4.23). The execution time of all versions of the program is always lower on the Cray XT4 even though the time spent on program routines is always greater than the equivalent time of the runs on the Cray XT6 system. So, as it is previously discussed, it is now proven that the major bottleneck in the performance of Phase IIb machine is the interconnection of the nodes.

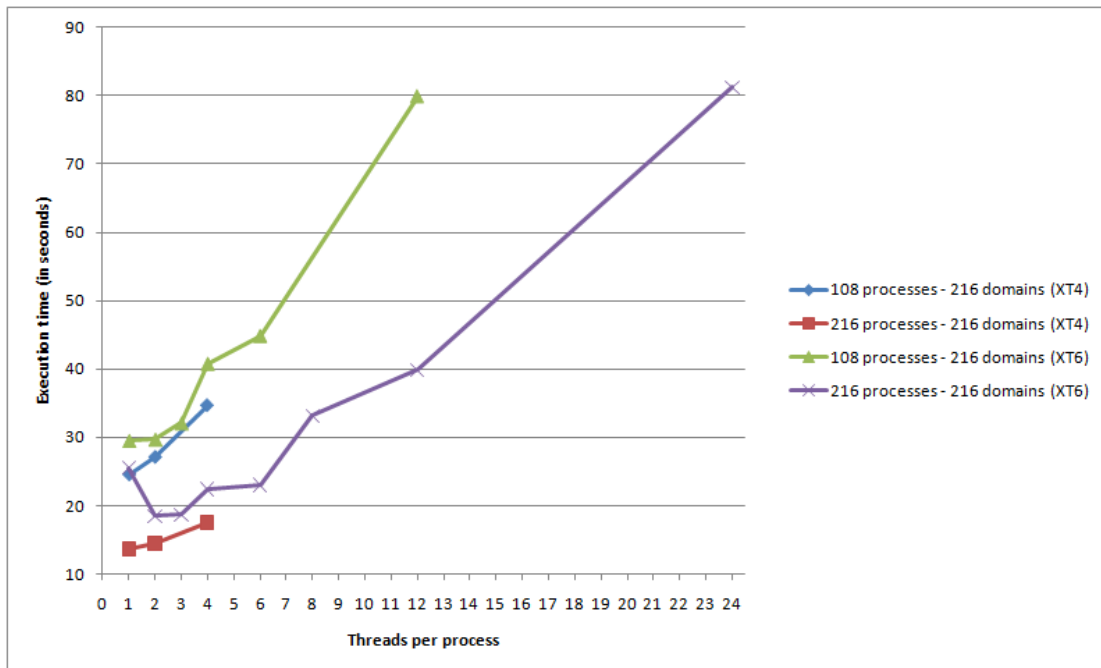

Figure 4.23: IRS execution time on both Cray XT4 and Cray XT6 systems.

The experiments on both machines showed that the threaded parallelisation is not sufficient in order the mixed mode versions of the program to be as efficient as the equivalent pure MPI version. OpenMP parallelism is only applied on some kernels of the code and on the Cray XT4 system the performance was improved by only 4%.

4.5.3 Memory usage of IRS

Similarly to the previous benchmark codes, mixed mode programming helped to reduce the memory usage per node on both systems of HECToR. The memory usage per node for 216 domains on 216 processors is presented in Figure 4.24. Although each process has the same portion of total data on both systems, the data per node is different between the Cray XT4 system and the Cray XT6 system, beacause there are more processes per node on the XT6 system.

Thus, when the program was run using four threads per process on the Cray XT4 system, the memory usage per node was 2.1 times lower than the equivalent memory usage of the pure MPI version. However, on the Cray XT6 system, the memory high water mark per node of the mixed mode version that used 24 threads per process was about 2.9 times lower than the memory usage per node of the pure MPI version.

Moreover, the main improvement in the memory usage per node on the Cray XT6 system is noticed for thread counts up to six. For more than six threads per process, the reduction of the memory usage per node is insignificant. Especially, when the number

of threads per process is increased from 12 to 24, the memory high water mark per node is reduced only by almost 30MB.

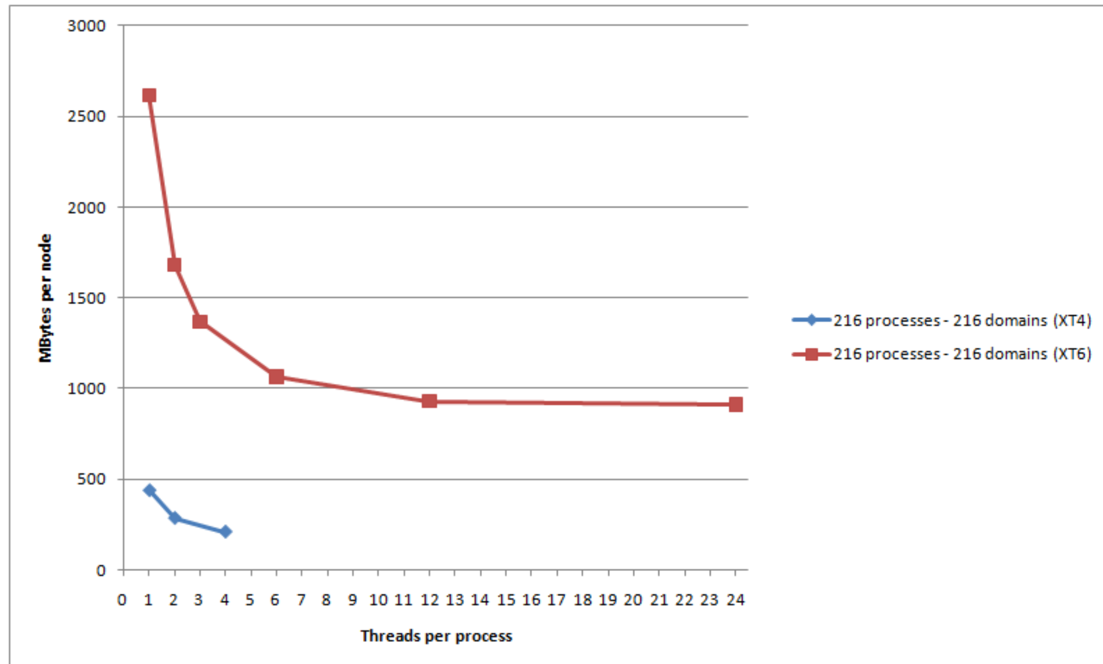

Figure 4.24: Memory high water mark per node of IRS benchmark on both systems.

4.6 Jacobi kernel

In contrast with the previous codes, the fundamental goal of this benchmark is to investigate the performance of mixed mode programming. Each of the three mixed mode schemes is examined separately and it is compared with the pure MPI implementation which is included in the benchmark codes. The pure MPI implementation is also compared with the mixed mode code when it is run using one thread per process in order to identify the overheads introduced by the alteration of the code from pure MPI to mixed mode.

Along with the total execution time, the time spent on the following parts of the program is also measured:

- **Jacobi computation**: The time that is spent on this part of the program is presented as *"jacobi"* in the graphs. This is the total time needed to performed the Jacobi computation on all the elements of the array.
- **Halo swapping**: The program measures the total time spent on the exchange of the halo data between processes. This time is reported as *"halo swap"* in the graphs. In the codes except for the multiple code, the halo swapping time is measured on master process/threads. However, in the multiple code, this time is

the average of the time needed by each thread of the master process to perform halo swapping.

- **Delta calculation**: This routine comprises two tasks. The first task is the calculation of the local delta value squared and the second task involves an all-reduce communication in order for all the processes to have the total delta value squared. So, the measured time for this routine includes the time needed for both these tasks and it is reported as *"delta"* in the graphs.

The time that is not spent on the previous bits of the program is reported as *"other"* which has different meanings among the various mixed mode schemes. The time needed for updating the old array with values of the new array is not measured because instead of using two three dimensional arrays for old and new data, only a four dimensional array is used and the alteration between new and old data is done by just altering the index of the first dimension from zero to one and vice versa.

The timing of multiple runs of the same version of the program did not produce exactly the same results. In order to eliminate as much as possible this fluctuation in the execution time of the program, all the experiments were performed three times and the average of these runs was taken into account for analysing the performance of the program.

PGI compiler is used for the build of the benchmark codes on both machines of HECToR, version 9.0.3 on the Cray XT4 system and version 10.3.0 on the Cray XT6 system. The compiler was called with the following compiler flags:

- **-fastsse**: This flag is equivalent with the -fast option that was used on the CPMD benchmark.
- **-mp**: Exactly the same options with the CPMD benchmark about the OpenMP library are applied for the build of the codes of this benchmark.
- **-lm**: This flag instructs the compiler to link the executable with the math library.
- **-lmpich_threadm**: This library is linked to program when multiple scheme code is compiled in order to support maximum thread safety level.

Furthermore, the environmental variable *MPICH_MAX_THREAD_SAFETY* is set to specify the proper thread safety level, especially for multiple level which has to be declared explicitly.

4.6.1 Jacobi kernel on Cray XT4 system

The problem size per process is fixed on this benchmark. Thus, each process has exactly the same amount of data to work on. The size and the shape of the global three dimensional array is determined by the number of the processes and their topology. On the Cray XT4 system, the size of the array of each process and the process topology are the same as in [15]. The local array has dimensions 192x192x192 and the process

topology of pure MPI version is 16x4x2 and 32x4x2 for 128 and 256 processes , respectively. This results a total problem size of 3072x768x384 for 128 processes and 6172x768x384 for 256 processes. The number of iterations of the program is set to 20 so as the execution time is sufficient for measuring the various parts of the program but not very demanding in HPC resources.

In the mixed mode versions, OpenMP parallelism is applied on the first dimension of the local array. In addition, threads replace the processes that lie on the first dimension of the process grid so the number of the processes in Y and Z dimension is kept constant. In order to have the same problem size in all versions, as the number of processes of X dimension is reduced, the first dimension of the local array is increased.

Master Only

There is not much difference between pure MPI and Master Only code. The only modification made to the code is the insertion of OpenMP directives before the loop of the Jacobi calculation and the loop that performs the calculation of the delta value. Those are the only parts of the program that are executed in parallel by OpenMP threads. Thus, this mixed mode scheme provides the less effort of all the other schemes to be implemented.

The great similarity of Master Only code with the pure MPI code is also depicted in the resulting performance of those two codes on 128 processors(Figure 4.25). When the mixed mode code is run using one thread per process achieves the same performance with the pure MPI code. However, when more threads per process are used, master only code is outperformed by the pure MPI code.

The time spent on parts of the code where OpenMP directives are added (Jacobi computation and calculation of delta value) is slightly reduced. Although the iterations of these loops are increased as more threads per process are used, the calculation of the new values of the local array and the calculation of the delta value are performed faster. Furthermore, the reduced number of processes involved in the all-reduce communication is also a reason of the decrease in the time needed for the calculation of the delta value. However, the time needed to perform halo swapping is increased when fewer processes are involved in point-to-point communications and the total execution time is increased.

The profiles of the mixed mode code show that the total sent message size of each process, and consequently the average message size, are increased. This was expected because by increasing the size of the first dimension of the local data, the halo data of each process is also increased. The average message size becomes about two times larger every time that the number of the MPI processes of each mixed mode version is halved. Furthermore, in all cases the average message size is above 128,000 bytes which is the maximum message size that eager protocol can handle. So, the increase in the execution time comes from the increased communication load of each process and not from the protocol that MPICH library uses to deliver the messages.

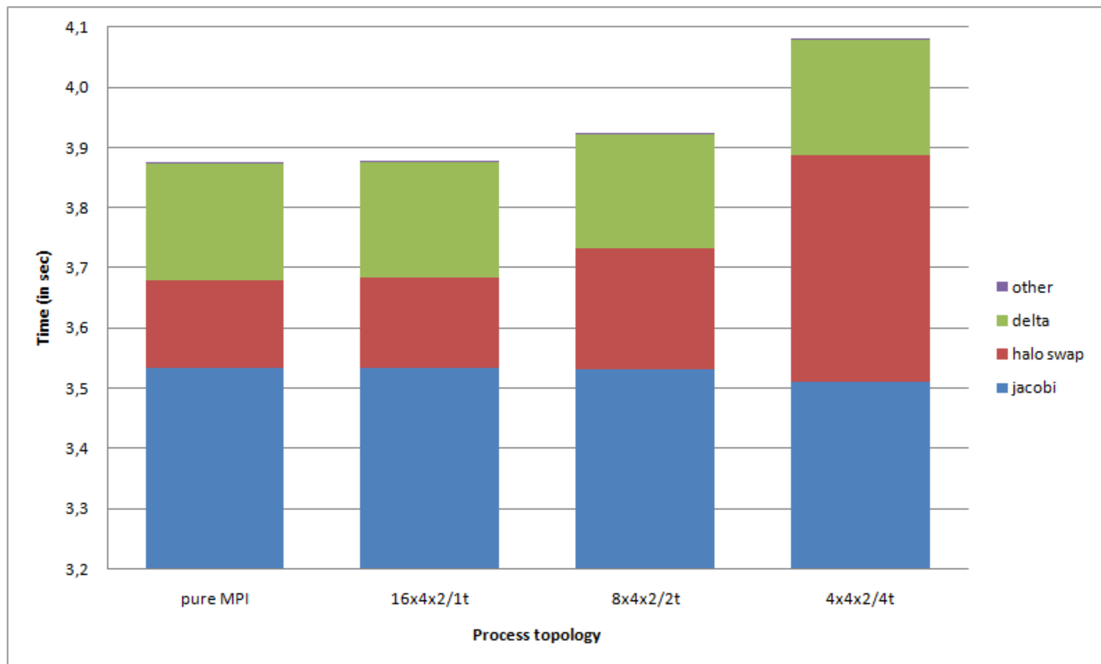

Figure 4.25: Execution time in seconds of Pure MPI and Master Only versions using 128 processors.

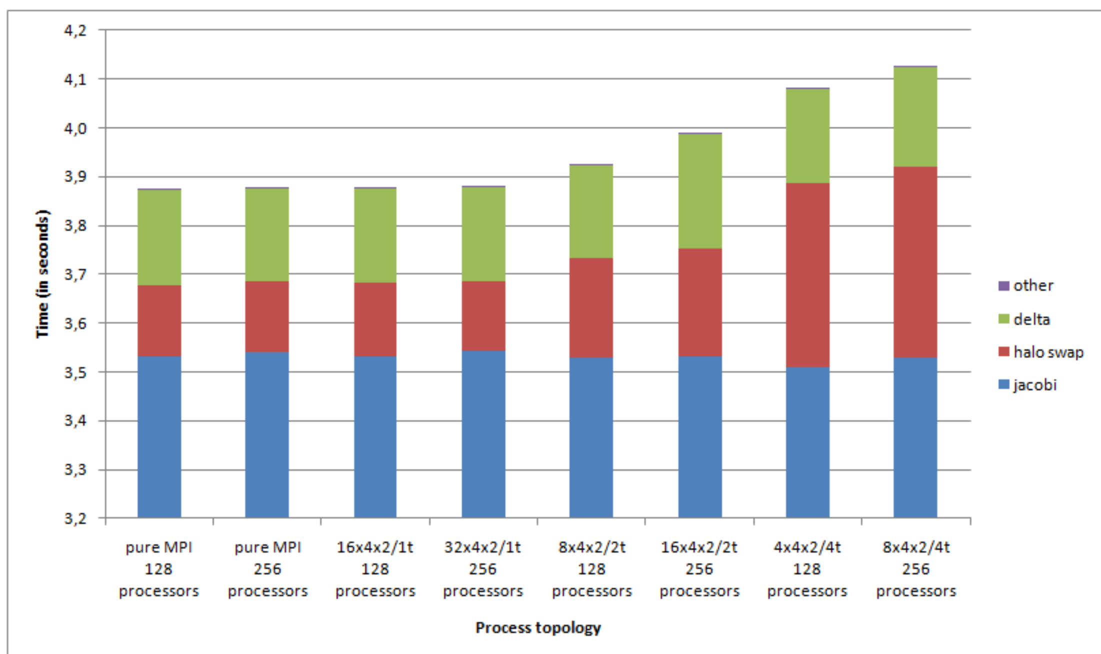

Figure 4.26: Comparison of the execution time of Pure MPI and Master Only versions on 128 and 256 processors.

Although the total problem size is increased when the pure MPI code is run using more

processors, the size of the local data remains the same. So, as it is expected, the pure MPI code has almost identical performance when it is run on 256 processors. The time spent on the calculation of delta time is not increased on pure MPI code despite the increase of the number of processes which are involved in the all-reduce communication.

Master only code has almost the same behaviour (Figure 4.26) with some slight differences in the timing of the various parts of the code. In some cases, the time spent on halo swapping is greater than the equivalent time on 128 processors. Furthermore, when master only code is run with two threads per process, the measured delta time is about 0.03 seconds greater than the equivalent time when the code is run on 128 processors. The cause of this can be identified in the timings produced by the successive runs of the same version. In this cases, one of the three runs produces very different results than the other two. So, it is possible that these insignificant differences are produced by the fluctuation of the timings of the program.

Funnelled

The funnelled code that is provided for the investigation of mixed mode programming on this project has some differences from the original code that is described and used in [15]. In this code, there is no explicit barrier and "omp single nowait" clause. Instead of using these synchronisation mechanisms, every thread other than the master thread is working on the inner elements of their part of the local array while master thread performs halo swapping and the Jacobi computation on its part of the local array and on the elements that lie on the halo regions of the array.

As master thread has more work to do than the rest of the threads and only this thread performs the halo swapping, the timings measured by master thread are used for creating the graphs and analysing the performance of the code. Besides, the time of the other threads spent on the Jacobi step is less than the equivalent time of the master thread.

The performance of funnelled code on 128 processors is presented in Figure 4.27 along with the performance of the pure MPI code. The comparison of the performance of the pure MPI code and the funnelled code using one thread per process depicts the difference in the implementation of some parts of the code. At first, the Jacobi step of the funnelled code is about 0,25 seconds slower than the equivalent part of the pure MPI code. The difference between those two codes is that the pure MPI always works on contiguous elements of the arrays and achieves better cache utilisation, whereas in funnelled code, the Jacobi computation is initially applied on the inner elements of the local array and then on the halo elements of each dimension separately. This is also illustrated by the figures of the hardware counters which are produced by the runs of those two codes (Table 4.35).

However, the time spent on halo swapping on funnelled code is reduced by one third of the equivalent time on pure MPI code. This is a result of using asynchronous point-to-point communications in funnelled code whereas pure MPI code (and the codes of the other two schemes) uses synchronous communications to perform halo swapping. An

assumption about this choice is that asynchronous communications were used to overlap communications with Jacobi computation which was not used for this project because it would be impossible to accurately time these two distinct parts of the program.

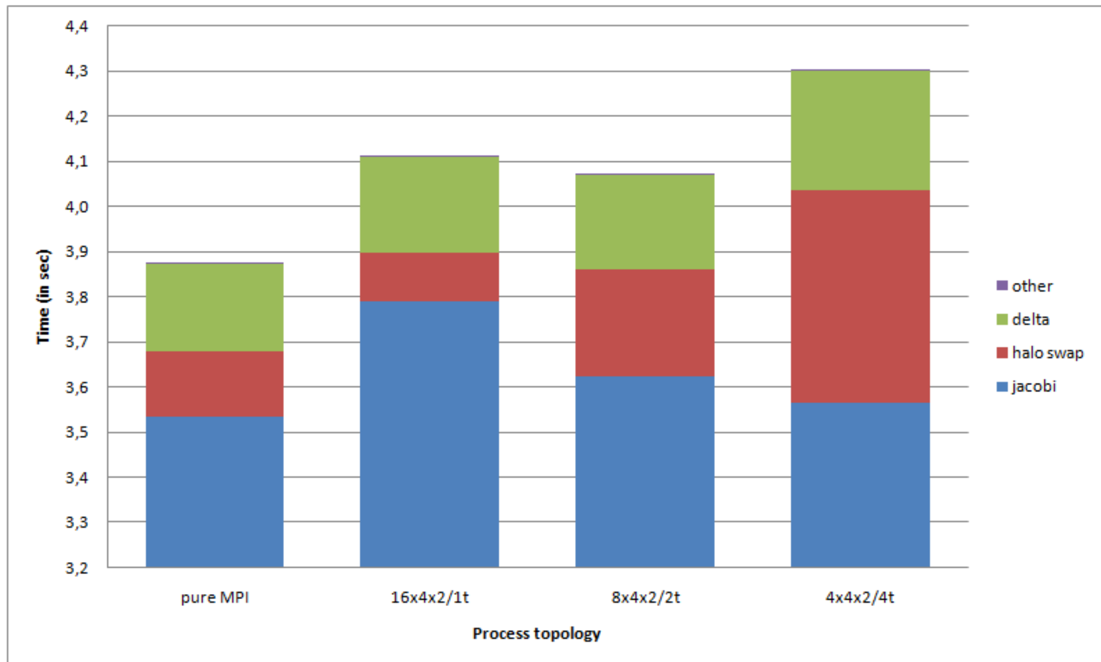

Figure 4.27: Execution time in seconds of Pure MPI and Funnelled versions using 128 processors.

Metric	Pure MPI		Funnelled	
TLB utilization	2616.89 refs/miss	5.111 avg uses	609.07 refs/miss	1.190 avg uses
D1 cache hit,miss ratios	99.5% hits	0.5% misses	98.8% hits	1.2% misses
D1 cache util. (misses)	213.32 refs/miss	26.666 avg hits	82.75 refs/miss	10.344 avg hits

Table 4.35: Hardware counter metrics for pure MPI and funnelled codes.

The delta calculation time of those two codes is very similar. Even when funnelled code is run using two or four threads per process. The amount of the execution time spent in this function depends on two factors, the OpenMP overheads introduced in the program by the parallel region that lies in this function, and the number of the processes that are involved in the all-reduce communication. When more threads per process are used, the OpenMP overheads introduced by the implicit synchronisation of the threads at the start and the end of the parallel region are increased. However, as the number of the processes involved to the collective communication is reduced, the time needed for this communication to complete may also be reduced. These changes are not measured separately in the code. So, the change of these two factors is a possible explanation about the slight increase of the delta calculation time (about 0.06 seconds) when the program is run with four threads per process.

Similarly to the master only code, the Jacobi step time is reduced as more threads per process are used, but it never becomes less than the equivalent time of the pure MPI code. However, what is really reduced is the Jacobi calculation time of the master thread. The Jacobi time of the rest of the threads is the same in both mixed mode versions. There is no obvious reason, as the master thread has more halo data to work on because X dimension is increased along with the number of the threads per process. Furthermore, halo swapping in the funnelled code needs more time to be completed when the program is run using fewer processes. The reason is the same as in the case of the master only code; the average message size per process is increased and requires more time to be delivered.

Even though the complexity of the program is increased in order to reduce the time that the threads other than the master thread are idle, the total performance of all mixed mode versions of the funnelled code is worse than the performance of pure MPI code. Even master only code outperforms the funnelled code. The main reason is the uneven work distribution among the threads. Although, the local data is split equally across the threads, master thread has to work also on the region elements of the entire array.

The funnelled code provides the capability to the programmer to define explicitly the size of the X dimension that will be assigned to the master thread. So, experiments were perform in order to reduce the work load of the master thread in such way that the time spent by the rest of the threads on the Jacobi calculation would be equal to the time that master thread needs to perform the halo swapping and the Jacobi computation on both the elements of its part of the array and the elements of that lie on the halo regions of

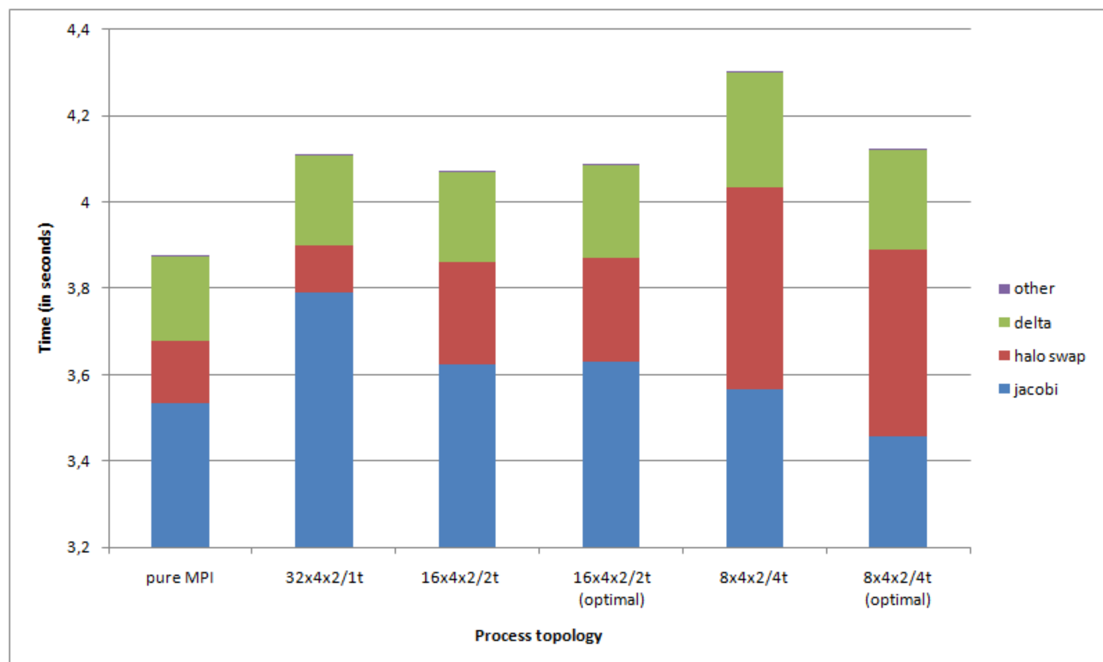

Figure 4.28: Execution time in seconds of Pure MPI and optimised Funnelled versions using 128 processors.

the array.

The results of this attempt are presented in Figure 4.28. The greatest improvement is achieved for four threads per process where the execution time was reduced by 0.18 seconds. The size of the X dimension that is assigned to the master thread is 168 for two threads per process and 127 for four threads per process. Even with this change on the work distribution across the threads that helps to reduce the Jacobi time of the master thread, funnelled code is outperformed from both pure MPI and master only codes which is mainly caused to the increased time needed for halo swapping.

When the funnelled code is run on 256 processors, as it was expected, the performance of the program remains exactly the same. The difference in the timings of the various mixed mode versions is not more than two hundredths of a second which mainly comes from the fluctuation in the timings of the program.

Multiple

This code is the most complicated of the four Jacobi codes. A part of the local array is assigned to each thread and the halo swapping is performed by each thread independently. There were many implementations of the multiple code. The only working code does not use a derived datatype for the halo swapping as the pure MPI code and instead of the derived datatype two routines for packing and unpacking the halo data into a buffer are used. This extra work has an impact in the performance of the multiple code which is presented in Figure 4.29.When the multiple code is run on 128 processors using one thread per process, all the parts of the program except for the halo swapping take the same time with the pure MPI code to complete. The packing and unpacking of the halo elements into/from a buffer increases the time spent on halo swapping by 0.1 seconds.

The timing of the halo swapping and the Jacobi computation is performed for each thread separately. The time that is presented in the graph for those parts of the program is the average of the times that are recorded from all the threads. Moreover, the time marked as “other” represents the difference in time between the average time needed for halo swapping and the Jacobi computation and the actual time that took the slowest thread to complete these tasks and made the faster threads to wait at the end of the parallel region.

The best performance of multiple code is achieved when two threads per process are used. The time spent on the Jacobi computation is 0.12 seconds less than the equivalent time of both pure MPI code and the multiple code with one thread per process. The halo swapping is slightly improved when more threads per process are used but it is never better than the halo swapping time of the pure MPI code. Except for the packing and unpacking routines there is another factor that affects the performance of the MPI communications; there is not a high performance MPI implementation on HECToR for multiple thread safety level and for this reason a different MPICH library is used during the linking of the code which may degrades the performance of the program.

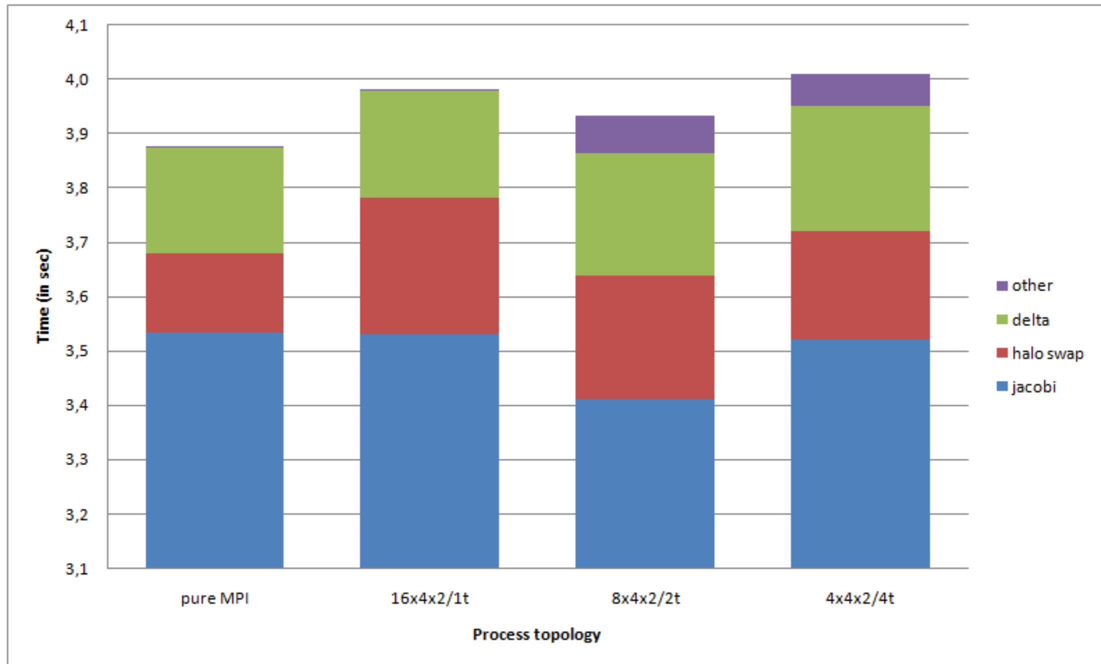

Figure 4.29: Execution time in seconds of Pure MPI and Multiple versions using 128 processors.

For two threads per process, the time spent on halo swapping on the master thread is greater than the equivalent time on thread one by almost 0.15 seconds. But the Jacobi computation time of the master thread is about 0.05 seconds less than the Jacobi time of thread one. When the number of threads is increased to four, these differences between the master thread and the rest of the threads are increased to 0.25 and 0.3 seconds respectively and this is the reason that the average time for halo swapping and Jacobi computation is increased producing worse performance.

Furthermore, when more threads per process are used, the delta calculation time is slightly increased. This routine is the same in all the codes so there is not any specific reason for this increase except for the fact that the code is linked with the non high performance MPICH library and the time needed by the all-reduced communication is not reduced (less processes are involved to this communication) to compensate the OpenMP overheads that are increased by the use of more threads.

This code has the same scalability with the previous three Jacobi codes. Although the number of processors is doubled (from 128 to 256) the performance is almost the same (Figure 4.30). The difference in the various timings of the program is negligible and is not a strong evidence of change of the performance when more processes are used.

The sophistication of the multiple code was not enough to achieve better performance than the pure MPI code. Although the Jacobi calculation of all the mixed mode versions of the multiple code was faster than the Jacobi computation of the pure MPI code, the main reason for this bad performance is the slow halo swapping. There are two possi-

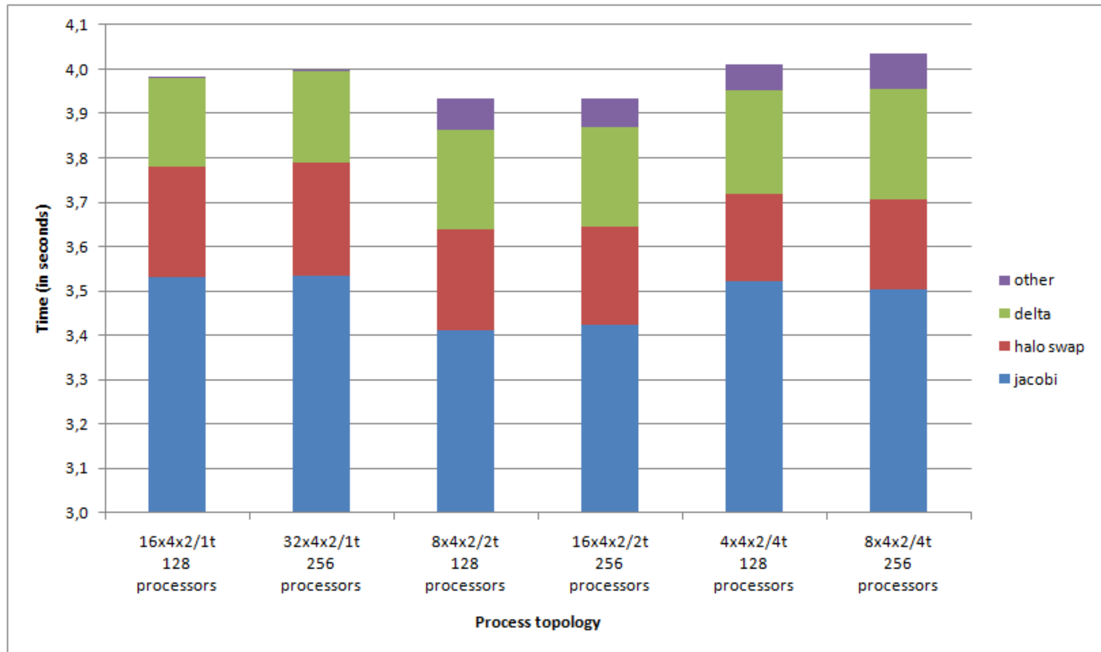

Figure 4.30: Comparison of the execution time of Multiple code on 128 and 256 processors.

ble roots of this overhead. The first cause of the increased halo swapping is the packing and unpacking of the halo data and the second cause is the non high performance implementation of the MPI library that was used for this code. If those two problems are eliminated, it is possible that multiple code will perform better than the pure MPI code.

4.6.2 Jacobi kernel on Cray XT6 system

The same Jacobi codes were run on the Phase IIb machine of HECToR. The problem size per process is reduced for this system in order the size of the local array to be lower than the maximum size that C compiler can handle when the program is run using 24 threads per process. So, for the pure MPI version, the local array has dimensions 144x144x144. The codes were run on 192 and 384 processors. The process topology of the pure MPI versions has dimensions 24x4x2 for 192 processors and 48x4x2 for 384 processors producing a total problem size of 3456x576x288 (about 4.27 GB) and 6912x576x288 (about 8.54 GB) respectively. In order to compensate the decrease in the execution time caused by the reduced problem size of each process, the iterations of the program are increased from 20 to 50.

Master Only

Pure MPI and master only codes were run on the Cray XT6 system and their performance is presented in Figure 4.31. The master only code has almost the same execution

time with the pure MPI code when it is run using one thread per process. Similarly to the Cray XT4 system, when the number of process is reduced and replaced with OpenMP threads, the time needed for halo swapping is increased and become even 20 times greater than the equivalent time of pure MPI when 24 threads per process are used. The main reason for this increase is the larger messages that are exchanged during halo swapping.

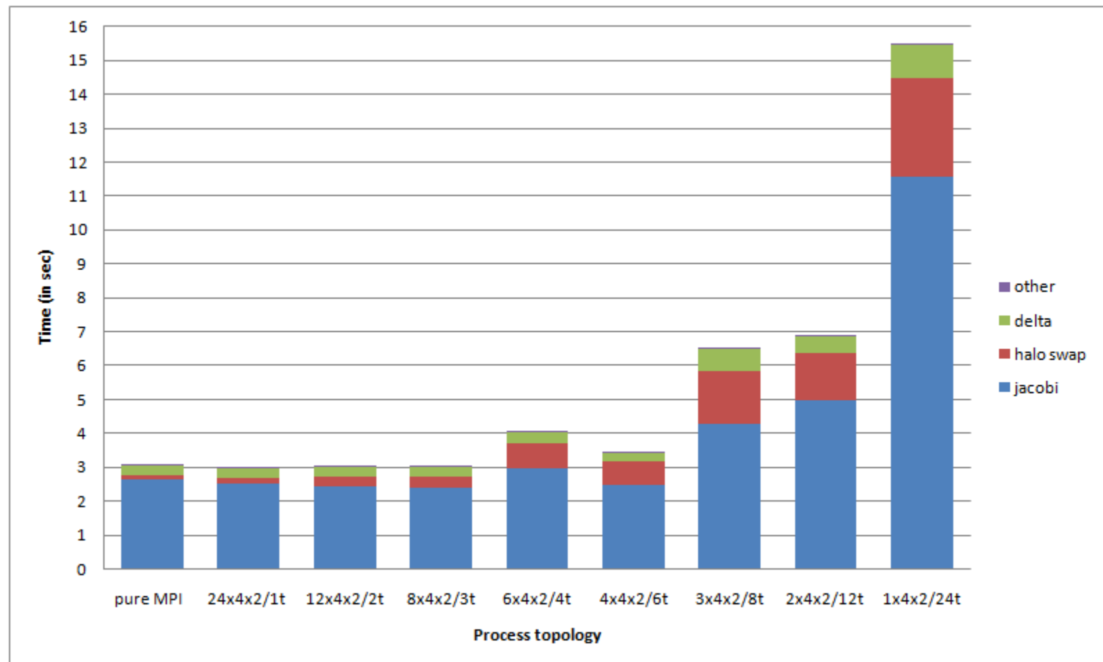

Figure 4.31: Execution time in seconds of Pure MPI and Master Only versions using 192 processors.

The overall performance of the master only code is almost equal or slightly better than the performance achieved by the pure MPI code for two and three threads per process, despite the increase of the halo swapping time. The reason is the reduce of the Jacobi computation time and the delta calculation time of these mixed mode versions which compensate the increase of the halo swapping time. Moreover, the reduce of both Jacobi computation and delta calculation times shows that the reduce of the later time mainly comes from the faster execution of the nested loop which is parallelised across the threads of each process.

For four threads per process, the performance of the program is significantly worse and gets better again when the program is run using six threads per process. The time spent on each part of the code is increased making the execution of the program to be about one second slower. It is one of the runs where the processes of the node cannot be divided equally across the NUMA regions of the node so all the processes lie on the fist six-way NUMA region and the threads that are spawn by each process are placed in different regions. Thus, the shared data of the threads lie on different NUMA region where the access time of this data is higher and this impairs the performance of the

program. Unfortunately, the use of hardware counters was not feasible on Phase IIb at the time that this report was written, in order to verify the previous assertion.

When the program is run with six threads per process, the Jacobi computation time and the delta calculation time are lower than the equivalent times of the pure MPI code, but the halo swapping time is about six times higher which results in a slightly poorer performance than the performance of pure MPI code. For 8, 12 and 24 threads per process, where again the number of processes is not divided by the number of NUMA regions and the task affinity is not possible, the execution of the program is considerably slower (up to five times) than the rest mixed mode versions and the pure MPI code.

Like in the case of the Cray XT4 system, pure MPI and master only codes were also run on 384 processors in order to check their scalability. The resulting performance was almost the same with insignificant disparity of the timings which was not more than 0.02 seconds. Again, this disparity is caused by the fluctuation of the execution times recorded in successive runs of the same version of the program. There is no evidence that the increase of the processors affected the performance of the various parts of the program.

Funnelled

The characteristics of the codes that were identified when it was run on the Cray XT4 system, such as asynchronous communications, bad cache utilization, have also the same impact in the performance of the program when it is run on the Phase IIb engine of HECToR. Using one thread per process to run funnelled code, point-to-point communications needed for halo swapping are performed faster than in the case of the pure MPI code (Figure 4.32). However, the use of asynchronous instead of synchronous communications is not as beneficial as it was when the same code was run on Phase IIa system. However, the poor cache utilization introduces the same overheads in the execution time of one threaded funnelled code.

The time spent on the Jacobi computation is reducing when the number of threads per process are increasing from one to eight and this time is always less than the equivalent time of the pure MPI code, except for the case of four threads where the same problem with the task affinity occurs which is described in the analysis of the master only code. However, the lower Jacobi computation time cannot compensate the significant increase of the halo swapping time which results in greater execution times than the pure MPI code.

The work distribution is equal among the threads. When the number of threads is less than eight, the time spent by the master thread on the Jacobi computation is always more than the equivalent time of the rest of the threads, which was expected due to the fact that master thread has more work to do performing the Jacobi update to the halo elements of the local array.

However, when the program is run using more than eight threads per process, the Jacobi

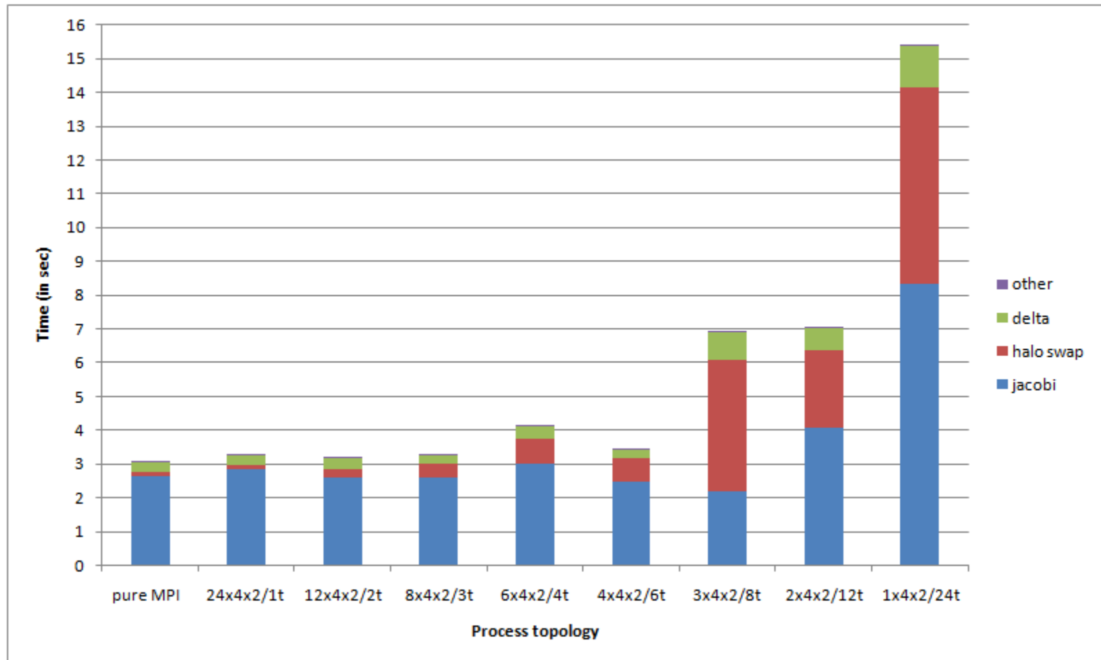

Figure 4.32: Execution time in seconds of Pure MPI and Funnelled versions using 192 processors.

time of the master thread is below the average time that is spent on this function by all the threads. But the sum of the Jacobi computation time and the halo swapping time of the master thread is higher than the maximum Jacobi computation time that is recorded during the execution of the program. Thus, all the threads are waiting the master thread to finish its work at the end of the parallel region. This effect was not so severe on the Cray XT4 system because the maximum number of threads per process was four. So, the same effort was made on this system to find a work distribution that produces the optimal performance of each funnelled mixed mode version.

After experimenting with different workloads for the master thread, the best performance for each mixed mode version was achieved by assigning to the master thread the workload that is presented in Table 4.36. Although the Jacobi computation time of the master thread is reduced, there is a significant increase in time that is spent on routines other than the Jacobi computation, halo swapping or delta calculation.

However, when funnelled code was run using three threads per process, the total execution time is less than the execution time of pure MPI code (Figure 4.33). So, it is the only case where funnelled code is faster than the pure MPI code, and for this run, the size of the X dimension that is assigned to master threads is 96 instead of 144. For more threads per process, reducing the part of the local array that is assigned to the master thread resulted in significant reduction of the execution time which is up to a second for 24 threads per process.

The scalability of funnelled code was checked by running the code on 384 processors.

Decomposition	X size of master thread
48x4x2/1t	144
24x4x2/2t	119
16x4x2/3t	95
12x4x2/4t	67
8x4x2/6t	29
6x4x2/8t	20
4x4x2/12t	14
2x4x2/24t	11

Table 4.36: Optimal work distribution of X dimension on master thread for funnelled code

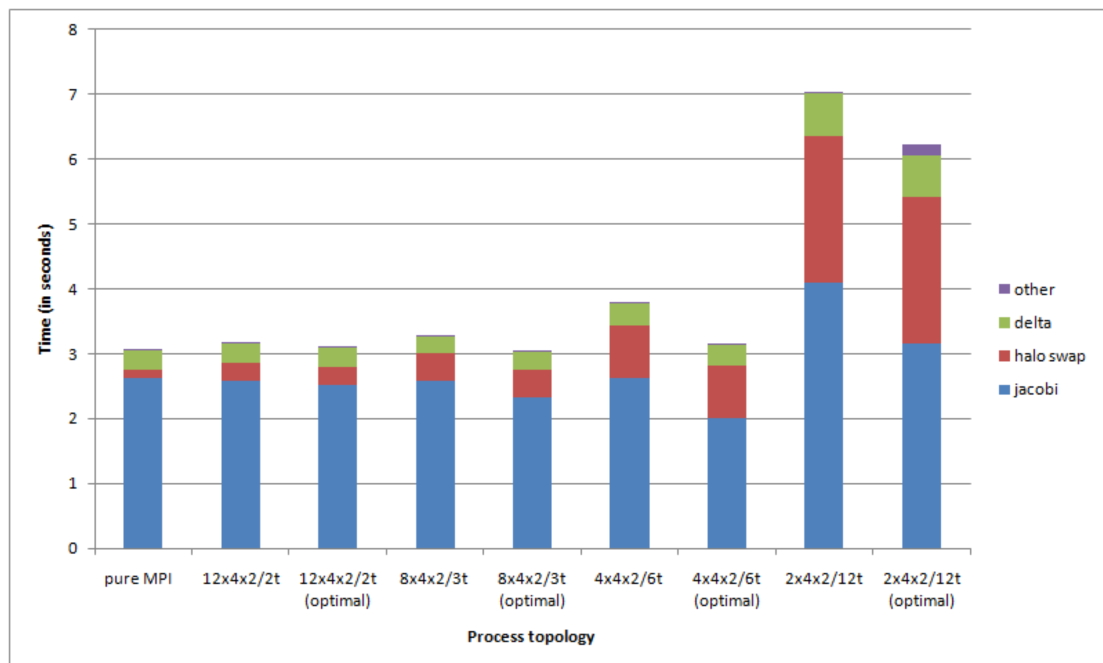

Figure 4.33: Execution time in seconds of Pure MPI and optimised Funnelled versions using 192 processors.

The minor differences in the execution times of the program between the runs on 192 and 384 processors shows that the program scales almost perfectly even when each process spawned 24 threads.

Multiple

Similarly to the Cray XT4 system, there is not any high performance implementation of the MPI library for multiple thread safety level on the Cray XT6 system. Thus, a different library from the one that was used for the previous codes is linked to the program.

The performance of the multiple code compared with the equivalent performance of the pure MPI code is presented in Figure 4.34. The lowest execution time was recorded when the program was run using two and three threads per process which is about 0.04 seconds faster than the equivalent time of the pure MPI code. For those thread counts, the average Jacobi computation time is reduced and the average time needed for halo swapping is almost the same. However, the difference between these average values and the time needed by the slowest thread to execute this tasks is increased (timing marked as “other”).

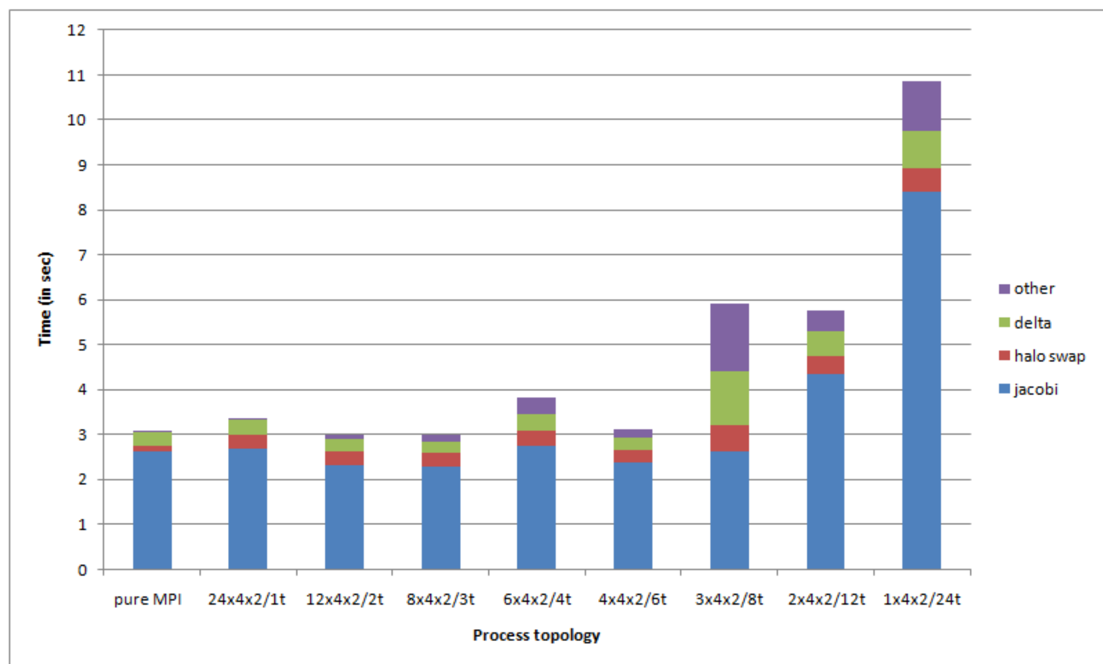

Figure 4.34: Execution time in seconds of Pure MPI and Multiple versions using 192 processors.

When the number of processes is equally divided by the number of NUMA regions (for 2, 3, and 6 threads per process) the delta calculation time is reduced as the number of threads is increased. The fact that this time is affected by the task affinity shows that the main reason of this reduction is the faster execution of the loop that lies inside this routine by the threads of each process. This also presents the importance of the task affinity over the NUMA regions in order the program to achieve a good performance.

For more than six threads per process, the execution time of the multiple code is increased substantially. The total time of the program is roughly doubled for eight and 12 threads per process and tripled for 24 threads per process. Although the work distribution across the threads is even and each thread has the same tasks to perform (Jacobi computation, halo swapping), the measured times for each thread show that there is a wide variation in the Jacobi time and halo swap time across the threads. In particular, the Jacobi computation time is suddenly increased for thread ids above five and the master thread has the lowest Jacobi computation time.

However, the master thread always spends five to eight times more time than the rest of the threads in halo swapping. Similarly to the Jacobi computation time, the halo swap time among the rest of the threads increases along with the id of the threads. There is not any good explanation for the variation in the halo swapping time between master thread and the rest of the threads and it is possible caused by the way that the non-high performance implementation of the MPI library handles the communications of the master thread.

But the cause of the variation in the Jacobi computation time can be identified with a careful look to the times of each thread especially for 24 threads per process. In Figure 4.35, it is more clear how the placement of the threads on to the NUMA region affects the Jacobi computation time. When the threads lie on the same NUMA region with the process that spawns them, the Jacobi execution time is kept low because they access the memory faster than the threads that lie on different NUMA region. This increase is easily noticed when multiple code is run using 24 threads per process. In this case, only master thread and the first five spawned threads lie on the same NUMA region, so the memory access is increase for the rest of the other threads.

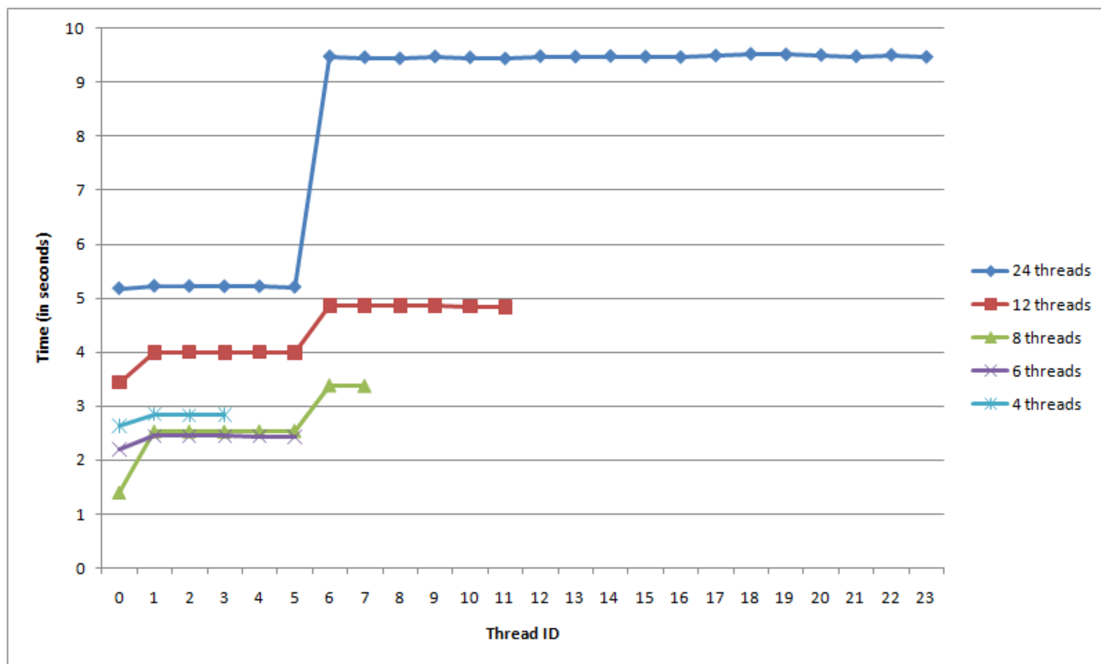

Figure 4.35: Jacobi computation time of each thread of Multiple code on 192 processors.

The master thread is always faster than the rest of the threads of the same NUMA region because the data is already loaded to the cache memory of the processor before spawning the rest of the threads. For six threads per process, the programmer can manually set the task affinity using the proper parameter of the job launcher, so all threads lie on the same NUMA region and they spent almost the same time on the Jacobi computation.

As it was expected, there was not much difference in the performance of the multiple code when the number of processors was doubled. Both the absolute performance and the trend of the execution times, when increasing the threads per process, are exactly the same with the performance of the code on 192 processors.

The gains in performance of mixed mode programming on the Cray XT6 system are insignificant. The codes of every mixed mode scheme managed to achieve slightly better performance than the pure MPI code. This improvement was not more than 2% in the total execution time and it was achieved due to the reduction of the Jacobi computation time as more threads per process were used. In contrast, all mixed mode versions were outperformed from the pure MPI code on the Cray XT4 system. So, the additional programming effort for this case is not reflected in the resulting performance.

4.6.3 Memory usage of Jacobi kernel

Even though there is no replication of data across the MPI processes, mixed mode programming helped to reduced the program memory usage per node. Figure 4.36 presents the result of the memory "high water mark" per process recorded by CrayPAT multiplied by the number of processes per node for the master only code.

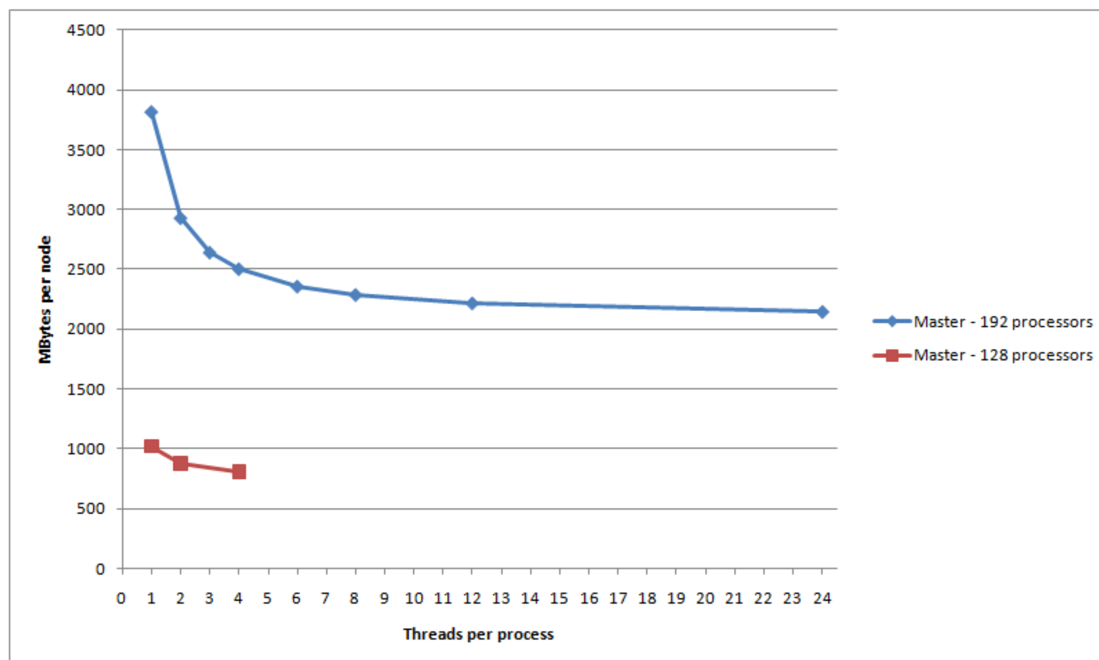

Figure 4.36: Memory high water mark per node of master only code on both systems.

The reduction of the memory usage per node that is reported for this code when mixed mode programming is applied, is the lowest of all the benchmark codes that were included in this project. On the Cray XT4 system, although the problem size per process of the pure MPI version is about 54MB, the average memory usage per process is

255MB. This was expected due the use of multiple arrays which are necessary for the execution of the program, such as the arrays that hold the old and the new values of the problem data. However, by reducing the number of processes per node and increasing the problem size of each process, the memory usage per node was reduced by more than 20% on the Cray XT4 system which is very small reduction in comparison with the equivalent reduction on other benchmark codes.

The existence of more processors per node on the Cray XT6 system provided the opportunity to use fewer processes for running the program and thus each process held a greater share of the problem data. For this reason, the gains from mixed mode programming were higher on this system, and the memory usage per node of the mixed mode versions using 24 threads per process was about 1.8 times lower than the memory usage of the equivalent pure MPI version.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This project investigated the performance of mixed mode programming on HECToR by running five benchmark codes on both systems of HECToR. These codes come from different scientific areas and thus the investigation of mixed mode programming covers a wide variety of applications. The performance of the pure MPI version was the main criterion of evaluating the performance of the mixed mode versions of each code.

This investigation encountered most of the cases where mixed mode programming would be beneficial for the performance of the program which are described in Chapter 2. In almost all the tested benchmarks, there was at least one mixed mode version which was faster than the equivalent pure MPI version. However, in most of the cases where pure MPI performed better than the mixed mode versions of a code, the main reason was the overheads which were introduced to the program by the lack of sufficient OpenMP parallelism, so the replacement of the MPI processes with OpenMP threads does not achieve the same parallelism with pure MPI. The amount of these overheads depends on the sophistication of the implementation, the work load of each process and the number of OpenMP threads per process.

Nevertheless, mixed mode programming reduced the load imbalance of the pure MPI versions, especially on the CPMD code where for small problem sizes and large number of processes, this load imbalance was severe. Furthermore, the use of fewer MPI processes per node improved the communications across the processes of the most of the benchmark codes, except for the Jacobi code, where the time needed for the MPI communications to complete was increased.

Moreover, the execution time of most of the parallelised functions was reduced when more than one thread per process was used, even though the work of those functions was increased. However, the OpenMP overheads that were introduced to the program were insignificant even for 24 threads per process, but these overheads are not the same when different compilers are used for the build of the same code.

Furthermore, when mixed mode programming was applied on the BQCD code, the reduction of the MPI processes changed the shape of the process grid which affected the performance of the program. The resulting performance mainly depended on the number of processes that lay on each of the dimensions of the new process grid. So, there is no a clear evidence that this helped or worsened the performance of the program.

Mixed mode programming reduced the memory usage per node of all the benchmark codes, especially on the Cray XT6 system where the number of processes per node of the pure MPI version is 24. The increase of the memory usage per MPI process was not proportional to the reduction of the number of processes per node which resulted in the reduction of the total memory usage per node.

During the investigation of mixed mode programming, there was the opportunity to compare the performance of the programs between the two systems of HECToR. This comparison revealed that the main bottleneck of the XT6 system is the interconnect mechanism. SeaStar2+ could not service efficiently the simultaneous requests of 24 processes. This the reason that the MPI communications' time of the pure MPI version of the CPMD code on XT6 system is more than eight times higher than the equivalent time on the XT4 system. Furthermore, mixed mode programming was more beneficial for the MPI communications on the XT6 system than it was on the XT4 system because the reduction of the MPI processes reduced the congestion on the network.

Furthermore, the Jacobi code provided a clear view about the importance of the task affinity on the NUMA regions of each node on the Cray XT6 system. The threads that lay on different NUMA regions from the region of master thread performed the Jacobi computation almost two times slower than the master thread. For this reason, only mixed mode versions with 2, 3 and 6 threads per process ensure task affinity on the XT6 system and the best performance of mixed mode programming on this system was usually achieved by these thread counts.

5.2 Future Work

The Phase IIb machine of HECToR will be further upgraded this year. The GEMINI interconnect is going to replace the current Cray SeaStar2+ interconnect chip. A suggestion for future work is to perform the same runs on the upgraded system of HECToR. These runs will reveal if the better interconnect mechanism will further improve the performance of the mixed mode versions of the programs compared with the performance of the equivalent pure MPI versions, or it will only benefit the pure MPI versions of the program, making the use of mixed mode programming imperative only for extreme cases.

Furthermore, there was the intention to use Integrated Forecast System code for the investigation of the project. However, this code was not available by ECMWF[25] on time in order to be included to the project. The large size and the great complexity of this code would provide useful information about the effectiveness of mixed mode

programming on this kind of program, so a suggestion for future work is to port and run this code on HECToR. However, the use of this code would require a great amount of HPC resources.

Finally, the limitation on HPC resources and the time limitations did not allow an in-depth investigation of mixed mode programming using different compilers. So, the last suggestion for future work is to investigate in more detail the difference in performance of the mixed mode versions when the same programs are built by different compilers. However, not all the benchmark codes work with all the available compilers on HECToR, for instance, the CPMD code works only with PGI compiler, but the rest of the benchmarks can be compiled by more than one compiler.

Appendix A

Experimental Data

This chapter contains the timings from the benchmark code that were used for creating the graphs of Chapter 4.

A.1 CPMD

A.1.1 CPMD on Cray XT4 system

Threads/process	1 mole	128 moles	256 moles
pure MPI	22.73	201.07	973.39
2 threads	14.08	119.84	997.52
4 threads	9.43	81.7	1322.92

Table A.1: CPMD benchmark execution time (in seconds) using 64 cores.

Threads/process	1 mole	128 moles	256 moles	384 moles	512 moles
pure MPI	11.37	93.01	337.51	810.80	1561.41
2 threads	7.88	81.26	318.24	791.6	1576.62
4 threads	9.4	82.17	379.55	1032.7	1999.54

Table A.2: CPMD benchmark execution time (in seconds) using 256 cores.

A.1.2 CPMD on Cray XT6 system

Threads/process	1 mole	128 moles	256 moles
pure MPI	95.73	229.68	715.25
2 threads	50.93	174.48	777.27
3 threads	24.24	187.72	772.11
4 threads	36.58	215.97	1014.77
6 threads	18.56	218.86	1112.17
8 threads	28.21	339.41	1660.94
12 threads	27.81	346.90	1877.18
24 threads	39.13	624.24	3497.42

Table A.3: CPMD benchmark execution time (in seconds) using 96 cores.

Threads/process	1 mole	128 moles	256 moles	384 moles	512 moles
pure MPI	48.77	206.49	229.68	1093.21	1734.69
2 threads	24.94	94.58	174.48	1006.34	1744.17
3 threads	17.02	85.31	187.72	789.32	1482.45
4 threads	43.15	108.04	215.97	857.76	1700.41
6 threads	18.86	94.04	218.86	1049.72	2140.32
8 threads	21.01	105.76	339.41	1325.87	2806.50
12 threads	15.89	134.52	346.90	1535.45	3346.11
24 threads	17.47	172.97	624.24	2775.78	over 3600.00

Table A.4: CPMD benchmark execution time (in seconds) using 256 cores.

A.2 BQCD

A.2.1 BQCD on Cray XT4 system

Threads/process	64 processors	256 processors	1024 processors
pure MPI	75.57	18.86	6.69
2 threads	73.88	18.68	5.67
4 threads	76.5	19.79	5.69

Table A.5: BQCD benchmark execution time (in seconds) on 24x24x24x48 lattice.

Threads/process	256 processors	1024 processors
pure MPI	301.32	81.76
2 threads	299.89	82.38
4 threads	311.02	91.23

Table A.6: BQCD benchmark execution time (in seconds) on 48x48x48x96 lattice.

A.2.2 BQCD on Cray XT6 system

Threads/process	192 processors	384 processors	768 processors	2304 processors
pure MPI	39.62	29.01	22.38	19.51
2 threads	34.2	23.02	16.94	12.99
3 threads	30.52	20.02	15.59	10.04
4 threads	30.26	19.3	12.95	9.5
6 threads	31.41	17.75	12.61	9.1
8 threads	35.16	19.28	14.89	8.58
12 threads	32.45	19.37	11.86	8.26
24 threads	38.1	24.09	15.7	9.34

Table A.7: BQCD benchmark execution time (in seconds) on 24x24x24x48 lattice.

Threads/process	384 processors	768 processors	2304 processors
pure MPI	241.69	141.23	67.26
2 threads	240.84	134.2	58.59
3 threads	228.61	133.24	58.3
4 threads	241.26	131.49	56.32
6 threads	227.96	126.87	51.94
8 threads	250.25	140.36	58.85
12 threads	250.78	137.13	56.31
24 threads	314.12	162.46	70.37

Table A.8: BQCD benchmark execution time (in seconds) on 48x48x48x96 lattice.

A.3 SP-MZ

A.3.1 SP-MZ on Cray XT4 system

Threads/process	64 processors/class C	256 processors/class D
pure MPI	5.72	26.93
2 threads	4.17	27.36
4 threads	4.42	25.15

Table A.9: SP-MZ benchmark execution time (in seconds) compiled with PGI compiler.

Threads/process	64 processors/class C	256 processors/class D
pure MPI	6.26	28.52
2 threads	5.03	29.66
4 threads	5.49	28.56

Table A.10: SP-MZ benchmark execution time (in seconds) compiled with GNU compiler.

A.3.2 SP-MZ on Cray XT6 system

Threads/process	96 processors/class C	384 processors/class D
pure MPI	3.66	19.26
2 threads	3.08	18.46
3 threads	2.78	16.05
4 threads	3.13	15.74
6 threads	3.22	14.06
8 threads	3.92	15.69
12 threads	4.5	16.36
24 threads	6.87	26.09

Table A.11: SP-MZ benchmark execution time (in seconds).

A.4 IRS

A.4.1 IRS on Cray XT4 system

Domains/Processors	pure MPI	2 threads/process	4 threads/process
8/8	60.22	61.79	67.30
64/64	64.58	63.00	67.32
216/216	67.69	65.08	67.47
64/16	244.54	253.42	278.74
216/108	122.08	121.63	134.36

Table A.12: IRS benchmark, microseconds per zone-iteration.

Domains/Processors	pure MPI	2 threads/process	4 threads/process
8/8	4.54	5.11	6.4
64/64	8.76	9.24	11.19
216/216	13.64	14.47	17.55
64/16	32.74	36.55	45.35
216/108	24.54	27.09	34.67

Table A.13: IRS benchmark execution time (in seconds).

A.4.2 IRS on Cray XT6 system

Threads/process	216 domains/216 processors	216 domains/108 processors
pure MPI	133.05	147.55
2 threads	85.93	132.71
3 threads	78.60	129.42
4 threads	91.39	161.70
6 threads	78.15	150.79
8 threads	118.11	- - -
12 threads	122.65	247.98
24 threads	255.095	- - -

Table A.14: IRS benchmark, microseconds per zone-iteration.

Threads/process	216 domains/216 processors	216 domains/108 processors
pure MPI	25.52	29.56
2 threads	18.49	29.75
3 threads	18.68	32.11
4 threads	22.41	40.83
6 threads	23.05	44.92
8 threads	33.15	- - -
12 threads	39.9	79.97
24 threads	81.23	- - -

Table A.15: IRS benchmark execution time (in seconds).

A.5 Jacobi kernel

A.5.1 Jacobi kernel on Cray XT4 system

Decomposition	Jacobi	Halo swap	Delta	Other	Total
pure MPI	3.533353	0.145059	0.194561	0.000094	3.873067
16x4x2/1t	3.532583	0.150797	0.191692	0.000077	3.875157
8x4x2/2t	3.530217	0.201626	0.190408	0.000066	3.922317
4x4x2/4t	3.510483	0.376002	0.192636	0.000061	4.079187

Table A.16: Master Only time(in seconds) on 128 processors

Decomposition	Jacobi	Halo swap	Delta	Other	Total
pure MPI	3.540473	0.146378	0.187707	0.000082	3.874640
32x4x2/1t	3.541843	0.144088	0.192912	0.000103	3.878947
16x4x2/2t	3.533007	0.218529	0.234922	0.000078	3.986537
8x4x2/4t	3.528487	0.391072	0.205106	0.000072	4.124737

Table A.17: Master Only time(in seconds) on 256 processors

Decomposition	Jacobi	Halo swap	Delta	Other	Total
pure MPI	3.533353	0.145059	0.194561	0.000094	3.873067
16x4x2/1t	3.790343	0.107600	0.211168	0.000240	4.109350
8x4x2/2t	3.623847	0.236994	0.207469	0.000204	4.068513
4x4x2/4t	3.565823	0.469363	0.265359	0.000188	4.300733

Table A.18: Funnelled time(in seconds) on 128 processors

Decomposition	Jacobi	Halo swap	Delta	Other	Total
pure MPI	3.533353	0.145059	0.194561	0.000094	3.873067
16x4x2/1t	3.790343	0.107600	0.211168	0.000240	4.109350
8x4x2/2t	3.629320	0.240106	0.217108	0.000202	4.086733
4x4x2/4t	3.456767	0.433224	0.231819	0.001962	4.125493

Table A.19: Funnelled optimal time(in seconds) on 128 processors

Decomposition	Jacobi	Halo swap	Delta	Other	Total
pure MPI	3.540473	0.146378	0.187707	0.000082	3.874640
32x4x2/1t	3.770427	0.117687	0.221114	0.000196	4.109423
16x4x2/2t	3.628200	0.240877	0.201115	0.000212	4.070403
8x4x2/4t	3.595170	0.459299	0.281201	0.000191	4.335857

Table A.20: Funnelled time(in seconds) on 256 processors

Decomposition	Jacobi	Halo swap	Delta	Other	Total
pure MPI	3.533353	0.145059	0.194561	0.000094	3.873067
16x4x2/1t	3.531240	0.249986	0.196893	0.000257	3.978377
8x4x2/2t	3.410127	0.227603	0.225696	0.069785	3.933210
4x4x2/4t	3.511633	0.197268	0.227457	0.057818	3.994177

Table A.21: Multiple time(in seconds) on 128 processors

Decomposition	Jacobi	Halo swap	Delta	Other	Total
pure MPI	3.540473	0.146378	0.187707	0.000082	3.874640
32x4x2/1t	3.535483	0.252556	0.206948	0.000280	3.995267
16x4x2/2t	3.422360	0.222237	0.223860	0.065453	3.933910
8x4x2/4t	3.489267	0.205433	0.272806	0.079771	4.047277

Table A.22: Multiple time(in seconds) on 256 processors

A.5.2 Jacobi kernel on Cray XT6 system

Decomposition	Jacobi	Halo swap	Delta	Other	Total
pure MPI	2.630680	0.122860	0.287154	0.000105	3.040800
24x4x2/1t	2.530850	0.149855	0.297120	0.000111	2.977937
12x4x2/2t	2.417173	0.291456	0.284660	0.000120	2.993410
8x4x2/3t	2.404150	0.336758	0.254238	0.000141	2.995287
6x4x2/4t	2.962337	0.726665	0.352723	0.000152	4.041877
4x4x2/6t	2.456330	0.703379	0.257364	0.000128	3.417200
3x4x2/8t	4.263827	1.578860	0.649930	0.000130	6.492747
2x4x2/12t	4.956950	1.423477	0.469108	0.000122	6.849657
1x4x2/24t	11.554000	2.940657	0.977957	0.000119	15.472733

Table A.23: Master Only time(in seconds) on 192 processors

Decomposition	Jacobi	Halo swap	Delta	Other	Total
pure MPI	2.619807	0.121792	0.298513	0.000105	3.040217
48x4x2/1t	2.543363	0.148345	0.294702	0.000106	2.986517
24x4x2/2t	2.435857	0.269132	0.285554	0.000121	2.990663
16x4x2/3t	2.411803	0.322843	0.264725	0.000138	2.999510
12x4x2/4t	2.971213	0.717040	0.356724	0.000129	4.045107
8x4x2/6t	2.465207	0.700576	0.269584	0.000116	3.435483
6x4x2/8t	4.284873	1.557003	0.661611	0.000142	6.503630
4x4x2/12t	4.973293	1.408617	0.469267	0.000116	6.851293
2x4x2/24t	11.576533	2.989983	0.985060	0.000157	15.551733

Table A.24: Master Only time(in seconds) on 384 processors

Decomposition	Jacobi	Halo swap	Delta	Other	Total
pure MPI	2.630680	0.122860	0.287154	0.000105	3.040800
24x4x2/1t	2.851497	0.113080	0.290436	0.000501	3.255517
12x4x2/2t	2.579583	0.271210	0.312891	0.000402	3.164083
8x4x2/3t	2.584820	0.418334	0.255700	0.000461	3.259313
6x4x2/4t	3.234903	0.579013	0.422895	0.000389	4.237197
4x4x2/6t	2.618620	0.816460	0.340291	0.000413	3.775783
3x4x2/8t	2.184943	3.905747	0.826721	0.006080	6.923493
2x4x2/12t	4.091210	2.272240	0.661062	0.000365	7.024880
1x4x2/24t	8.320257	5.843370	1.202930	0.000381	15.366967

Table A.25: Funnelled time(in seconds) on 192 processors

Decomposition	Jacobi	Halo swap	Delta	Other	Total
pure MPI	2.630680	0.122860	0.287154	0.000105	3.040800
24x4x2/1t	2.851497	0.113080	0.290436	0.000501	3.255517
12x4x2/2t	2.515547	0.273120	0.304907	0.001257	3.094830
8x4x2/3t	2.325440	0.419709	0.280904	0.004733	3.035210
6x4x2/4t	2.156063	1.057153	0.431921	0.351578	3.996717
4x4x2/6t	1.998087	0.811542	0.321280	0.005516	3.136427
3x4x2/8t	1.400420	3.533787	1.311060	0.285329	6.530593
2x4x2/12t	3.159080	2.260897	0.628633	0.177528	6.226143
1x4x2/24t	7.129560	5.896207	1.224393	0.000392	14.250567

Table A.26: Funnelled optimal time(in seconds) on 192 processors

Decomposition	Jacobi	Halo swap	Delta	Other	Total
pure MPI	2.619807	0.121792	0.298513	0.000105	3.040217
48x4x2/1t	2.841233	0.116858	0.303163	0.000322	3.261573
24x4x2/2t	2.578067	0.268940	0.319054	0.000417	3.166477
16x4x2/3t	2.534927	0.423940	0.314789	0.000456	3.274113
12x4x2/4t	3.243640	0.576876	0.427896	0.000380	4.248790
8x4x2/6t	2.607580	0.836651	0.345101	0.000345	3.789677
6x4x2/8t	2.215777	3.906620	0.847544	0.005900	6.975843
4x4x2/12t	4.113140	2.265987	0.661332	0.000383	7.040843
2x4x2/24t	8.289160	5.890900	1.203837	0.000410	15.384300

Table A.27: Funnelled time(in seconds) on 384 processors

Decomposition	Jacobi	Halo swap	Delta	Other	Total
pure MPI	2.630680	0.122860	0.287154	0.000105	3.040800
24x4x2/1t	2.690373	0.314062	0.336031	0.000447	3.340913
12x4x2/2t	2.310093	0.305329	0.288948	0.091846	2.996217
8x4x2/3t	2.279917	0.305283	0.261782	0.151862	2.998843
6x4x2/4t	2.748007	0.349342	0.365400	0.368685	3.831433
4x4x2/6t	2.379150	0.274773	0.270331	0.183549	3.107803
3x4x2/8t	2.625780	0.572910	1.218397	1.481517	5.898603
2x4x2/12t	4.358500	0.379851	0.571682	0.444597	5.754630
1x4x2/24t	8.401497	0.514797	0.823249	1.123224	10.862767

Table A.28: Multiple time(in seconds) on 192 processors

Decomposition	Jacobi	Halo swap	Delta	Other	Total
pure MPI	2.619807	0.121792	0.298513	0.000105	3.040217
48x4x2/1t	2.711390	0.322539	0.322668	0.000463	3.357060
24x4x2/2t	2.225243	0.335307	0.309948	0.142805	3.013303
16x4x2/3t	2.298947	0.291384	0.262782	0.148221	3.001333
12x4x2/4t	2.763760	0.342968	0.364324	0.361048	3.832100
8x4x2/6t	2.395847	0.274589	0.268829	0.178068	3.117333
6x4x2/8t	2.626130	0.573587	1.216623	1.476869	5.893210
4x4x2/12t	4.348147	0.395815	0.575398	0.508644	5.828003
2x4x2/24t	8.376673	0.517722	0.861951	1.112821	10.869167

Table A.29: Multiple time(in seconds) on 384 processors

Bibliography

[1] Fundamental Concepts of HPC, *Parallel Architectures*, (course slides, EPCC, The University of Edinburgh, 2009).

[2] Fundamental Concepts of HPC, *Message Passing Concepts*, (course slides, EPCC, The University of Edinburgh, 2009).

[3] Fundamental Concepts of HPC, *Shared Variables and Data Parallel Concepts*, (course slides, EPCC, The University of Edinburgh, 2009).

[4] Performance Scaling on Modern HPC Architectures, *Hybrid Programming*, (course slides, EPCC, The University of Edinburgh, 2009).

[5] Shared Memory Programming, *Introduction to OpenMP*, (course slides, EPCC, The University of Edinburgh, 2009).

[6] L.A.Smith *Mixed-mode MPI/OpenMP Programming*, UK High End Computing, EPCC, Edinburgh,[online]. Available: http://www.ukhec.ac.uk/publications/tw/mixed.pdf [Accessed: 20th June 2010].

[7] R.Rabenseifner: *Hybrid Parallel Programming on HPC Platforms*, in the proceedings of the Fifth European Workshop on OpenMP, EWOMP '03, Aachen, Germany, Sept. 22-26, (2003).

[8] R.Rabenseifner, G.Wellein: *Comparison of Parallel Programming Models on Clusters of SMP*, in the proceedings of the International Conference on High Performance Scientific Computing, March 10-14, Hanoi, Vietnam, (2003).

[9] Fundamental Concepts of HPC, *Parallel Architectures*, (course slides, EPCC, The University of Edinburgh, 2009).

[10] HECToR: UK National Supercomputer Service, *HECToR Hardware*, [online]. Available: http://www.hector.ac.uk/support/documentation/userguide/hardware.php [Accessed: 30th June 2010].

[11] Performance Scaling on Modern HPC Architectures, *HECToR Hardware*, (course slides, EPCC, The University of Edinburgh, 2009).

[12] Top500 Supercomputer Sites, *Top500 List - November 2009*, [online]. Available: http://www.top500.org/list/2009/11/100 [Accessed: 29th June 2010].

[13] Cray Inc., The supercomputer company, *Cray XT6 Brochure*, [online]. Available: http://www.cray.com/Assets/PDF/products/xt/CrayXT6Brochure.pdf [Accessed: 30th June 2010].

[14] Top500 Supercomputer Sites, *Top500 List - June 2010*, [online]. Available: http://www.top500.org/list/2010/06/100 [Accessed: 30th June 2010].

[15] Michal Piotrowski, *Mixed Mode Programming on Clustered SMP Systems*, (MSc thesis, EPCC, The University of Edinburgh, 2006).

[16] DEISA: Distributed European Infrastructure for Supercomputing Applications, *Benchmarking & Benchmark Suite*, [online]. Available: http://www.deisa.eu/science/benchmarking [Accessed: 12th February 2010].

[17] CPMD Consortium, *CPMD Consortium page : entry page*, [online]. Available: http://www.cpmd.org/ [Accessed: 1st July 2010].

[18] Thomas Streuer and Hinnerk Stuben: *Simulations of QCD in the Era of Sustained Tflop/s Computing*, NIC Series Volume 38, [online]. Available: http://opus.kobv.de/zib/volltexte/2008/1091/pdf/ZR_07_48.pdf [Accessed: 3rd July 2010].

[19] Yoshifumi Nakamura and Hinnerk Stuben: *BQCD manual - June 2010*, [online]. Available: https://www.zib.de/stueben/bqcd/bqcd_manual.pdf [Accessed: 3rd July 2010].

[20] NAS: NASA Advanced Supercomputing Division, Jill Dunbar, *NAS Parallel Benchmarks Changes*, March 29, 2009, [online]. Available: http://www.nas.nasa.gov/Resources/Software/npb.html [Accessed: 3rd July 2010].

[21] Rob F. Van der Wijngaart, Haoqiang Jin, *NAS Parallel Benchmarks, Multi-Zone Versions*, NAS Technical Report NAS-03-010, NASA Advanced Supercomputing (NAS) Division, (2003).

[22] Lawrence Livermore National Laboratory, *ASC Sequoia Benchmark Codes*, [online]. Available: https://asc.llnl.gov/sequoia/benchmarks/ [Accessed: 5th July 2010].

[23] pyMPI.sourceforge.net, *pyMPI: Putting the py in MPI*, [online]. Available: http://pympi.sourceforge.net/index.html [Accessed: 5th July 2010].

[24] Patrick Miller: *pyMPI - An introduction to parallel Python using MPI*, [online]. Available: https://computing.llnl.gov/code/pdf/pyMPI.pdf [Accessed: 5th July 2010].

[25] ECMWF: European Centre for Medium-Range Weather Forecasts, *ECMWF CEPMMT EZMW*, [online]. Available: http://www.ecmwf.int/ [Accessed: 5th July 2010].