Vector Processors and Novel Languages

Jim Enright
s0792366

August 22, 2008

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2008
Abstract

The requirements of HPC applications are varied and wide ranging. Thus, no one type of supercomputer is ideally suited to all applications. As a result, a number of companies have released multi-architecture 'hybrid' systems. One such system is HECToR, the main supercomputing service in the UK. HECToR consists of a Cray XT4 massively parallel scalar system and a Cray X2 vector system.

In this project, four application codes, three computational fluid dynamics (CFD) and one molecular dynamics (MD), were ported to the HECToR system and another vector system, the Cray X1E. A performance analysis of each application was then carried out to characterise them to be suitable for the vector or scalar system. The three CFD codes were found to have a better performance on the vector systems, while the MD code, LAMMPS, performed better on the XT4 scalar system.

Each code was then profiled to understand the differences in performance across the three systems. The most computationally intensive routines in the CFD codes were found to be vectorised, explaining the good performance of these codes on the vector systems. Due to its irregular memory access pattern, LAMMPS was not vectorised.

Two of the CFD codes, BenchC and Incompact3D, were optimised for the vector system. For BenchC, an attempt to improve the UPC broadcast and reduction routines was made by implementing a number of alternative algorithms. It was found that a 2-level tree UPC broadcast algorithm and a mixed mode of programming, where MPI calls replaced the UPC broadcast and reduction routines, reduced the time spent in these routines. The addition of compiler flags and some changes to the Incompact3D code reduced its execution time on 64 Cray X2 processors by 18%.
# Contents

1 Introduction 1

2 Background 3
   2.1 Introduction to the vector architecture 3
      2.1.1 Characteristics of codes which vectorise 5
   2.2 Overview of target architectures 5
      2.2.1 HECToR Cray XT4 5
   2.3 Phoenix Cray X1E 8
      2.3.1 HECToR Cray X2 10
   2.4 Applications 12
      2.4.1 BenchC 12
      2.4.2 PDNS3D 13
      2.4.3 Incompact3D 16
      2.4.4 LAMMPS 18
   2.5 Novel Languages 19
      2.5.1 Unified Parallel C (UPC) 20
      2.5.2 Co-Array Fortran (CAF) 21

3 Porting and Benchmarking 22
   3.1 BenchC 22
      3.1.1 Description of Benchmark 22
      3.1.2 Porting 22
      3.1.3 Performance Analysis 23
   3.2 PDNS3D 26
      3.2.1 Description of Benchmark 26
      3.2.2 Porting 26
      3.2.3 Performance Analysis 27
   3.3 Incompact3D 29
      3.3.1 Description of Benchmark 29
      3.3.2 Porting 30
      3.3.3 Performance Analysis 32
   3.4 LAMMPS 34
      3.4.1 Description of Benchmark 34
      3.4.2 Porting 34
      3.4.3 Performance Analysis 36
3.5 Fast Fourier transform ........................................... 38
   3.5.1 Installation of FFTW library on Cray vector systems .... 39
4 Profiling ................................................................. 40
   4.1 BenchC ................................................................. 41
      4.1.1 BenchC: UPC versus MPI communications on X1E .... 43
   4.2 PDNS3D ................................................................. 45
   4.3 Incompact3D ......................................................... 46
   4.4 LAMMPS ................................................................. 47
5 Optimisation ............................................................. 50
   5.1 Optimisation of UPC communications in BenchC ......... 50
      5.1.1 Broadcast ......................................................... 50
      5.1.2 Reduction ....................................................... 55
      5.1.3 BenchC performance with optimised UPC communications . 60
   5.2 Optimisation of Incompact3D ..................................... 62
      5.2.1 Compiler optimisation ....................................... 62
      5.2.2 Code optimisation ........................................... 64
6 Conclusions and Future Work ....................................... 68
A Work Plan ............................................................ 74
List of Tables

3.1 Values of $n_x$, $n_y$, $n_z$, $n_yb$ and $n_zb$ variables for different processor counts ........................................ 32
3.2 Time spent in main MD loop for different optimisation flag combinations using the Rhodo benchmark, simulating 32,000 atoms for 100 timesteps on 64 X1E MSPs. ............................................. 35
4.1 Breakdown of the time spent in different sections of the BenchC code running the Sphere_Big benchmark for Cray XT4 and X1E systems. 41
4.2 Breakdown of the time spent in different sections of the PDNS3D code running the T3 benchmark for Cray XT4 and X1E systems. ............... 45
4.3 Breakdown of the time spent in different sections of the Incompact3D code running on the Cray XT4 and X1E systems. ................................. 47
4.4 Breakdown of the time spent in different sections of the LAMMPS code running the Rhodo benchmark for Cray XT4 and X1E systems. ........ 48
5.1 The execution time of Incomact3D on 64 X2 processor using different combinations of compiler flags. .................................................. 65
List of Figures

2.1 Execution of array addition on a vector system. 4
2.2 Execution of array addition on a scalar system. 4
2.3 Cray XT4 cache hierarchy. Diagram reproduced from [1]. 6
2.4 The Cray XT4. 8
2.5 Cray X1E Multi-Streaming Processor (MSP). Reproduced from [2]. 9
2.6 The Cray X1E. 10
2.7 Flow of execution through BenchC. 14
2.8 Flow of execution through PDNS3D. 15
2.9 Flow chart showing the typical flow of execution of LAMMPS 19
2.10 PGAS memory model. 20

3.1 A comparison of the performance of the BenchC Sphere_Big benchmark between the Cray XT4, X1E and X2 systems. 24
3.2 A comparison of the performance of the PDNS3D T3 benchmark between HECToR XT4 (diamonds) and Phoenix X1E vector system (squares for MSP mode, triangles for SSP mode). The X-axis shows the number of PEs used - for the XT4 this is the number of cores; for the X1E it is the number of MSPs. 28
3.3 A comparison of the performance of Incompact3D between the HECToR XT4 scalar system (diamonds) and HECToR X2 vector system (squares). The X-axis shows the number of PEs used - for the XT4 this is the number of cores; for X2 it is the number of processors. 33
3.4 A comparison of the performance of the LAMMPS Rhodopsin benchmark between HECToR XT4 (diamonds) and Phoenix X1E vector system (squares). The X-axis shows the number of PEs used - for the XT4 this is the number of cores; for the X1E it is the number of MSPs. 37

4.1 A comparison of the amount of time spent in the BenchC’s Gather and Scatter functions using MPI and UPC routines. 44
4.2 A comparison of the amount of time spent in the BenchC’s Reduce and Broadcast functions using MPI and UPC routines. 45

5.1 Diagram showing how the original version of UPC broadcast works 51
5.2 A comparison of the amount of time spent in the broadcast routine of BenchC using MPI and UPC. For this the code was run for 20 time steps using the Sphere_Big benchmark. .................. 51

5.3 Diagram showing how the UPC get broadcast algorithm works .... 52

5.4 Diagram showing how the UPC 2-level tree broadcast algorithm works. 53

5.5 Diagram showing how the UPC 4 level tree broadcast algorithm works. 54

5.6 A comparison of the amount of time spent in the broadcast routine of BenchC using different UPC algorithms. ......................... 55

5.7 Diagram showing how the original version of UPC reduction works . . 56

5.8 A comparison of the amount of time spent in BenchC’s reduction routine using MPI and UPC. For this the code was run for 20 time steps using the Sphere_Big benchmark. .................. 57

5.9 Diagram showing how the full 2-level tree version of UPC reduction works ................................................................. 58

5.10 Diagram showing how the semi 2-level tree version of UPC reduction works. ................................. 60

5.11 A comparison of the amount of time spent in the reduction routine of BenchC using different UPC algorithms. ......................... 61

5.12 A comparison of the performance of the BenchC Sphere_Big benchmark on the Cray X1E using optimised UPC communication routines. . 63

5.13 A comparison of the performance of the optimised versions of Incompact3D on the Cray X2. ................................. 67

A.1 Work plan showing the changes to the initial project schedule given in the Project Preparation report. The red arrows indicate the final schedule. 74
Acknowledgements

I wish to sincerely thank my supervisors, Dr. Alan Gray and Jason Beech-Brandt for their guidance and continued support throughout the project, and Dr. Lorna Smith for her help in the initial stages of the project.

Also, to all the staff and students at EPCC who provided a good working environment and made my time here very enjoyable. In particular, I wish to thank Toni Collis for her help during the writeup stage.

I also wish to acknowledge EPSRC who provided funding for me to undertake this Masters programme.
Chapter 1

Introduction

The majority of supercomputers today use commodity based microprocessors. In the latest Top500 list \(^1\) detailing the 500 most powerful supercomputers, 430 are based on either AMD or Intel processors [3]. The main variation between microprocessor based supercomputers is the structure of the memory hierarchy and the architecture of the interconnect.

However, as the density of transistors on a microprocessor chip continue to increase with Moore’s Law [4], the power and cooling requirements for the chip start to become a significant factor in the costing of a supercomputer. Moreover, it has become apparent that the requirements of applications which execute on supercomputers are varied and wide ranging - from the simulation of particles to large scale climate modelling. Thus, no one type of supercomputer is ideally suited to all applications.

In response to these issues, DARPA launched their High Productivity Computing Systems (HPCS) programme in 2002. The HPCS programme offered funding to industry and academia in order to "provide a new generation of economically viable high productivity computing systems" [5]. To achieve this goal, the programme focusses on two main areas of computing. The first area is the specification of novel languages which are easy to use and give high performance. The second is the design of a supercomputer which has high performance and scalability, has a low power consumption and is easy to use.

Cray were one of the companies to receive funding under the HPCS programme. The Cray roadmap for the period 2004 - 2010 aims to meet the objectives of the HPCS programme with the release of a multi-architecture 'hybrid' system [6]. HECToR, the main supercomputing service in the UK, is one such Cray 'hybrid' system. HECToR is a Cray XT5h system consisting of a Cray XT4 massively parallel scalar system and a Cray X2 vector system. When running a code on HECToR, users are able to choose either the vector or scalar units of the system.

In this project, the performance of a number of application codes on two Cray vector

\(^1\)published in June 2008 at the International Supercomputing Conference in Dresden, Germany
supercomputers, the Phoenix X1E at ORNL ² and HECToR’s X2, is investigated. The performance of these codes is compared to the performance on HECToR’s XT4 scalar system. From this, the characteristics of applications best suited to vector systems will be identified. These results will allow users of the HECToR service to be advised on whether to use the vector or scalar units of the system in order to get the best performance for their code.

Also, a number of novel languages have been released as part of the HPCS programme, including Unified Parallel C and Co-Array Fortran. The afore mentioned Cray vector supercomputers, the X1E and X2, support these two languages at a hardware level. As part of this project, an investigation takes place into the usability and performance of UPC on the vector systems compared to that of traditional parallel computing languages such as MPI.

The structure of this thesis is based on the order in which the different stages of work have been carried out. Chapter 2 gives an overview of vector architectures and the characteristics of codes which are amenable to vectorisation. A description of the three target architectures, the Cray X1E, X2 and XT4 is then given. It also includes a brief overview of each of the application codes used in the project, BenchC, PDNS3D, Incompact3D and LAMMPS. Finally, the chapter gives an introduction to the UPC and CAF novel languages.

Chapter 3 describes the steps required to port the four codes to each system and details any problems which were encountered during porting. Also, the results of performance tests of each code on the three target systems are presented and analysed. The chapter concludes by describing the steps needed to install the FFTW library on the Cray vector systems.

To understand any differences in performance across the systems, a profile of each code is presented in Chapter 4.

Finally, Chapter 5 describes the optimisations made to two of the codes, BenchC and Incompact3D. For BenchC, an attempt to optimise the broadcast and reduction routines in the UPC version of the code was made. An attempt to reduce the overall runtime of Incompact3D was made by applying compiler optimisation flags and by modifying certain areas of the code.

²Oak Ridge National Laboratory in the United States
Chapter 2

Background

2.1 Introduction to the vector architecture

Vector architecture systems have a number of advantages over scalar microprocessor systems. These include a high floating point operation (FLOP) rate and high bandwidth memory access [7]. Also, vector processors have a more efficient use of memory than their scalar counterparts. Nearly every value loaded from memory is used by the vector processor [8]. However, as discussed below, an application must have certain characteristics to exploit the advantages offered by vector systems.

Vector architecture systems operate in a Single Instruction Multiple Data manner, performing the same operation on a series of scalar data elements such as arrays [9]. Scalar architecture systems, on the other hand, only operate on a single pair of operands at a time. This means that a vector system can perform a much higher number of floating point operations per instruction than scalar systems, and less complex logic is required for the instruction fetch/decode circuitry [10].

Vector systems use a type of data pipelining, where the processor performs different parts of the same vector instruction on different data elements, to increase the throughput of the vector processor. This allows a new result to be produced every clock cycle once the pipeline is full [11].

Also, the memory of vector systems is divided into banks with vector elements stored such that successive elements are in successive memory banks [8]. This makes it possible to load/store many elements of an array in a single cycle which reduces the load/store latency and gives a high memory bandwidth.

To demonstrate how a vector processor system works, we look at the execution of the following code snippet on both a vector and scalar system.

```plaintext
do i = 1, 5
   x(i) = a(i) + b(i)
end do
```
To execute this snippet on a vector system the following steps (as shown in Figure 2.1) are required.

1. Load a series of elements from array \(a\) and array \(b\) to vector registers\(^1\).
2. Using the vector add unit, add corresponding the values of \(a\) and \(b\) in the vector registers and store the result in \(x\).
3. Load the next series of elements of array \(a\) and \(b\) to the vector registers.
4. Repeat until all values of \(x\) have been calculated.

\[
\begin{array}{ccc}
\text{a(1)} & + & \text{b(1)} \\
\text{a(2)} & + & \text{b(2)} \\
\text{a(3)} & + & \text{b(3)} \\
\text{a(4)} & + & \text{b(4)} \\
\text{a(5)} & + & \text{b(5)} \\
\end{array} \rightarrow \begin{array}{c}
\text{x(1)} \\
\text{x(2)} \\
\text{x(3)} \\
\text{x(4)} \\
\text{x(5)} \\
\end{array}
\]

**Figure 2.1:** Execution of array addition on a vector system.

In comparison, the scalar system must perform the following steps (shown in Figure 2.2).

1. Read one element of array \(a\) and \(b\) to scalar registers.
2. Add the two elements using the scalar add unit and store in corresponding element of \(x\).
3. Increment loop index and load next two elements of \(a\) and \(b\).
4. Repeat for all elements of \(a\) and \(b\).

\[
\begin{array}{ccc}
\text{a(1)} & + & \text{b(1)} \\
\text{a(2)} & + & \text{b(2)} \\
\text{a(3)} & + & \text{b(3)} \\
\text{a(4)} & + & \text{b(4)} \\
\text{a(5)} & + & \text{b(5)} \\
\end{array} \rightarrow \begin{array}{c}
\text{x(1)} \\
\text{x(2)} \\
\text{x(3)} \\
\text{x(4)} \\
\text{x(5)} \\
\end{array}
\]

**Figure 2.2:** Execution of array addition on a scalar system.

\(^1\)Vector registers are areas of high speed memory which hold multiple data elements.
2.1.1 Characteristics of codes which vectorise

In general applications with explicit loops (do loops in Fortran, for loops in C/C++) and implied loops (Fortran array syntax) are suited to vectorisation [9]. If loop iterations are independent (i.e. there are no dependencies between iterations) it allows for parallel evaluation in the vector pipeline, giving maximum throughput. Furthermore, if the loops have a stride 1 data access pattern, where array elements contiguous in memory are accessed, full utilisation of the high memory bandwidth of the vector system is achieved.

There are also a number of characteristics which inhibit vectorisation of an application including:

- loops with data dependencies or dependencies between iterations;
- loops with non-inlined procedure calls;
- loops which make I/O calls;
- return, exit, stop or pause statements in the body of a loop.

When a piece of code is not vectorised there are two ways in which it can be executed on the vector system.

1. Run on the accompanying scalar processor. This is usually used for address translation and is of much lower performance than the vector processor;
2. Run on the vector processor with a vector length of 1.

On Cray systems non-vectorised code is usually executed on the slower scalar processor. This is because the the cost of loading vector elements of length 1 for execution on the vector processor is more expensive than execution on the scalar processor. Thus, the performance of the non-vectorised codes is much worse than code which can be successfully vectorised.

2.2 Overview of target architectures

2.2.1 HECToR Cray XT4

The Cray XT4 is a massively parallel processor (MPP) distributed memory system. It uses AMD commodity microprocessors with a custom built memory and communication system. This is combined with a tightly coupled operating and management system making the XT4 a highly scalable and reliable system [1].

The XT4 as part of the HECToR service consists of 5,446 dual core AMD Opteron processors [12]. This gives a total of 11,328 cores with each core clocked at 2.8 GHz. The AMD Opteron core is able to perform 1 floating point addition and 1 floating point
multiplication per clock cycle, giving each core a peak performance of

\[ 2 \text{ FLOP per cycle} \times 2.8 \text{ cycles per second} = 5.6 \text{ GFLOPs}. \]

The HECToR XT4 achieved a peak performance of 54 TFLOPs for the LINPACK benchmark [13]. This places it at number 29 in the Top500 list announced in June 2008 at the International Supercomputing Conference in Dresden, Germany [3].

The processors on the XT4 are used as either compute PE’s or service PE’s: **Compute PE’s** are used for user applications and run a lightweight kernel to ensure there is little OS noise during a application’s execution; **Service PE’s** run SuSE Linux and can be configured for I/O, login, network or system functions.

There is 6 GB of memory shared between each dual core processor, giving the XT4 a total memory capacity of 33.2 TB. Each core has two levels of cache with no shared cache between cores. There is a separate data and instruction L1 cache. This is a 2-way set associative cache of size 64KB [14]. L2 cache is a 1 MB, 16-way set associative combined instruction and data cache. All caches have 64 byte cache lines. Figure 2.3 shows the memory hierarchy for an XT4 processor.

The XT4 has two modes of operation:
- **Single node mode (SN)** where one core of each chip performs computation while the second core is left idle. In this case all of the memory is available to the core performing the computation.
- **Virtual node mode (VN)** where both cores of the processor perform compute tasks and the memory is shared between the two cores i.e. each task has 3 GB of memory.

![Figure 2.3: Cray XT4 cache hierarchy. Diagram reproduced from [1].](image)

A 3D torus communication network consisting of a SeaStar chip connects the processors in the XT4. This directly connects each processor to 6 of its nearest neighbours with a bidirectional bandwidth of 7.6 GB/second and latency of 5\(\mu\)s. The SeaStar chip
has a Power440 core which manages the 6 links and performs some of the communication, freeing the Opteron chip for computation [15].

Twelve of the service nodes on HECToR are configured as I/O nodes. These are integrated with the 3D torus network and connect the machine to a 576 TB RAID disk via an infiniband network. A Lustre parallel file system is used to allow access to the disk by all I/O nodes.

HECToR runs version 2.0.53 of the Unicos/lc operating system. This consists of two components; a lightweight kernel for compute nodes and a full version of SuSE Linux for service nodes.

On login to HECToR, the Cray XT programming environment is loaded. This sets the path for the compilers and parallel libraries. For this project version 2.0.53 of the Cray XT programming environment was used. This loads the following compiler and library versions:

- version 7.0.4 of the Portland Group C/C++ compiler, pgcc;
- version 7.0.4 of the Portland Fortran 90/95 compiler, pgf90;
- version 2.0.49 of the Cray Message passing toolkit (MPT);
- OpenMP, to allow shared variable programming between 2 cores.

As well as the Portland Group compilers the PathScale and GNU compiler suites are also available on the XT4 and it is possible to load these if needed. Cray provide compiler wrapper scripts, ftn for the Fortran compiler and cc/CC for the C/C++ compiler. These wrappers give a single command to access the compilers and links all the necessary parallel libraries. The actual compiler used depends on which compiler suite module has been loaded.

The portable batch system, PBS, is used to queue and run production jobs on HECToR [16]. A submission script launches the parallel job using the aprun command. Two important parameters which need to be specified in the PBS submission script are -n and -N:

- The -n parameter specifies the number of MPI tasks which are required.
- The -N parameter indicates if the application will be run in single node or virtual node mode. Selecting -N 2 selects virtual node mode allowing two MPI tasks per dual core processor, while -N 1 selects single node mode.

- N 2 is the usual choice but -N 1 could be used if an MPI tasks uses a large amount of memory or uses multiple threads per MPI task.

To queue a job the PBS qsub command is used, passing the submission script as an argument. To run on HECToR, users are given a budget of allocation units, AUs. These AUs are deducted from according to the number of cores requested and the runtime of each job submitted, with an one core hour equal to 4.82 AUs [17].
2.3 Phoenix Cray X1E

The Cray X1E is a distributed shared memory vector system. It has features that were available in previous high performance vector systems such as the Cray X-MP and Cray J90. These include powerful vector processors, high memory bandwidth and special vector instructions [18]. The Cray X1E also has a highly scalable memory hierarchy and optimised communication network which was available in the Cray T3E MPP system.

The basic processing block of the X1E system is a multi-steaming processor, MSP (shown in figure 2.5). An MSP is made up of four single-streaming processors, SSPs, each with a two-pipe vector processing unit clocked at 1.13GHz and a superscalar processor with a clock speed of 565 MHz [19]. The vector processor of the SSP has 32 vector registers, each having a length 64 elements. These act as a type of cache for vector operands. Array elements needed by a vector operation are read into the vector registers. In comparison to previous vector systems this is a large number of short length registers, making it easier for a wide range of algorithms to effectively use the vector pipes [18].

Each vector pipe is capable of performing one floating point multiplication and one addition per clock cycle. This gives an SSP a peak performance of:

\[
2 \text{ FLOP per cycle} \times 2 \text{ vector pipes per SSP} \times 1.13 \text{ cycles per second} = 4.52 \text{GFLOPS}
\]

and a total performance of 18.08 GFLOPS per MSP.

The Phoenix Cray X1E, housed at Oak Ridge National Laboratory in the United States, consists of 1,024 MSPs [20]. This has a LINPACK performance of 14.96 TFLOPs which places it at number 175 in the Top 500 list of the most powerful supercomputers [3].

On the Phoenix system, each MSP has 2 GB of memory with four MSPs combining to form a node with 8GB of shared memory. An MSP also has two levels of cache with
the structure helping to provide an efficient shared memory programming model within an X1E node. The L1 cache is a 16 KB 2-way set associative data and instruction cache [2]. This cache is coherent between the four single streaming processors of an MSP. The L2 cache is 2MB in size and is coherent between the four other MSP’s in a node. A fast L2 cache/CPU bandwidth of 76.8 GB/second ensures that the vector registers are kept full throughout program execution.

The total memory is physically distributed between nodes but a process can directly address memory on any other node using remote memory requests. These remote requests go over a low latency (5 \( \mu \)s), high bandwidth (51 GB/s per compute module) interconnect connected in a 2D torus topology [2]. Having such fast remote memory access allows for efficient use of novel PGAS languages (these will be described further in Section 2.5).

Each compute module has four I/O ports which connect to the I/O cabinet and storage devices. Each I/O port has a bandwidth of 1.2 GB per second. The I/O ports are connected to the memory and interconnect subsystems and can be accessed by all processors, allowing I/O access from anywhere in the system.

Phoenix runs the UNICOS/mp 3.1.46 operating system. This is a single system image OS which allows for easy system administration, ease of interaction with the I/O subsystem and a single login to the system [19].

As well as the X1E, the Phoenix service offers a fast cross compiler system, Robin. This is a Linux based system with 4 dual-core AMD Opteron processors. It can be used to compile, submit and manage jobs for the X1E system instead of interactively using the front-end X1E processors.

---

2Two nodes form a compute module, with 8 MSPs and 16GB of shared memory
3PGAS = Partitioned Global Address Space
For this project version 5.6.0.3 of the Cray X1E programming environment was used. This loads the following versions of the compiler and parallel libraries:

- Version 5.6.0.3 of the Cray C/C++ compiler.
- Version 5.6.0.3 of the Cray Fortran compiler which supports the Fortran 2003 standard.
- Version 2.4.0.7 of the Cray Message passing toolkit (MPT) which provides an interface to the MPI version 1.2.

Phoenix uses PBS to run and queue jobs on the back-end. In the submission script, the number of MSP resources needed are specified using the `#PBS -l mppe=N` command and the runtime specified with the `#PBS -l walltime=01:00:00` command. The submitted job is queued based on the requested time and the number of MSPs.

![Figure 2.6: The Cray X1E.](image)

### 2.3.1 HECToR Cray X2

Like the Cray X1E, the X2 is a distributed shared memory vector system [21]. However, its design is less complex than that of the X1E’s, with implements streaming between vector processors. This makes the X2 cheaper to produce while still maintaining a high performance.

The X2 processor includes an 8 pipe vector CPU clocked at 1.6 GHz and a 4-way dispatch superscalar core with a clock speed of 800 MHz [22]. Each processor has 32 vector registers. This is the same number as on the Cray X1E although the length of the registers are longer at 128 elements, in comparison to 64 on the X1E. Each vector pipe contains a floating point adder and multiplier unit, allowing 2 floating point operations to take place per clock cycle. This gives each X2 processor a performance of:
2FLOP per cycle $\times$ 8vector pipes per CPU $\times$ 1.6 cycles per second $= 25.6$ GFLOPS.

Toward the end of this project an X2 system was added to the HECToR service. This contained 112 X2 processors, giving the system a peak performance of approximately 2.8 TFlops [23].

Four X2 processors combine to form a compute node with 32 GB of shared memory. This allows them to be implemented as a 4-way shared memory node. However, unlike the X1E the compiler will not try to stream loops over a compute module and any parallelisation over a node must be performed explicitly by the programmer.

Three levels of cache exist to ensure that the functional units are kept saturated with data. A separate 16 KB, 2-way set associative data and instruction L1 cache is used for memory loads and stores by the scalar processor. The L2 cache is 512 KB in size with a bandwidth of 166 GB/second. This cache is divided into 32 banks and vector references are sent directly to this level of cache. The L3 cache is an 8 MB, 16-way set associative cache which is shared between the 4 processors in a compute node. This allows for efficient shared memory programming within a node.

The X2’s interconnect is a folded Clos or fat-tree network where nodes are connected to switches at the base of the tree and the number of links increase as one moves closer to the root of the tree. This interconnect gives each processor reliable access to all of the memory with a low-latency of 3.5 $\mu$s and point-to-point bandwidth of 16.0 Gb/second [22].

On the HECToR service, the X2 interconnect interfaces with the Cray Seastar interconnect of the XT4. This allows a common user login, application development, I/O and storage services for the X2 and XT4 systems.

On login to HECToR, the X2 programming environment is loaded using the following commands:

1. `module purge` to unload the current programming environment;
2. `module use /opt/ctl/modulefiles` to specify the location of the X2 module files;
3. `module load PrgEnv-x2` to load the X2 compilers and libraries.

This loads version 6.0 of the X2 programming environment which includes version 6.0.0.4 of the Cray C/C++ and Fortran compilers; version 1.0.5 of the Cray X2 message passing toolkit which implements the MPI-2 standard and the OpenMP library for shared memory programming within a compute node.

Like the XT4, the PBS command `qsub` is used to queue jobs for execution on the X2. The only addition to the PBS submission script is the line `#PBS -q vector` to place the job in a special vector queue indicating that the job is to be run on the X2. The `−N`
and \(-n\) parameters are also similar to that for the XT4. \(-n\) specifies the total number of tasks to be run and \(-N\) specifies how many tasks are run per node. As a node on the X2 is a 4-way SMP, the \(-N\) parameter can range up to 4, as opposed to 2 on the dual-core XT4.

Jobs on HECToR’s X2 are charged according to the number of processors requested and the runtime of the job submitted. This charge is deducted from a separate vector budget, with the cost of running on the vector system anticipated to be approximately 8 times more expensive than running on the scalar system [17].

2.4 Applications

2.4.1 BenchC

BenchC [24], developed by the Army High Performance Research Center (AHPCRC) in the United States, is a parallel computational fluid dynamics (CFD) application code. It uses Finite Element Methods (FEM) to solve incompressible flow in large unstructured mesh domains [25]. The code has been used at the AHPCRC since 1991 to find the velocity and pressure of airflow in areas such as high speed missile aerodynamics; parachute aerodynamics; unmanned aerial vehicle simulations and contaminant dispersion simulations [26].

As BenchC is very similar to other CFD codes in use at the AHPCRC, it is used as a representative code to benchmark performance on different supercomputers. It was originally developed for the Thinking Machine CM-5, a distributed memory vector processor system, and written in C with data-parallel language constructs [24]. Using the data-parallel model allowed for a quick and easy parallelisation of the code. However, there was a strong reliance on the compiler in order to achieve good performance and scaling. To give the programmer more control over the scalability of the code it was later re-written using the message passing model of parallel programming.

BenchC can also be compiled to use UPC calls (see Section 2.5.1) instead of MPI for the main inter-processor communication during the code. The mesh partitioning at the beginning of the program still uses MPI routines. A comparison between the two versions of the code will be explored in section 3.1.3.

More detail about BenchC

BenchC has two stages of execution, pre-processing and time marching, as shown in Figure 2.7

The pre-processing stage consists of the following steps.

1. Read the unstructured finite element mesh.
2. Partition and distribute the mesh amongst the processes using Recursive Co-
ordinate Bisection (RCB) algorithm [27]. RCB recursively cuts the mesh, in a
direction parallel to a co-ordinate axis, into two sub-domains until the mesh is
equally divided amongst the processes.

3. Setup of the communication pattern between processes.

4. Allocate the dynamic memory needed during the program.

The majority of the computation is performed in the time marching section of the code.
It involves four nested loops. The execution time is proportional to the number of
iterations performed by each of the four loops.

The outer-most loop is the number of time steps as specified in the input file. The second
nested loop is due to the non-linearity of solving the Navier-Stokes equations [28]. The
Navier-Stokes equations are solved within this loop using the GMRES Krylov subspace
method [29]. The GMRES solver has two internal loops.

Two methods are available to solve the system of Navier-Stokes equations in the GM-
RES loop.

The first method, sparse matrix, stores the equations as sparse matrices. The second,
matrix free uses less memory than the first by computing the matrix-vector product as
part of the initial Finite Element calculation. Both methods will return exactly the same
answers, however, because of its low memory usage, the matrix-free method is usually
used for simulations with very large mesh sizes.

BenchC reads in a finite element mesh, made up of tetrahedral elements, from four files.
Three of these, mien, mrng, mxyz, are stored in a specific binary format. The other,
mesh.info, is a text file which specifies the number of nodes and number of elements
in the mesh.

The operation of BenchC is controlled using parameters in an input file. The parameters
include the number of time steps, nts; the number of non-linear iterations, nit; the
size of the Krylov search space, kspace and whether the simulation will use the sparse
or matrix free mode of operation.

It is also possible to continue a simulation from a previous run by specifying the restart
parameter in the input script. If this is enabled a restart file, dresti, will be read to
get the initial values for velocity and pressure of the flow.

Upon successful completion of the code the amount of time spent in each part of the
code is reported to the user, along with more detailed information from the GMRES
solver during each time step.

2.4.2 PDNS3D

PDNS3D is a computational fluid dynamics code, developed by the UK turbulence
consortium (UKTC), used to solve the Navier-Stokes equations using direct numer-
Figure 2.7: Flow of execution through BenchC.

Numerical simulation (DNS) finite difference techniques [30]. The code is used to model turbulent fluid flow. Particular emphasis is given to the interactions occurring at the shock/boundary layer.

It is based on a sophisticated DNS code which includes [31]:

- a ‘high order central difference scheme’ to give a more accurate approximation of derivatives in the finite difference method;
- a shock-preserving advection scheme that allows the interactions at the boundary layer to be modelled more precisely;
- entropy splitting of the Euler terms;
- and a stable boundary condition scheme.

The code is written in Fortran 90. The source files are first passed through a C preprocessor to resolve a series of #ifdef directives in the code. These directives specify certain compile time options including the use of dynamic or static memory, the architecture on which the code is being compiled, and the enabling of certain optimisations.

Parallelisation is achieved using MPI. At runtime the user can choose to run code with synchronous or asynchronous communication routines. The use of Fortran and MPI
allows the code to be efficient and easily used on a wide range of scalable HPC architectures.

**More detail about PDNS3D**

There are three main stages in the execution of PDNS3D, as illustrated in Figure 2.8.

**Figure 2.8:** Flow of execution through PDNS3D.

The first stage is the simulation setup. This includes the grid generation which is carried out in two parts. First, a small independent serial code, Jacobs_ch, is run. This generates a 2 dimensional x-y grid that is outputted as a unformatted binary file. Second, this file is read by the master process at the beginning of the parallel code. The grid is then extended into the third dimension. By splitting the grid generation in this way, the fraction of serial code within the parallel program is minimised. Following this the master process uses a recursive bisection algorithm [27] to partition the grid into sub-domains. This works by cutting the grid parallel to its co-ordinate axes. These sub-domains are then sent to the other processes to form local meshes.
The second stage performs the time marching. During each time step, a Runge-Kutta based iterative method [32] is used to solve a set of ordinary differential equations. At the end of each time step there is a halo exchange to send boundary data to neighbouring processes.

The final stage is the data gathering and reporting stage. Information about the velocity, volume and pressure of the fluid across each point in the problem domain are output to a text file, stats.dat while setup and timing information is written to standard output. Two options are available to perform the I/O during the data gathering stage. The first is master I/O, where all the information to be written is gathered onto the master process and written to a file. The second option, a more scalable solution, uses MPI I/O, allowing each process to read and write in parallel.

PDNS3D also allows the option of compiling with dynamic or static memory. The dynamic memory option uses allocatable arrays and gives a single executable for any combination of problem size and processor grid. This is a more convenient solution, but studies have found that it has an adverse impact on the performance of the code [33]. Thus, static memory, where the value of the problem size and the number of processors are set at compile time in the incld3d.F file, is recommended for better performance. A script, called compile, is provided with the PDNS3D distribution to allow for easier building of static executables for different processor counts.

The operation of a PDNS3D simulation is controlled by an input file. This file contains parameters such as:

- the dimensions of the grid;
- the location of the binary grid file;
- the number of time steps for which the simulation will run;
- whether the code will use synchronous or asynchronous MPI communications;
- and the type of I/O strategy that will be used in the data gathering stage.

The input file is passed to the to the code at job submission.

2.4.3 Incompact3D

Like PDNS3D, Incompact3D is a CFD application code which simulates turbulent fluid flow by directly solving the incompressible Navier-Stokes equations [34]. It was developed at the Department of Aeronautics at Imperial College London. It is used in order to gain insight into the dynamics of turbulence and to explain wind tunnel measurements carried out. The code has been given early access to the HECToR X2 system.

Incompact3D uses a sixth-order finite difference scheme to accurately solve the Navier-Stokes equations for turbulent flow on a cartesian grid. A fractional step method, which introduces a Poisson equation for the pressure of the flow, is used to ensure that incompressibility is maintained throughout the simulation. The Poisson equation is solved via
a 3D fast fourier transform. Periodic and free-slip boundary conditions, as well as a grid stretching technique, can all be applied by using the Poisson equation. Grid stretching allows the extension of the simulation domain along one of its dimensions, helping to minimise numerical errors.

The code is written in Fortran and parallelised using MPI. It uses a regular grid domain decomposition. Each MPI process works on an equal size of the simulation domain. A distributed 3D FFT, performed using calls to the FFTW library [35], is needed to solve the Poisson equation.

More detail about Incompact3D

Like the other CFD codes seen, there are three main stages in the Incompact3D’s execution.

In the first stage, the setup, the parallel environment is initialised and the simulation parameters are read from the input file. Also in this stage the grid stretching is applied to the domain. The Navier-Stokes ordinary differential equations (ODEs) are formed either by reading from a restart file or by initialising the appropriate variables.

The second stage is the time marching loop. At each iteration the ODEs are solved using an iterative temporal scheme. There are two temporal methods available in the code, Adams-Bashforth and Runge-Kutta [32]. The choice of method to be used is specified in the input script. As part of the Poisson equation solver, a 3 dimensional FFT is applied during each time step. At the end of each time step the maximum and minimum velocity of the flow in each dimension of the domain are found using a set of MPI_Reduce calls.

At the conclusion of execution each process writes the value of the final velocity of the flow to an output file and reports the total runtime for the simulation.

The operation of Incompact3D is controlled by an input file, called incompact.prm. The input file is split into four main sections.

1. Flow parameters - this includes the Reynolds number, the maximum and minimum velocity of the initial flow, the turbulence intensity and the size of a time step.
2. Flow configuration - this allows the user to specify the number of time steps for which the simulation will run; whether or not to perform the grid stretching and the type of temporal scheme to use to solve the ODEs.
3. File parameters - including the name of the output file and how often to write a backup file.
4. Body forces - this specifies the size and position of the body forces acting on the fluid.
2.4.4 LAMMPS

Large-scale Atomic/Molecular Massively Parallel Simulator, LAMMPS, is a classical molecular dynamics simulation code [36]. It is used to model the forces and positions of a collection of particles in a liquid, solid or gaseous state. These particles can be part of an atomic, polymeric, biological, metallic or granular system. The forces on the particles and the type of boundary conditions depend on the type of system used.

LAMMPS was developed as part of a mid 1990’s research and development agreement between two United States Department of Energy laboratories (Sandia and Lawrence Livermore National Laboratory) and three companies; Cray, Bristol Myers Squibb and Dupont [37]. The first version of LAMMPS, LAMMPS 99, was written in Fortran 77 but was converted to Fortran 90 for the LAMMPS 2001 version. The later version took advantage of Fortran 90 features such as free-form source code input, a modular programming style and dynamic memory allocation.

In 2004 the code was re-written in object oriented (OO) C++. The new code also included many new features from other parallel molecular dynamics codes developed at Sandia labs [37]. The OO style provided by C++ allows for encapsulation of data within classes and makes the code easy to modify. Also, the code can be easily extended by adding a new class and this should not affect other parts of the code.

All versions of LAMMPS use a spatial domain decomposition strategy [37]. This divides the data in the simulation domain into 3D sub-domains which are then assigned to each processor. MPI_Send and non-blocking MPI_Irecv calls are used to exchange atom information between neighbouring processes.

More detail about LAMMPS

LAMMPS simulates a molecular system by applying Newton’s equations of motion at each time step to update the positions and velocities of each atom. The code attempts to perform these calculations efficiently for atoms which are close to each other by using neighbour lists which keep track of nearby atoms.

Depending on the physical system being simulated, the type of interaction between atoms and the force acting on them differ. The LAMMPS distribution provides solvers, which define formulae and coefficients for various types of forces including pairwise potentials (Lennard-Jones), CHARMM force fields, polymer potentials and many-body potentials (EAM).

A LAMMPS simulation works by reading commands from an input script text [37]. The input script is generally split into four stages:

1. Initialisation - this sets parameters which define the molecular system and simulation domain. For example, the size of the processor grid in each dimension.
2. Atom definition - this reads the position and forces of atoms from a new simulation data file or from an existing simulation restart file.
3. Settings - this sets simulation parameters such as boundary conditions, number of time steps, force field options and output options.

4. Run simulation - once all commands in the input script are executed, the MD simulation is started using the run command.

Figure 2.9 summarises the flow of execution of the LAMMPS code.

![Flow chart showing the typical flow of execution of LAMMPS](image)

2.5 Novel Languages

Recently, a number of new programming languages, such as Unified Parallel C (UPC), Co-Array Fortran (CAF) and Titanium, have emerged. These languages adopt a partition global address space (PGAS) programming model [38]. This model combines features from the message passing and shared memory models of programming.

Like MPI, PGAS programs run in a single program multiple data (SPMD) fashion where multiple processes execute the same program but with different inputs. This allows the program to have high performance and to achieve good scaling.

Like the shared memory model PGAS uses a global address space. This address space is partitioned among processors with each shared data element having an affinity to a processor. The PGAS model also defines a two level memory hierarchy with private and shared memory, as shown in Figure 2.10. The private memory can only be accessed by the owning processor while the shared memory can be accessed remotely by other processors [39].

The Cray X1E and X2 vector systems can access memory across the entire system at a hardware level through the interconnect. Thus access to remote shared memory needed by PGAS languages can be done with a low latency and high bandwidth. The
compilers on the Cray X1E and X2 systems support the UPC and CAF libraries. A brief description of each of these languages is given below.

![PGAS memory model](image)

**Figure 2.10:** PGAS memory model.

### 2.5.1 Unified Parallel C (UPC)

Unified Parallel C is an extension to the ANSI C standard [40]. A number of commercial UPC compilers exist including versions from Cray, SGI and HP while the UPC Berkeley distribution is an open source version.

UPC threads operate independently executing in a SPMD fashion. The number of threads can be specified at run time or at compiler time [41]. The total number of threads is stored in a program variable called `THREADS` while the thread index is stored in the `MYTHREAD` variable. Threads are indexed from 0 to `THREADS - 1`.

Normal C variables are allocated in the private memory space of a thread [39]. Shared scalar variables are only allocated by thread 0 and reside in thread 0’s shared memory space. Shared array elements are spread across the threads. By default shared elements are distributed in a cyclic fashion. For example, the declaration

```c
shared x[THREADS];
```

assigns one element of `x` to each thread. However, it is also possible to assign blocks of array elements to each thread at declaration. For example,

```c
shared y[4][THREADS];
```

gives 4 elements of the array `y` to each thread.

In UPC there is no explicit synchronisation between threads. Thus, a number of synchronisation mechanisms are provided by UPC including:

- a global barrier (`upc_barrier`) which blocks a thread until all threads have arrived at the barrier;
• split-phase barriers which allows a thread to perform some work unrelated to the barrier while others wait;
• and locks to protect access to a shared resource.

UPC also defines a work sharing directive, upc_forall, which distributes loop iterations amongst threads and routines for parallel I/O.

2.5.2 Co-Array Fortran (CAF)

Co-Array Fortran is a parallel extension to the Fortran 95 language [42]. Like UPC, it is uses the SPMD model of execution. It works by replicating the program into a fixed number of images [43]. Each image has its own set of variables and is executed asynchronously. The number of images is available using the num_images() intrinsic function and the index of any image can be found with the image_index() function.

Sharing of data between images is achieved by declaring an object as a co-array [44]. Declaration and usage of a co-array is an extension of Fortran array syntax. For example,

```
integer, dimension(n)[:] :: x
```

declares an array of size \( n \) for each image. Remote memory access can be achieved by using a co-array with array syntax. For example to allow image \( p \) access to all elements of image \( p+1 \) the following instruction is used:

```
x(:,:,p+1).
```

CAF also provides a number of synchronisation routines including barriers and critical sections.
Chapter 3

Porting and Benchmarking

3.1 BenchC

3.1.1 Description of Benchmark

A typical BenchC application works on a mesh with between 2 and 100 million tetrahedral elements. However it has also been used on meshes with up to 1 billion elements [25]. Four non-linear iterations per time step, 20 GMRES solver iterations, and between 100 and 2000 time steps are used for a typical simulation.

To compare the performance of BenchC on the target systems, the Sphere_Big benchmark [45] is used. This models the flow of air around a large sphere, consisting of a mesh with over 11.5 million tetrahedral elements. The total memory requirement for the benchmark is approximately 4.2 GB, which is dynamically allocated during the pre-processing stage of the code. The benchmark uses the matrix-free method (see Section 2.4.1) with 4 non-linear iterations per time step, 20 GMRES solver iterations and 60 time steps.

3.1.2 Porting

The code is written in C and only requires an MPI library (or a UPC library, if compiled with this option). This makes it easy to port to a variety of platforms.

The C source files and the makefile for the code are in the src/ directory. The makefile contains compile options, including optimisation flags, for both the Cray X1E and XT4. For the X1E the -O3 -haggress -hinline4 -hfp3 optimisation flags are used, while on the XT4 the -fast -O4 -Mipa=fast,inline -Msmartalloc -Minfo -Mneginfo flags are used.

As part of this project compile options for the Cray X2 were added to the makefile. The X1E build rules were used as a template for the X2 compilation.
With the X1E build rules, compile time errors highlighted undefined references to some functions. For example the 
/nfs01/d04/d04/s0792366/src/BenchC/src/main.c:271: undefined reference to ‘sparse_setup’
error occurred when compiling main.c.

On inspection of the code it was found that some functions had different argument lists if it was to be run on a vector system. This was due to optimisations carried out to improve the performance of the code on vector systems [25]. These optimised are selected using a series of #ifdef __crayx1 pre-processor directives. The __crayx1 variable is specific to the Cray X1 series.

To allow the code to compile on a variety of vector platforms the __crayx1 variable was replaced by a CRAY_VECTOR variable. This was passed to the pre-processor using the -DCRAY_VECTOR flag at compile time. The #ifdef __crayx1 statements were then changed to #ifdef CRAY_VECTOR. With this change the code successfully compiles and runs on both the X1E and X2 vector systems with the same set of build rules.

3.1.3 Performance Analysis

To benchmark and compare the performance of BenchC on the three target systems - the Cray XT4, X1E and X2 - the Sphere_Big benchmark was run for 60 time steps on each system. This number of time steps gives a reasonable runtime on each of the systems, and the pre-processing stage of the code does not dominate the overall runtime. The minimum execution time over two runs of the benchmark is used for performance comparison.

Figure 3.1(a) shows the total execution time of the code multiplied by the number of processing elements. On the XT4 a processing element corresponds to the number of cores, on the X1E it is the number MSPs and on the X2 it is the number of processors. The total execution time includes the time spent in I/O and the pre-processing part of the code. Figure 3.1(b) shows the number of floating point operations performed per second within the main computational part of the code for the three target systems.

On the XT4 the Sphere_Big benchmark was run between 4 and 1024 cores. On the X1E and the X2 vector systems the problem size of the benchmark is too large to distribute between 4 processing elements. Thus, on the X1E successful runs were completed over 8 and 512 MSPs and on the X2 between 8 and 64 processors. The MPI version of the code was run on all three systems while the UPC versions was also benchmarked on the X2 and X1E vector systems. We were unable to run the UPC version of BenchC on the XT4 since UPC is not installed on this system.

We can see the scalability and efficiency of the code on the three systems in Figure 3.1(a). Ideal scaling would be shown by a horizontal line as the problem size per process decreases with increasing number of processing elements.
(a) The dependence of the number of processing elements multiplied by total time for the benchmark against the number of processing elements used. For the XT4 a processing element is the number of cores; for the X1E it is the number of MSPs and for the X2 it is the number of X2 processors.

(b) The overall speed of the code, in GFLOP/second, against the number of processing element used.

Figure 3.1: A comparison of the performance of the BenchC Sphere_Big benchmark between the Cray XT4, X1E and X2 systems.
For the XT4 a better than ideal scaling displayed between 32 and 512 cores. The execution time on 8 cores is 10041 seconds, whereas on 512 this has reduced to 96 seconds which is 103 times faster. For ideal scaling we would expect a speedup of 64. As the problem size decreases the mesh becomes suited to the XT4’s cache structure. This reduces the number of memory accesses and may explain the better than ideal scaling observed. Between 512 and 1024 cores the scaling is poor as communication between processes dominates over the amount of computation per process.

Ideal scaling is achieved on the X1E between 8 and 256 MSPs for both the MPI and UPC versions of the code. At 256 MSPs the MPI and UPC versions have speedups of 33 and 36 respectively over 8 MSPs. This shows that the code is efficiently utilising the extra MSPs. Above 256 MSPs a drop off in the code’s scalability is observed, due to the decrease in the ratio of computation to communication. However, the UPC scales worse than the MPI between 256 and 512 MSPs. The speed (Figure 3.1(b)) at 512 MSPs for the UPC version is 1231 GFLOPs compared to 1684 GFLOPs for the MPI. This indicates that the UPC version is affected more than the MPI with the increase in communications (this is discussed further in Section 4.1.1).

Similar scaling to the X1E was achieved on the X2 between 8 and 64 processors. At 64 processors, both the MPI and UPC version achieves a speedup of 7 relative to 8 processors (an ideal speedup at this processor count is 8).

Figure 3.1(a) also shows the cost of running the code. The y-axis, time*number of PEs, is the cost, and a low value for this indicates a high performance.

BenchC performs better on the X1E and X2 vector systems than on the scalar XT4 system. Using 8 PEs the vector systems both perform approximately 7 times better than the XT4. On 512 XT4 cores the cost is lower than on 8 cores because of the better than ideal scaling. The number of AUs used reduces from 154 AUs on 8 cores to 100 AUs on 512. However, the X1E vector system is still performing 3.5 times better than the XT4 on 512 processing elements. This suggests that BenchC is being successfully vectorised on the vector systems.

The X2 has both a similar cost and speed, Figure 3.1(b), to the X1E. The X2 performs slightly better than the X1E, with the X2 MPI version of the code achieving a speed of 144 GFLOPs on 64 processors compared to a speed of 132 GFLOPs when 64 X1E MSPs were used (the UPC versions have speeds of 154 and 129 GFLOPs on the X2 and X1E respectively). This improvement in speed may be explained by the faster vector processors on the X2; 1.6 GHz as opposed to 1.13 GHz on the X1E.

1The speedup (S) is defined as the time taken to execute the code one processor, divided by the time taken to execute the code on n processors. $S = \frac{T_1}{T_n}$
3.2 PDNS3D

3.2.1 Description of Benchmark

The PDNS3D distribution comes with three different benchmarks, T1, T2, and T3, each of which simulates turbulent flow in a channel. Each benchmark uses a different sized mesh giving different memory requirements. For performance tests on the three target systems the largest of the benchmarks, T3, is used. This has a mesh size of 360x360x360 and a total memory requirement of 128 GB.

The input script file which comes with the T3 benchmark specifies the following parameters for the simulation.

- The name and the location of the grid file, `data/fchan_360cube.bin`.
- Use of the master I/O strategy for the data gathering stage.
- Use of asynchronous sends and receives for inter-process communication.

For performance analysis the number of simulation time steps in this input script file is varied.

3.2.2 Porting

The Fortran source files and a makefile are within the `new/` directory of the PDNS3D distribution.

The code is used extensively on the HECToR XT4 service [46] and performance tests have been carried out on the Cray X1E [31]. Thus, build rules for the XT4 and X1E already exist in the makefile. On the X1E the code can be compiled in either the MSP or SSP mode.² It should be noted that MSP mode is the default compilation option on the Cray X1E and is the mode in which all other application codes are run. For the Cray X2 system a new set of build rules were added to the makefile.

Initially the `Jacobs_ch` program needs to be compiled and run on each platform to generate the grid for the T3 benchmark. In the `Jacobs_ch.F` source file the following parameters are specified:

- The problem size of the mesh for the benchmark,
  `parameter(nx=360, ny=360, nz=4)`.
- The name of the grid file,
  `open(57, file='fchan_360cube.bin',form=unformatted')`.

This program is then compiled with the command

```
make pdns3d-craxy1-mpi PROGRAM=Jacobs_ch EXE=Jacobs_ch
```

²In MSP mode the compiler tries to automatically distribute work across the 4 SSPs of an X1E MSP. In SSP mode all four processors in an MSP are independent.
and executed sequentially on the target machine. The compute nodes of the X1E and XT4 can be logged into directly. Therefore the grid generation program can be executed from the command line. However, this is not possible on the X2 and so it is necessary to submit a batch job to run the sequential program on one X2 processor. On completion of the Jacobs_ch the output file, fchan_360cube.bin, is generated.

Correctness testing of the code on the three target systems was carried out by attempting to compile the code, running the benchmark and comparing the results to a sample output file which accompanied the code.

The code was compiled with static memory allocation for 32 processors. To do this, the size of the processor grid was set to 4x4x2 in the incl3d.F source file. On the XT4 the parallel code compiled successfully, ran and generated the correct output.

On the X1E two types of error were returned during compilation. Firstly, there were a series of #ifdef CRAY directives in code. These are used on older Cray platforms (such as the Cray T3E) to call the asnunit library routine which assigns attributes to a Fortran unit number. However, this is not supported, and also is not needed, on the Cray X1E. Thus the #ifdef CRAY directives were removed for the X1E version of the code.

The second error involved the call made to the mclock() routine in the timer2.F source file. The mclock() routine is not supported by the X1E Fortran compiler. To solve this problem the call to mclock() was replaced by a call to the more portable MPI_Wtime() function. With these changes applied the code compiled and executed without error.

The code used on the X1E was then used to successfully generate an executable on the X2. However, a runtime error occurred when the code was executed. A traceback, which details the runtime errors, was enabled using the TRACEBK=30 environment variable. This reported that an invalid floating point operation occurred in the ent_euler subroutine.

This may have been the result of non-standard floating point operations caused by compiler optimisations. Thus, some of the compiler optimisations (by setting the compiler flags to -O0, -Ovector0 and -fp0) were turned off. However, this did not eliminate the runtime error. As the Totalview debugger was not working correctly on the X2 at the time of writing, a more detailed debug of the code could not take place. Due to time constraints porting of the code to the X2 could not be continued at this time.

With successful correctness tests carried out on the XT4 and X1E a series of executables were generated for a range of PE counts. This was done using the compT3 and compile scripts.

3.2.3 Performance Analysis

The T3 benchmark was used to measure and compare the performance of PDNS3D on the Cray XT4 and X1E systems. Two runs of the benchmark were performed, one for
Figure 3.2: A comparison of the performance of the PDNS3D T3 benchmark between HECToR XT4 (diamonds) and Phoenix X1E vector system (squares for MSP mode, triangles for SSP mode). The X-axis shows the number of PEs used - for the XT4 this is the number of cores; for the X1E it is the number of MSPs.

50 time steps and the other for 100. To eliminate the setup stage overhead and OS noise the runtime for the 50 time step benchmark was subtracted from that of the 100 time step benchmark.

Figure 3.2 shows the cost (the total runtime of the PDNS3D T3 benchmark multiplied by the number of processing elements) for the Cray XT4 and X1E systems for a range of processing elements. Two plots are shown for the X1E system, one for the code running in MSP mode and the other for the code running in SSP mode.

Due to the memory requirements of the T3 benchmark the code could only be run on 32 processing elements or greater. On the XT4 system the PDNS3D benchmark was run on a range of cores between 32 and 1024. On the X1E the MSP version of the code was run between 128 and 1024 MSPs. In SSP mode the code was run between 32 and 256 MSPs, which is equivalent to 128 and 1024 SSPs (with four SSPs in an MSP).

In Figure 3.2 the scaling of the code with increasing processor counts can be seen. Ideal scaling would correspond to a horizontal line which indicates efficient use of processors.

For the XT4 good scaling is achieved as shown by the near-horizontal line. At 1024 XT4 cores a speedup of 31.5 relative to 32 cores is achieved. Ideal speedup at this
processor count would be 32. This suggests that even at these high processor counts the problem size on each process is still large enough to mitigate the increased amount of communication between processes.

Similarly good scaling is seen for the code running in MSP mode on the X1E. At high MSP counts a slightly better than ideal speedup is seen. For 576 MSPs a speedup of 19 relative to 32 MSPs is achieved (ideal speedup would be 18). As the domain size on each process gets smaller it may be fitting into the cache structure of the X1E, explaining the better than ideal speedup observed.

In SSP mode super-linear speedup is observed as the number of SSPs increase. This may be explained by the large runtime which the code has with a small SSP count. As the number of SSPs increase we see the benefits of the parallel code as the ratio of computation to communications becomes more optimal. Thus, when the code is run in SSP mode it performs better at higher processor counts.

Figure 3.2 also allows us to the performance of the code on each system. On the X1E, for low processor counts (32 and 64 MSPs), MSP mode is much better than SSP. For 64 MSPs the performance of running the code in MSP mode is approximately twelve times better than SSP mode. However, for 128 MSPs and higher the cost of SSP and MSP mode are similar. In general the MSP version of the code is performing better than the SSP version. This suggests that for PDNS3D, and code with similar characteristics, it is better to run in MSP mode (i.e. allow the compiler to attempt to automatically stream loops across SSPs).

Comparing the performance of the code on the XT4 and on the X1E (in MSP mode) we see that the X1E is a factor of 4 better than the XT4. The better performance on the X1E suggests that the code is being vectorised. On inspection of the source files we can see that the majority of the code consists of loops over three dimensions of the problem domain. These loops are amenable to vectorisation by the X1E compiler. Also calculations performed in these loops involve large arrays which require the data to be constantly fetched from memory. This means that the code is reliant on the memory bandwidth of the executing system. Thus the better performance seen on the X1E vector system may also be explained by the higher bandwidth (34 GB/second) when compared to the XT4 (12 GB/second). The results shown here agree with previous performance studies [31] of the code.

### 3.3 Incompact3D

#### 3.3.1 Description of Benchmark

A sample input file which created a simple turbulent flow simulation accompanied the Incompact3D distribution. This input file sets the following parameters for the simulation:
• Flow parameters - the Reynolds number is set to 300, the maximum and minimum velocity of the flow is initialised to 0 m/s and the size of a time step is set to 0.004 seconds.

• Flow configuration - the number of time steps is set to 100, the use of grid stretching is disabled and the ODEs are solved using the Adams-Bashforth technique.

• File parameters - the name of the backup file is set to pouet and the frequency of writing to this backup file is set to 50 iterations.

• Body forces - the radius of the body force is set to 0.5 and given a center position of (5.0, 6.0, 0.0).

The code was initially setup to run on 256 processing elements. However, the number of PEs which the code runs on can be varied by altering the size of the data arrays. This is done by changing the values of the ny, nz, nyb and nzb variables, such that the ratio of ny/nyb and nz/nzb are both equal to the number of processing elements. These variables are found on line 9 and line 13 of the incompact3d.f90 source file:

9: integer, parameter :: nx=1537,ny=512,nz=512
13: integer, parameter :: nyb=2,nyb=2,mzb=nzb+2,nzbm=nzb

3.3.2 Porting

All Fortran source files and a sample makefile are in the top level directory of the Incompact3D distribution. To port the code to a new architecture the makefile is edited with system specific options. The initial version of the makefile did not contain a clean option. Thus, a simple clean rule, removing all object and module files, was added:

clean:
    rm -f *.o *.mod incompact3d

To port to the Cray XT4 a Makefile.XT4 was created from the sample makefile. The ftn Fortran compiler is specified and the -fast flag was added for optimisation. -fast enables a set of common optimisations which give good performance for a wide range of applications. The optimisations include:

• setting the optimisation level to 2 (-O2);
• loop unrolling (-Munroll=c:1)
• and inlining of procedure calls (-Mautoinline).

For the Poisson solver a call to the rfftwnd_f77_mpi routine is made. This a 3 dimensional single precision distributed memory FFTW 2 library call. On the HECToR XT4 the single precision FFTW 2.1.5 library is already installed. It is loaded using the module load fftw command before compilation. Following this the code can then be successfully compiled and run without error.
On the Phoenix X1E only the double precision FFTW 2 library existed. Thus it was first necessary to install the single precision version. This procedure is detailed in Section 3.5. A Makefile.X1E was then created which uses the ftn Fortran compiler and links to the newly installed single precision FFTW library. At this stage an executable was successfully generated.

A sample run of the executable resulted in a floating point exception. The location of where this exception occurred in the code was unclear. Thus, the code was then compiled using the -hieee_nonstop flag to turn off trapping of floating point exception. This enabled the code to run to completion. From running the newly compiled code it was seen that the setup stage executed correctly. However, the velocity of the flow outputted at each time step contained floating point NAN exceptions. This suggested that there was a problem with calculations being performed during the time marching stage of the code.

To attempt to identify the source of the error the Totalview parallel debugger was used to trace the values of the main data arrays through the code. From this debugging it was noticed that the tr array, used as input to the FFTW routines, contained some NAN values in the X1E version which were not present for the XT4 version. All variables which were used to modify the tr array were then traced. A series of collapsed loops were found in the navier.f90, convdiff.f90 and forcage.f90 source files. These loops modified three dimensional arrays using a single loop rather than triple nested loops. These loops were changed to the standard triple loop form as it was suspected that the compiler may not support such collapsed loops.

For example, a loop from the divergence subroutine, which modifies the ppm array, has the collapsed form:

```fortran
nxyz=nxm*nym*nzbm
do ijk=1,nxyz
   ppm(ijk,1,1)=sy4(ijk,1,1)+sy5(ijk,1,1)+sy6(ijk,1,1)
enddo
```

This array is subsequently assigned to the tr array in the slfft3d_shift subroutine. The modified triple nested version of the loop has the form:

```fortran
k=1,nzbm
   do j=1,nym
      do i=1,nxm
         ppm(i,j,k) = sy4(i,j,k) + sy5(i,j,k) + sy6(i,j,k)
      enddo
   enddo
enddo
```

However, with these changes the code still generated floating point errors. Instead of continuing to debug the code it was decided to wait until the X2 vector system came online and to test this code on this.
On the HECToR Cray X2 there was no FFTW library installed on the system. Again the FFTW 2.1.5 library was installed (see Section 3.5) before attempting to compile. The same build rules as for the X1E system were used to create the X2’s makefile. With this, the original code (i.e. with collapsed loops) successfully compiled on the X2. When executed there were no runtime errors and results returned matched those from the XT4.

With the code now compiling and executing without error on both the X2 and XT4 systems a performance comparison of the code on the vector and scalar systems could take place. For this it was necessary to generate a series of executables for different processor counts. As mentioned, this is achieved by editing the values of the $n_y$, $n_z$, $n_yb$ and $n_zb$ variables. As there are only 112 processors on the Cray X2 system it was decided to conduct performance runs for between 4 and 64 processing elements. The values of the above mentioned variables for the different processor counts are shown in Table 3.1.

<table>
<thead>
<tr>
<th>No. PE</th>
<th>$n_x$</th>
<th>$n_y$</th>
<th>$n_z$</th>
<th>$n_yb$</th>
<th>$n_zb$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1537</td>
<td>128</td>
<td>128</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>1537</td>
<td>128</td>
<td>128</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>1537</td>
<td>128</td>
<td>128</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>32</td>
<td>1537</td>
<td>128</td>
<td>128</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>64</td>
<td>1537</td>
<td>128</td>
<td>128</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3.1: Values of $n_x$, $n_y$, $n_z$, $n_yb$ and $n_zb$ variables for different processor counts

### 3.3.3 Performance Analysis

Once ported the code was run on both the XT4 and X2 systems for between 4 and 64 processing elements. At each processor count two runs of the code are made, one for 50 time steps and another for 100. The two execution times are then subtracted. This helps to eliminate any overheads from the setup stage or due to OS noise.

In the code each process reports its execution time. Thus, each process may report different execution times, which is not ideal for performance analysis. To guarantee that the reported time is governed by the process which takes the longest to execute, a set of `MPI_BARRIER` calls were added to the code before the start and end time are taken.

Figure 3.3 plots the total PE time (the total execution time of the code multiplied by the number of processing elements) against the number of processing elements used. Again we can analyse the scaling of the code from the graph where ideal scaling is indicated by a horizontal line.

For the XT4, better than ideal scaling is observed at 32 cores when compared to 4 cores. The cost of running on 32 XT4 cores is more than two times cheaper than running on 4 cores. As the core count increases to 64 cores a deterioration in the scaling is seen,
**Figure 3.3:** A comparison of the performance of Incompact3D between the HECToR XT4 scalar system (diamonds) and HECToR X2 vector system (squares). The X-axis shows the number of PEs used - for the XT4 this is the number of cores; for X2 it is the number of processors.

with an increase in cost of 1.7 and 3.9 when compared to 4 and 32 cores respectively. This suggests that the optimum balance between communications and computation for this problem size is achieved with 32 cores.

On the X2 a close to ideal scaling, as seen by the horizontal line, is achieved up to 32 processors. With 32 X2 processors the run time for 100 time steps is 229 seconds. This is six times faster than the execution time of 2089 seconds on 4 processors. This speedup deteriorates slightly when 64 processors are used, as inter-process communication begins to dominate over computation.

Comparing the performance of the code on the two systems, we see that up to 16 processing elements the X2 vector system is approximately 4 times better than running on the XT4 scalar system. For 32 processing elements the improvement seen in the XT4 scaling reduces the vector-scalar performance difference to be 1.5 times better on the X2 vector system. For 64 PEs both systems see a drop off in the speedup. However, the drop off is greater on the XT4 resulting in the X2 performing 4.7 better at this PE count. The better performance of the code on the X2 suggests that the most time consuming parts of the code are being vectorised successfully.
3.4 LAMMPS

3.4.1 Description of Benchmark

The LAMMPS distribution comes with a number of benchmarks to test the performance of the code. Each benchmark represents a different physical system and uses different force and interaction solvers. This allows for a comprehensive test of the correctness and performance of the entire code. Each benchmark has 32,000 atoms and runs for 100 time steps by default. The number of atoms simulated can be increased by replicating the system in the X, Y and Z directions while the number of time steps can be changed in the input script.

To compare the performance of LAMMPS on the XT4 and the X1E the Rhodo benchmark was used. This benchmark models the Rhodopsin protein in a solvated lipid bilayer [37]. Each atom in the protein has a neighbour list of 440 atoms and is modelled with the CHARMM force field. Long range interactions are approximated by the Particle-Particle Particle-Mesh (PPPM) method.

On completion of the Rhodo benchmark, a log file and an output file are returned. The log file contains information about the simulation setup such as the number of atoms, the number of bonds, the size of processor grid in each dimension and any errors that may have occurred. The output file gives information on the memory usage of each processor, the thermodynamics (temperature, energy and volume) of the system at regular intervals during the simulation and a summary of the amount of time spent in the various routines of the code. The loop time, which measures the total time spent in the main MD loop, is also written to the output file.

It is also possible to perform checkpointing at regular intervals during the simulation by specifying the restart tstep option in the input script. This writes the positions and energies of the atoms to a restart file every tstep time steps such that the simulation can be restarted if necessary.

3.4.2 Porting

The latest distribution of LAMMPS, version dated 21st May 2008, was downloaded from http://lammps.sandia.gov/download.html. The C++ source and header files for LAMMPS are within the src/ directory of the download.

This directory also contains a top-level makefile, which controls the overall compilation of the code, and a MAKE sub-directory with low-level makefiles for specific systems. These low-level makefiles have a name of the form Makefile.systemName.

The code was already ported to the HECToR XT4 system and a Makefile.hector already existed [47]. To use the PPPM method in LAMMPS it is necessary to perform a 1D FFT. Therefore the code must be linked to the FFTW 2 library using the
### Optimisation flags

<table>
<thead>
<tr>
<th>No flags</th>
<th>27.84</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-O3</code></td>
<td>27.80</td>
</tr>
<tr>
<td><code>-O3 -haggress -hfp3</code></td>
<td>27.65</td>
</tr>
<tr>
<td><code>-O3 -haggress -hfp3 -hipa4</code></td>
<td>27.05</td>
</tr>
<tr>
<td><code>-O3 -haggress -hfp3 -hipa4 -hvector3 -hstream3</code></td>
<td>26.63</td>
</tr>
</tbody>
</table>

**Table 3.2:** Time spent in main MD loop for different optimisation flag combinations using the Rhodo benchmark, simulating 32,000 atoms for 100 timesteps on 64 X1E MSPs.

-DFFT_FFTW pre-processor flag. On the XT4 the necessary FFT libraries are loaded using the `module load fftw` command before compilation.

To port to the X1E vector system a file called `Makefile.x1e` was created in the MAKE directory using the HECToR XT4 makefile as a template. The Cray C++ compiler, `CC`, is used and the code is linked with the mpi (`-lmpi`) and math (`-lm`) libraries. Again, the FFTW 2 library is loaded with the `module load fftw` command before compiling the code. When the code was attempted to be compiled at this stage a compile error was reported. The error specified that the code contained variables declared within `for` statements that had already been declared in the current scope. The code snippet below shows an example of this type of error.

```c
void fooClass::foo(int input)
{
    int i; /* declare variable i */
    .
    .
    /* i declared again within for statement */
    for ( int i=0; i < MAX_ITER; i++ )
```

This error was resolved by compiling with the `-h new_for_init` flag. This allows for variable declarations within `for` statements, by defining a new scoping rule for such variable declarations.

With this change the code compiled without error and produced an executable. A sample run of the Rhodo benchmark, using the default input script which models 32,000 atoms over 100 time steps, was conducted on 64 X1E MSPs. The output for this run was compared to and matched a sample output which was supplied with the distribution.

At this point some optimisation flags were added to the makefile in an attempt to reduce the execution time of the code. Table 3.2 shows the execution time for various optimisation flag combinations. With each new optimisation, the output file was compared to that of the original run to ensure that optimisations did not result in errors in the code.

The combination of flags which achieve the best run time are `-O3 -haggress -hfp3 -hipa4 -hvector3 -hstream3`. For subsequent performance tests on the X1E
the code will be compiled with these flags.

To port to the X2 the same makefile, including optimisation flags, as on the X1E was used to create a `Makefile.x2`. The code compiled and executed successfully with this.

### 3.4.3 Performance Analysis

To benchmark performance on the X2, X1E and XT4, a 256,000 atom Rhodopsin particle was modelled. This was achieved by replicating the system twice in the X, Y and Z directions.

Two runs of the benchmark, one for 100 time steps and the other for 200, on each system are taken and the loop time noted. These times are then subtracted to get rid of any overheads which may be caused at the initialisation phase or due to instabilities during runtime such as OS noise.

Figure 3.4 compares the performance of LAMMPS on the three target systems. It shows the cost (the time spent in the main MD loop multiplied by the number of processing elements) for different PE counts. For the X1E a processing element is the number of MSPs, for the XT4 it is the number of cores and for the X2 it is the number of processors. As well as the cost the graph gives an indication of the scalability of the code on each system, with a horizontal line indicating ideal scaling.

On the X2 vector system the Rhodo benchmark was run between 4 and 112 vector processors and between 4 and 600 MSPs on the X1E. On the HECToR XT4 the benchmark was run between 8 and 1024 cores. However, the 1024 cores run did not complete successfully. The setup of the program ran without any problem but the simulation would then timeout. A study of previous versions of LAMMPS found that the code deadlocked for more than 512 processors [48]. This was fixed, however it is not clear if this has been added to newer versions of the code.

LAMMPS exhibits very good scaling on the XT4. This is seen by the near horizontal line of the XT4’s plot. 64 cores has a speedup of 7.85 relative to 8 cores where the ideal speedup is 8. A slight tail off in this speedup occurs as the number of cores increases. For 512 cores a speedup of 38 is obtained (an ideal speedup relative to 8 cores would be 64). A reason for this may be because the problem size is small at this high PE count and the proportion of communication to computation increases.

On the X1E close to ideal scaling is seen up to 64 MSPs. Above 64 the speedup decreases as the processor count goes up. When 600 MSPs are used a speedup of 122 is seen with respect to 4 MSPs (ideal speedup at this MSP count is 200).

On the X2 vector system good scaling is observed across all processor counts. With 112 processors a speedup of 26 relative to 4 processors is achieved where an ideal speedup would be 28.
**Figure 3.4:** A comparison of the performance of the LAMMPS Rhodopsin benchmark between HECToR XT4 (diamonds) and Phoenix X1E vector system (squares). The X-axis shows the number of PEs used - for the XT4 this is the number of cores; for the X1E it is the number of MSPs.
From Figure 3.4 we see that LAMMPS performs better on the XT4 compared to the two vector systems. The code performs at least a factor of 8 times better on the XT4 than on the X1E and close to 5 times better than on the X2.

LAMMPS, and MD applications in general, need to regularly update atom positions and forces based on the interaction with other atoms. As the simulation progresses this requires access to information for different atoms which are scattered throughout memory. Thus, the code has an irregular memory access pattern, needing to load atom information from different parts of memory quickly, i.e. low latency. Vector systems cannot easily vectorise codes with such irregular access patterns. On the Cray X1E and X2 this means the code will run on the slower scalar processors. All three systems have similar low latency memory access. The performance of the code on the X1E and X2 are therefore limited by running on the slower processor. This explains the good performance on the XT4 system over the two vector systems.

Comparing the performance of the code on the X2 and X1E it is seen the the X2 performs approximately 3 times better than the X1E. This may suggest that some of the more time consuming methods, that fail to vectorise on the X1E, are being successfully vectorised by the X2 compiler. However, as will be seen in Section 4.4, this is not the case. The improvement in performance on the X2 is due to the faster scalar processor compared to the X1E (800 MHz versus 565 MHz on the X1E).

3.5 Fast Fourier transform

The Fast Fourier Transform (FFT) is a mathematical tool used in many areas of science which allows a problem to be solved more easily. It is a linear transform which converts temporal or spatial information into information which lies in the frequency domain [32]. The Discrete Fourier Transform (DFT) is expressed by the formula:

\[ F(n) = \sum_{k=0}^{N-1} f(k) e^{-2\pi i \frac{kn}{N}} \]

As seen from the formula each of the N points of the DFT is calculated in terms of all the N points in the original function. Thus, a naive DFT calculation would have complexity \( O(n^2) \). The FFT adapts this calculation using a divide-and-conquer method to divide the problem in sub-problems [49]. This allows the FFT calculation to work in \( O(n \log n) \) time.

A number of FFT libraries exist which provide an easy to use interface for program developers. This saves them the trouble of having to write the FFT routine for themselves. Also, the library routines are often very well optimised and give a much better performance than user written FFT code.

One such library used by many codes is the FFTW [35]. This is a portable and self-optimising library written in ANSI C language and is available for free download under
the GNU Public License. The LAMMPS and Incompact3D application codes use version 2 of the FFTW library. LAMMPS uses a series of 1D fft calls to find the positions and velocities of particles which interact at long range. Incompact3D uses single precision 3D fft calls to find the pressure of fluid flow at each time step. It uses MPI calls to perform the underlying communication to get the data for the FFT on each process.

### 3.5.1 Installation of FFTW library on Cray vector systems

On the X1E vector system only the double precision FFTW2 library was installed. Therefore, to compile Incompact3D it was necessary to install the single precision library with distributed memory options enabled. Also, the FFTW2 library was not installed on the X2 vector system. Thus, it was necessary to install this in order to get both Incompact3D and LAMMPS working on the X2. The steps required to install the FFTW2 in single and double precision are described below.

- The latest version of the FFTW2 library, version 2.1.5, was downloaded from http://www.fftw.org
- The `configure` script was run to configure the install of the FFTW in double precision. For this the command used was:
  ```
  .configure F77=ftn CC=cc CFLAGS= -host=cray
  -prefix=/home/d04/d04/s0792366/FFTW/2.1.5 -enable-mpi
  -enable-threads -enable-type-prefix
  -no-create -no-recursion.
  ```
  The `-enable-type-prefix` flag is used to install the FFTW libraries and header files prefixed with a character `d` for double precision or `s` for single precision.
  The `-prefix` option specifies an installation directory other than default (`/usr/local`). `-host=cray` specifies that the library is being built for a different architecture than the system on which the configure script is being run. This was needed as the X2 was only accessible through XT4 service node cross compiler. `-enable-mpi` and `-enable-threads` specify that parallel FFTW routines, for both distributed and shared memory systems, should be installed.

- The `make` command was then issued to compile the FFTW source files.
- A `make install` was used to install the FFTW, producing the double precision library and header files in the directory specified by the `-prefix` flag.
- To then install the FFTW in single precision a `make clean` was first used to remove all object files from the double precision install. The same configure command as above was then used with the `-enable-float` added to specify single precision library and header files. The `make` and `make install` commands are then used as for the double precision install.
Chapter 4

Profiling

Profiling allows us to identify which parts of a program are responsible for the most execution time. This helps in explaining the performance of the code on a particular system.

Profiling usually works by program counter sampling where the program is interrupted at regular intervals and the instruction currently being executed is recorded. From this it is possible to compute the relative amount of time spent in each procedure. A number of tools also use hardware counter profiling. This analyses events such as the number of cache misses or floating point operations performed.

The information obtained from a profile of the code, combined with results from performance analysis, help identify areas of a code that are potential candidates for optimisation. For this project, it will also help to better understand any differences in performance of the code on the vector and scalar systems.

It should be noted that there are some problems associated with profiling tools. Firstly, a significant increase in the execution time occurs due to profiling overheads. Also, the resolution of program counter sampling is small (the sampling interval is usually between 10 and 20 milliseconds) compared to the clock rate of the processor. Thus, a relatively long runtime is needed to get a good statistical representation of the code.

On Cray systems the CrayPat [50] performance analysis tools allows for a variety of profiling experiments including sampling, tracing and hardware counter analysis. CrayPat is installed and working correctly on the Phoenix Cray X1E and HECToR XT4 systems. However, at the time of writing CrayPat was not working correctly on the HECToR Cray X2. Thus, a profile of the application codes could not be carried out for this system. The steps needed to perform a performance analysis of a code using CrayPat are detailed below.

1. The CrayPat module is first loaded using the module load craypat command (or module load xt-craypat on the HECToR XT4). This command needs to be performed before compiling the code to be sampled.
<table>
<thead>
<tr>
<th>Procedure</th>
<th>% execution time on X1E</th>
<th>Time (seconds)</th>
<th>% execution time on XT4</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>blk3DSDV</td>
<td>64.6</td>
<td>53.78</td>
<td>70.9</td>
<td>401.09</td>
</tr>
<tr>
<td>agmresSD</td>
<td>14.7</td>
<td>12.24</td>
<td>14.5</td>
<td>82.03</td>
</tr>
<tr>
<td>blk3DSDR</td>
<td>4.3</td>
<td>3.58</td>
<td>4.1</td>
<td>23.19</td>
</tr>
<tr>
<td>rib_algor</td>
<td>5.2</td>
<td>4.33</td>
<td>0.1</td>
<td>0.57</td>
</tr>
<tr>
<td>blocker</td>
<td>3.2</td>
<td>2.664</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.1: Breakdown of the time spent in different sections of the BenchC code running the Sphere_Big benchmark for Cray XT4 and X1E systems.

2. The `pat_build` command is then used to instrument the program. This inserts analysis instructions at various points in the program. For example, to instrument a program to perform a profiling experiment use the command:

   `pat_build exeName profInst_exeName`.

3. The instrumented executable is submitted as normal using the `aprun` command. This generates a series of experiment data files of the form:

   `profInst_exeName.PID.xf`.

   One may want to export the `PAT_RT_EXPFILE_PER_PROCESS = 1` environment variable so that each process writes to a separate experiment data file.

4. Using these experiment files generate the profile report using the `pat_report` command. On the X1E it is necessary to directly log into the Phoenix vector system to execute this command.

Another tool which is useful in explaining the performance of an application on a vector system is loopmark listings. This is a compile time option which produces annotated source code listings describing the optimisations performed during compilation i.e. which parts of the code were vectorised and multi-streamed. It also gives explanations about why optimisations could not be performed.

To generate loopmark listings in Fortran the code is compiled with the `-rms` flag, while the `-hlist=ms` is used for C code.

### 4.1 BenchC

Table 4.1 shows the output from a profile of BenchC using CrayPat on the XT4 and X1E systems. The profile is made by running the code with the Sphere_Big benchmark on 64 processing elements for 30 time steps. The execution time for the benchmark with profiling was 565.71 seconds on the XT4 scalar system and 83.25 seconds on the X1E vector system.

The `blk3DSDV` is the most time consuming function on both the vector and scalar system. On the X1E this function accounts for 64.6% of the execution time and 70.9% on
the XT4. This function performs a matrix-vector calculation. The calculation is performed within a triple nested loop, which on inspection of the X1E’s loopmark listing output, is being fully vectorised and multi-streamed. This explains the good efficiency and performance seen on the vector systems compared with the XT4.

The \texttt{agmresSD} function is the GMRES iterative solver for the set of linear equations formed during the \texttt{blk3DSDV} function. This function accounts for approximately 15\% of the execution time on both the X1E and XT4 systems. It contains lots of short loops which the compiler is able to unroll and vectorise with a single vector iteration. A single vector iteration is when all the data in the loop fits completely into the vector registers and all instructions are executed without another vector load. There is a loop at the end of this function which sends the residual (as calculated in the GMRES Krylov subspace method) to each process. As shown below from the loopmark listing output, this loop is unable to be vectorised as it contains a call to a broadcast routine.

\texttt{CC-6287 CC: VECTOR File = agmresSD.c, Line = 246}
\hspace{1cm} A loop was not vectorized because it contains a call to function "my_broadcast" on line 251.

\texttt{CC-6734 CC: STREAM File = agmresSD.c, Line = 246}
\hspace{1cm} A loop was not multi-streamed because it contains a call to function "my_broadcast" on line 251.

The \texttt{blk3DSDR}, which forms the equations for use in the GMRES solver, accounts for approximately 4\% on both the X1E and the XT4. This function is fully vectorised and multi-streamed by the X1E’s compiler.

The \texttt{rib_algor} function executes the recursive bisection algorithm to partition the mesh. The main computational sections of this function contains recurrences (a process with multiple pointers to the same memory location) and communication calls. This means that this function is unable to be vectorised and multi-streamed on the X1E vector system. Thus, this function is executed on the scalar processor. This explains why the function consumes 5.2\% on the X1E compared to just 0.1\% of the total execution time on the XT4. As this function is only executed in the setup stage of the code it will become less significant for longer simulations.

The \texttt{blocker} function, accounting for 3.2\% of the execution time on the X1E, is only executed on vector systems during the setup stage to put mesh elements into vectorisable groups. This function reorders the data to achieve better vector performance resulting in an unpredictable memory access pattern. Therefore the function can not be vectorised. Again, as the \texttt{blocker} function is only run at the setup stage, it should not impact performance for long simulations.
4.1.1 BenchC: UPC versus MPI communications on X1E

It was seen in Section 3.1.3 that the MPI version of BenchC performed better at high processor counts than the UPC version. The only difference between these two versions is within the four communication routines during the main time integration loop (the setup is still performed using MPI calls in the UPC version). Two of these communication routines, the gather and scatter, are similar. The MPI version of the gather and scatter uses non-blocking sends and receives to transfer data while the UPC version transfers data by accessing the shared memory of the other processes. For the two other communication routines, broadcast and reduction, the MPI version uses standard MPI_Bcast and MPI_Reduce calls and the UPC version uses simple UPC equivalents.

Upon completion of the code the times for the scatter, gather and the combined time for the broadcast and reduction are printed to the output file. Figure 4.1 and 4.2 compare the amount of time spent in the communication routines for both the MPI and UPC codes.

A similar timing profile is seen for the both the gather, Figure 4.1(a), and scatter routines, Figure 4.1(b). Up to 256 MSPs the UPC version of both the gather and scatter routines requires less time than the MPI version. The UPC code is 3 seconds faster for both routines. At 512 MSPs the UPC version of both routines requires more time; circa 0.5 seconds for the gather and 1.8 seconds for the scatter.

The time spent in the reduce and broadcast is shown in Figure 4.2. Between 8 and 32 MSPs, the UPC version takes longer than the MPI version. At 16 MSPs the UPC code takes 4.81 seconds while the MPI version takes 1.76 seconds. However, at 64 the UPC version takes just 1.31 seconds compared to 1.78 seconds for MPI.

On the vector system memory loads and stores are vectorised, with the memory addresses read in bulk to/from vector registers. On the X1E the size of the vector register is 64 elements. When the UPC broadcast and reduction routines are run on 64 X1E MSPs the vector processor is able to load all the data required from memory in a single cycle. This may explain the good performance of the UPC routines at this MSP count compared to the MPI version.

Above 64 MSPs the time which the UPC version spends in the reduction and broadcast routines increases with the number of MSPs used. At 512 MSPs, the UPC version takes over 7 seconds longer than the MPI version.

In summary, we see that for a high number of MSPs, the UPC version of BenchC is performing worse than the MPI version for all communication routines. The reduce and broadcast routines exhibit poor scaling which may explain the poor performance of the UPC version at high MSP counts. An attempt to optimise the UPC broadcast and reduction routines is made in Chapter 5.1.
(a) The amount of time spent in BenchC’s Gather routines for the MPI and UPC versions of the code.

(b) The amount of time spent in BenchC’s Scatter routines for the MPI and UPC versions of the code.

Figure 4.1: A comparison of the amount of time spent in the BenchC’s Gather and Scatter functions using MPI and UPC routines.
Comparison of time in Reduce & Broadcast for MPI and UPC

Figure 4.2: A comparison of the amount of time spent in the BenchC’s Reduce and Broadcast functions using MPI and UPC routines.

4.2 PDNS3D

Table 4.2 shows the output from profiling of the PDNS3D T3 benchmark on 64 X1E and XT4 processing elements. The profile was taken for a simulation of 50 time steps. On the XT4 the execution time for this was 610.56 seconds and 108.63 seconds on the X1E.

The \textit{rhs} subroutine is the most time consuming procedure on both the scalar (XT4) and vector (X1E) systems. This subroutine is called during each time step to form the Navier-Stokes equations. It consists of a series of triple nested loops over each dimension of the problem domain. On the vector system the procedure accounts for

<table>
<thead>
<tr>
<th>Procedure</th>
<th>% execution time on X1E</th>
<th>Time (seconds)</th>
<th>% execution time on XT4</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs</td>
<td>35.1</td>
<td>38.13</td>
<td>56.6</td>
<td>345.57</td>
</tr>
<tr>
<td>d1xi_2</td>
<td>8.7</td>
<td>9.45</td>
<td>6.3</td>
<td>38.47</td>
</tr>
<tr>
<td>ent_euler</td>
<td>7.48</td>
<td>8.13</td>
<td>6.8</td>
<td>41.52</td>
</tr>
<tr>
<td>d1eta_2</td>
<td>6.60</td>
<td>7.17</td>
<td>6.1</td>
<td>37.24</td>
</tr>
<tr>
<td>d1z_2</td>
<td>6.15</td>
<td>6.68</td>
<td>6.0</td>
<td>36.63</td>
</tr>
</tbody>
</table>

Table 4.2: Breakdown of the time spent in different sections of the PDNS3D code running the T3 benchmark for Cray XT4 and X1E systems.
approximately 20% less time than the scalar system.

From the loopmark listing, it is seen that the vector system successfully vectorises and multi-streams all the loops of the rhs routine. Most of the loop calculations are performed in a single vector iteration \(^1\). Also, the vector compiler recognises that most loops in the subroutine have the same iteration space. To remove some of the loop overhead and to give better reuse of cached data a number of these loops are fused into one.

The d1xi_2 subroutine accounts for 8.7% of execution time on the X1E and 6.3% on the XT4. It applies the finite difference method using a central difference scheme in the x, y and z directions. Again, all loops in this subroutine are successfully vectorised and multi-streamed by the compiler on the X1E. The vector compiler also interchanges loop nests to achieve better vectorisation. Some loops with a small iteration space are completely unrolled to reduce loop overhead.

The remaining procedures reported in the profile (the ent_euler, d1eta_2 and d1z_2 subroutines) each account for between 6 and 7% of the execution time. These subroutines are being vectorised and multi-streamed without any problems.

The profiling and compiler reports highlight that the most time consuming procedures of the PDNS3D code are vectorised and multi-streamed. This explains the better performance on the vector system compared to the XT4.

### 4.3 Incompact3D

As discussed in section 3.3.2 it was only possible to get Incompact3D correctly working on the X2 and the XT4. At the time of writing the CrayPat profiling tool was not working correctly on the HECToR X2. Although the code was not functioning correctly on the X1E it was decided to profile the code on this system, as it still gives an idea of the most time consuming procedures on a vector system.

Table 4.3 shows the output from the profile of the code running for 50 time steps using 64 processing elements on the X1E and XT4 systems. The runtime for the code on the XT4 and X1E systems were 92 seconds and 24 seconds respectively.

On both systems the largest proportion of the execution time is consumed by the slfft3d_shift routine. This routine contains calls to the FFTW library. On the X1E the FFTW calls are shown separately from the routine, however on the XT4 it is not clear if the FFTW calls are combined with the slfft3d_shift or part of a separate system calls procedure. The total percentage time of the routine on the X1E (including the FFTW) is 50.5% whereas on the XT4 the routine accounts for 7.5%. The larger figure seen for the vector system is not surprising as the FFTW library is not

\(^1\)In a single vector iteration all the data in the loop fits completely into the vector registers and all instructions are executed without another vector load.
<table>
<thead>
<tr>
<th>Procedure</th>
<th>% execution time on X1E</th>
<th>Time (seconds)</th>
<th>% execution time on XT4</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>slfft3d_shift</td>
<td>0.4</td>
<td>0.095</td>
<td>7.5</td>
<td>7.09</td>
</tr>
<tr>
<td>_fftw</td>
<td>50.1</td>
<td>12.01</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>derz</td>
<td>0.1</td>
<td>0.02355</td>
<td>4.7</td>
<td>4.33</td>
</tr>
<tr>
<td>deryl</td>
<td>0.2</td>
<td>0.05</td>
<td>4.7</td>
<td>4.33</td>
</tr>
<tr>
<td>convdiff</td>
<td>0.2</td>
<td>0.05</td>
<td>4.1</td>
<td>3.77</td>
</tr>
<tr>
<td>System calls</td>
<td>32.8</td>
<td>7.72</td>
<td>39.2</td>
<td>36.09</td>
</tr>
</tbody>
</table>

Table 4.3: Breakdown of the time spent in different sections of the Incompact3D code running on the Cray XT4 and X1E systems.

expected to vectorise completely. There are some other loops in the slfft3d_shift routine which do not have calls to the FFTW library. From inspection of the loopmark listing generated by both the X1E and X2 compilers these loops are being vectorised successfully.

Other computational subroutines consume a very small percentage of the total execution time on the vector system. The loopmark listing for the subroutines shown in Table 4.3, shows that they are also being successfully vectorised. System calls, including memory access and MPI calls such as barriers, broadcasts and alltoallV, account for 32.8% of the execution time on the vector system. This high percentage indicates that the code is performing well with most of the computational routines being vectorised. However, it may also indicate that inter-process communication and memory are dominating due to a small data size on each process.

The profile information for the vector system is used in Section 5.2 to optimise Incompact3D for the vector system.

### 4.4 LAMMPS

A profile of the LAMMPS application code using the CrayPat tool was made on both the XT4 and X1E systems. The profile was made for the Rhodopsin particle simulated over 100 time steps on 64 processing elements.

On the X1E the execution time of the benchmark with profiling was 201.38 seconds, while it was 12.46 seconds on the XT4 scalar system. Table 4.4 shows a breakdown of the amount of time spent in the main methods of the code.

The compute method in the pairLJCharmmCoulLong class is the most time consuming method on both systems. This method calculates the long range forces between pairs of particles. It consists of a double nested loop which computes the force on an atom due to its neighbours.

On the X1E vector system the compute method consumes approximately 20% more of the execution time than the XT4 scalar system. The loopmark listing for the code on
the X1E shows that the method is not being vectorised or multi-streamed. A recurrence, where the method contains multiple pointers to the same memory location, is cited as the reason for this.

It is not possible to vectorise or multi-stream a loop with recurrences as it can not be guaranteed that variable updates are independent. Thus, the code is running on the X1E’s scalar processor, which accounts for the poor performance when compared to the XT4 scalar system.

The half_bin_Newton method in the Neighbor class accounts for 14.5% of the execution time on the X1E and 4.7% on the XT4. This method, which constructs the neighbour lists thus requiring many irregular memory accesses, also fails to be vectorised or multi-streamed on the X1E system. Again, the compiler reports recurrences as the reason for failing to vectorise and multi-stream the method. The compiler also reports that some loops in the method could not be vectorised due to a call to the add_pages function. This could not be inlined with -Oipa4 optimisation flag as the add_pages function contains too many operations.

The fieldforce, make_rho and findspecial methods are successfully vectorised and multi-streamed by the X1E compiler. All these methods consist of loops which contain no recurrences or dependencies and are thus amenable to vectorisation. As seen from the profile, this reduces the percentage execution time which the methods consume compared to the scalar system. However, these methods only account for a small percentage of the overall runtime of the code. As the most time consuming methods are failing to be vectorised, the performance of the code on the vector system is still worse than that on the scalar system.

The loopmark listing shows that the X2 behaves similarly to the X1E. The compute and half_bin_Newton methods fail to be vectorised. The fieldforce, make_rho
and *find_special* methods are successfully vectorised. Thus, the performance difference between the X2 and X1E (as seen in section 3.4.3) is due to the difference in the scalar clock speeds on the two systems and not due to the vectorisation performed by the compiler.
Chapter 5

Optimisation

5.1 Optimisation of UPC communications in BenchC

Section 4.1.1 shows that the UPC version of the broadcast and reduction routines in BenchC have poor scaling on the X1E vector system. In this section we consider the performance of the broadcast and reduce routines separately, and attempt to optimise each routine and the overall performance of the UPC version of the code.

A brief introduction to UPC was given in Section 2.5.1.

5.1.1 Broadcast

The broadcast function, defined in the my_broadcast.c source file, is used to send a single double value from one process to all others.

The UPC version, Figure 5.1, of this function works by having the sending thread loop over all other threads and placing the broadcast value into a shared array. This means that the broadcast is executed sequentially by the sending process. Thus, the time spent in the broadcast will increase with the number of threads. However, the broadcast loop is fully vectorised by the compiler. This allows the sending process to perform vector memory loads and stores across the system, whereby the broadcast value is stored to many distributed memory locations at once.

The MPI version of the code calls the MPI_Bcast routine.

To compare the amount of time being spent in the broadcast routine timing calls were added to the my_broadcast.c source file. The MPI and UPC versions of the code were run on the X1E system for between 8 and 512 MSPs. Each version was run twice for 20 time steps and the minimum time noted. Figure 5.2 compares the time spent in the broadcasts.

The broadcast routine is called a total of 164 times per time step. The UPC version
Figure 5.1: Diagram showing how the original version of UPC broadcast works

Figure 5.2: A comparison of the amount of time spent in the broadcast routine of BenchC using MPI and UPC. For this the code was run for 20 time steps using the Sphere_Big benchmark.
spends less time in the broadcast routine than MPI when 8, 32 and 128 MSPs are used. However, for 256 and 512 MSPs the UPC time increases and becomes worse than the MPI version. This illustrates the limitation of this UPC broadcast algorithm with increasing PE counts, as discussed above.

Below a number of alternative algorithms for the broadcast are described and analysed.

**UPC get broadcast**

**Theory:**
This optimisation uses an algorithm, shown in Figure 5.3, whereby each UPC thread gets the broadcast value from the sending process. This is a more parallel algorithm than the original. Each thread now performs some work during the broadcast, instead of a single process sending all the data. Thus, better scalability is expected with this algorithm.

**The change:**
The algorithm is implemented by having the sending process copy the variable to be broadcast into a distinct element in a shared array. Each thread then copies this variable to its own memory. The code to achieve this is shown below.

```c
/* get from root */
global_valueSH[MYTHREAD] = global_valueSH[0];
```

**Analysis of change:**
Figure 5.6 shows the timing run for this version of the UPC broadcast (triangles). It is seen that for up to 32 MSPs the algorithm is quicker than both the MPI and the original UPC put broadcast. At 128 MSPs the amount of time in all three routines is similar. For 256 and 512 MSPs the time increases substantially and is slower than the original UPC version. As the code to implement the get algorithm is not part of a loop it is not vectorised. Thus, memory access is through scalar requests, which are slower than vector loads and stores.
Mixed mode: Broadcast using MPI_Bcast

Theory:
As seen from Figure 5.2 the MPI version of the broadcast routine exhibits good scaling at high processor counts. Clearly the MPI_Bcast routine is well optimised for the vector system.

As it is possible to use both MPI and UPC calls together in a program, we can rewrite the UPC version of the code to use MPI calls for the broadcast routine.

The change:
To apply this change the ifdef statement and UPC instructions in the my_broadcast function were removed. This is used to check if the code was compiled with UPC or MPI. Now when the broadcast function is called, the MPI_Bcast routine is called irrespective of the communication mode used.

Analysis of change:
The timing for this mixed mode of programming, crosses in Figure 5.6, shows that this routine now performs similarly well to the pure MPI version. This shows that use of mixed mode communications does not affect the performance of the call to the MPI library.

UPC broadcast using 2-level tree

Theory:
This version of the UPC broadcast uses a 2-level tree algorithm, as shown in Figure 5.4. With this the sending process first sends the broadcast value to another process. These two processes then perform separate broadcasts, each sending to half the total number of processes. As this algorithm divides the work between processes it is inherently more parallel than the original broadcast algorithm. Also, as the broadcasts take place within a loop the code will be vectorised.

The change:
The code to implement this algorithm performs the following steps.
1. Each UPC thread finds the thread with index equal to half the total number of threads, \( HWThread \). This is done by dividing the total number of threads, stored in the predefined \( \text{THREADS} \) variable by 2.

2. The sending process copies the broadcast variable to \( HWThreads \) shared memory.

3. The \texttt{upc\_barrier} routine is used to ensure that the sending thread has copied to \( HWThread \).

4. The original sending thread copies the broadcast value to threads with index from 1 to \( HWThread \).

5. Thread \( HWThread \) copies the broadcast value to all threads with index greater than \( HWThread \).

Analysis of change:
Up to 32 MSPs the 2-level tree broadcast algorithm, shown as diamonds in Figure 5.6, performs similar to the original UPC version of the broadcast. At 64 and 128 MSPs, the algorithm spends shorter in the broadcast routine than both the original UPC and the MPI version. For 256 and 512 MSPs the time spent in the routine is greater than the MPI version but still shorter than the original UPC version. At these processor counts the advantages of the two level tree algorithm over the original UPC broadcast is seen. The work of the broadcast is now being split amongst two processes with each process storing to remote memory with vector operations.

UPC broadcast using 4-level tree

Theory:
Based on the improvement for the 2-level tree UPC broadcast, this optimisation implements a 4 level tree algorithm. With this algorithm the sending process first sends the broadcast variable to four other processes. All four processes then broadcast the variable to the other processes. Figure 5.5 shows how the algorithm works.

The change:
To implement this algorithm three threads which will first receive the broadcast value
Figure 5.6: A comparison of the amount of time spent in the broadcast routine of BenchC using different UPC algorithms.

from the master are selected. To easily do this the threads with index equal to 1/4, 1/2 and 3/4 the total number of threads are calculated. The sending process then copies the broadcast value to these three threads’ shared memory. Each of the four threads then loop over 1/4 of the other threads and copy the broadcast value to each thread.

Analysis of change:
For 8 and 16 X1E MSPs the UPC 4-level tree broadcast algorithm, shown as stars in Figure 5.6, takes a long time when compared to the other versions of the broadcast. The reason for this is that the overhead in initially sending the broadcast value to the three other processes is significant. As the number of MSPs used increases, the time spent in the routine reduces and the performance is close to that seen for the 2-level algorithm. The splitting of the work between 4 processes now becomes advantageous. At 512 MSPs the 4-level tree algorithm performs slightly better than the 2-level algorithm.

5.1.2 Reduction

The reduction function, defined in the my_reduction.c, performs an all-reduce operation. It finds the sum of double precision variables held by each process and stores the result in each processes memory.
Figure 5.7 shows how the UPC version of this function works. The UPC reduction is executed sequentially by the master thread using two loops which can be vectorised by the compiler. The first loop performs a reduce operation with the result stored in the shared memory of the master process at the conclusion. The second loop is a broadcast of the reduce value from the master to all the other processes. Again, the time spent in the reduction routine using this algorithm is expected to increase with the number of processes used.

For the MPI version, a call is made to the `MPI_Allreduce` routine with sum operator.

Figure 5.8 compares the amount of time spent in BenchC’s reduction routine using both UPC and MPI. For this, the code was run for 20 time steps. The reduction routine was called a total of 18,560 times during these 20 time steps. This is equivalent to 928 calls per time step.

The UPC version of the reduction is performing worse than the MPI version for all MSP counts. Between 8 and 32 MSPs the UPC version is over one second slower than MPI. When 64 and 128 MSPs are used the UPC version shows a slight improvement, but is still approximately 0.5 seconds slower than the MPI version. A possible reason for this improvement may be that the number of MSPs used are close to the number of elements in the X1E’s vector registers. Thus, the vector registers are being used optimally at 64 and 128 MSPs. At 256 and 512 MSPs the time spent in the UPC reduction increases. The poor scalability of the algorithm is illustrated at these two MSP counts as the master process becomes overloaded with the increased amount work needed to perform the reduction.

A number of alternative algorithms for BenchC’s UPC reduction are presented in the following sections.

**Mixed mode: all-reduce using MPI_Allreduce**

**Theory:**
The time spent in the MPI version of the reduction is much less than the UPC version for all MSP counts tested. To get a performance improvement in the UPC version of the
Figure 5.8: A comparison of the amount of time spent in BenchC’s reduction routine using MPI and UPC. For this the code was run for 20 time steps using the Sphere_Big benchmark.
code, it should be possible to change the reduction routine to call `MPI_Allreduce` instead of using UPC calls.

**The change:**
Like the broadcast routine, to implement this change the `ifdef` statements and all UPC calls are removed from the `my_reduction.c` source file. Now both the UPC and MPI version will make a call to the `MPI_Allreduce` when the reduction routine is called.

**Analysis of change:**
Timing of this new reduction routine, triangles in Figure 5.11, shows that it is performing similarly to the pure MPI version. At 256 and 512 MSPs, it is approximately one second faster than the original UPC reduction algorithm. Again we can conclude that mixing MPI and UPC in the program does not affect the performance of the `MPI_Allreduce` routine.

**UPC all reduce using 2-level tree**

**Theory:**
This optimisation attempts to perform the two stages of the all-reduce operation, the reduction and the broadcast, using a 2-level tree algorithm as shown in Figure 5.9. Two UPC threads will first perform a separate reduce with each thread summing the data held by half the total number of processes. These two threads then add their reduce values to get the total reduction value. This total value is then broadcast to all other threads. Each of the two threads broadcast the reduce value to half the total number of threads.

**The change:**
To implement this algorithm, shown in Figure 5.9, each UPC thread first calculates the thread with an index of half the total number of threads. This is stored in a variable called `HWThread`. The master thread (thread 0) and thread `HWThread` then perform a separate reduce over half the number of threads. The master thread sums the values stored on thread 0 to `HWThread - 1`. Thread `HWThread` sums from thread `HWThread` to the total number of threads (stored in the variable `THREADS`). A bar-
rier call (upc_barrier) is then used to ensure that both threads have finished their reduce before summing the reduce values to find the total reduction value. Finally, the two threads perform a two level broadcast where thread 0 loops over and copies the total reduction value to half the threads and thread HWThread broadcasts to the other half.

Analysis of change:
Between 8 and 32 MSPs the full 2-level tree all reduce, shown as crosses in Figure 5.11, requires less time than the original UPC version. This suggests that the amount of time spent synchronising between threads is small. Splitting the reduction routine between two threads at these MSPs counts is advantageous. As the number of MSPs used increases, the time spent in the 2-level all-reduce begins to increase. At 64 MSPs the time spent for both the original and two level tree algorithm are similar. The tree algorithm is almost 0.5 seconds slower than the original for 128 and 256 MSPs. A possible reason for this is that a long time is spent waiting for threads, at the upc_barrier call, for these high MSP counts. With 512 MSPs the code did not complete and would timeout. This may have been due to the high number of processes required to synchronise at the upc_barrier call.

UPC all reduce using semi 2-level tree

Theory:
From the implementation of the previous optimisation, the full 2-level tree algorithm, it was noticed that the resulting code had a large amount of upc_barrier calls. The upc_barrier call synchronises all UPC threads. However, for the 2-level tree algorithm it is only necessary to synchronise between two threads. To reduce the amount of synchronising in the reduction routine two new versions of the all-reduce operation were implemented. The first, shown in figure 5.10(a) performed the reduce stage of the all reduce using a 2-level tree algorithm and the broadcast stage using the original sequential loop algorithm. The second version, figure 5.10(b), used the reduce stage of the original algorithm and the broadcast stage with a 2-level tree algorithm. This should reduce the number of barrier calls in the routine while still executing one of the stages of the all reduce in parallel.

The change:
By using the source from the original UPC version of the reduction and the full 2-level tree reduction the implementation of these two algorithms was straightforward.

To implement the 2-level tree for the reduce stage of the reduction routine two UPC threads perform separate reduces by looping over half of the threads. The two reduce values are then combined by the master thread to get the value of the full reduction. The master thread then broadcasts this reduction value to all the threads. This algorithm removes one of the upc_barrier calls that was in the full 2 level tree algorithm.

For the two level broadcast version, the master thread first finds the full reduction value by looping over all threads and combining the reduce value at each thread. It then copies
Combine sums
putput put put
+ + +

(a) Operation of UPC all-reduce with 2-level tree reduction step and original broadcast step.

(b) Operation of UPC all-reduce with original reduction step and 2-level tree broadcast step.

**Figure 5.10:** Diagram showing how the semi 2-level tree version of UPC reduction works.

the full reduction value to another thread (with id HWThread). The two threads each broadcast the value to half the total number of threads. Again, this algorithm removes one of the upc_barrier that was in the full 2-level tree algorithm.

**Analysis of change:**

Figure 5.11 shows the amount of time spent in the reduction routine for each version during a sample run of BenchC. The diamonds show the 2-level reduce version and stars show the 2-level broadcast version. For both versions, the time with 512 MSPs again did not complete.

The 2-level tree reduce version has a similar performance as the original UPC version. Up to 32 MSPs the full 2-level tree reduction algorithm performs better. As the number of MSPs used increases, the effect of having a fewer synchronisation points in the code is clearly seen. The 2-level tree reduce algorithm is over 0.3 seconds quicker than the full 2-level tree algorithm at both 128 and 256 MSPs.

This effect of fewer synchronisations is also seen when 128 and 256 MSPs are used for the 2-level tree broadcast version. However, at lower MSP counts this version behaves unpredictably. At 64 MSPs, it spends 0.5 seconds in the all-reduce routine, the quickest of all UPC versions, while at 32 MSPs it take over 2.5 seconds. The reason for this large variation in time is unclear. However, it may be due to the amount of time spent waiting for threads to synchronise at the upc_barrier. It thus illustrates the potential impact that these synchronisation calls may have on performance of a code.

### 5.1.3 BenchC performance with optimised UPC communications

From the optimisations performed in Section 5.1.1 and 5.1.2, it was seen that time spent in the reduction and broadcast is best when mixed mode communications are used, i.e. MPI and UPC calls are mixed. Also, the use of two level tree UPC algorithm for the broadcast routine showed an improvement over the original version of the UPC broadcast routine.
Figure 5.11: A comparison of the amount of time spent in the reduction routine of BenchC using different UPC algorithms.
A full run of BenchC was carried out to analyse if these optimisations resulted in an overall performance improvement for the UPC version of the code. Figure 5.12 shows the cost of running the code on the Cray X1E. Plots are shown for two optimised UPC versions of BenchC. The first, (diamonds) shows the mixed mode optimisation where MPI calls are used to perform BenchC's broadcast and reduction routines. The second, (squares) is for a UPC version of the code which uses a 2-level tree algorithm to implement the broadcast routine. Also shown are the costs of the original UPC (triangles) and MPI (crosses) versions of the code.

All versions of the code have a similar performance up to 256 MSPs. At 512 MSPs the 2-level tree broadcast version of the code performs worse than both the original UPC and MPI versions. The 2-level tree broadcast version uses the original UPC reduction algorithm. As seen, The code has more calls to the reduction routine than the broadcast routine (928 calls per time step versus 164 for the broadcast). Thus, the effect of the optimised broadcast routine on the performance is not seen due to the low number of calls made to this routine in comparison to the amount of calls to the reduction. On the other hand, the mixed mode optimised version of the code performs better than both the original UPC and MPI versions at 512 MSPs. With this mixed mode version all communication routines, including the original UPC scatter and gather routines, are well optimised for the vector system resulting in a good overall performance of the code.

5.2 Optimisation of Incompact3D

In this section, we attempt to optimise Incompact3D on the Cray X2 vector system. Two types of optimisation techniques are used. The first is to apply compiler optimisations, which involves adding different compiler flags to the Incompact3D makefile. The second technique is code optimisation. With this, procedures of the code which have been identified from profiling (Section 4.3) are changed in order to reduce the overall runtime of the code.

To test each optimisation applied, the code is run for 50 time steps on 64 X2 processors. The execution time is then compared to the original runtime of 23.64 seconds for the same processor count. Also, the output file is verified to be correct by comparing it to the original output file. This ensures that any errors due to an optimisation are found.

5.2.1 Compiler optimisation

The Cray Fortran compiler, ftn, has a number of different flags which can be used to reduce the runtime of the code. Here, an analysis of different combinations of these flags on the performance of Incompact3D is carried out.

First, the use of different compiler optimisation levels is analysed. The compiler optimisation level is specified with -O <level> flag. Depending on the level selected
Figure 5.12: A comparison of the performance of the BenchC Sphere_Big benchmark on the Cray X1E using optimised UPC communication routines.
(between 0 and 3) the compiler attempts to perform a range of standard optimisations on the code. These include setting the levels for the vector and scalar optimisation. It may also turn on some local optimisation such as constant folding and common subexpression elimination or global optimisations such as branch elimination, copy propagation and dead store elimination.

Next, the -Ocache\_3 flags was tested. This flag specifies that an aggressive cache management strategy should take place during compilation. It works by analysing the code and predicting areas where the possibility of cache reuse exist in order to increase the number of cache hits.

The -Ofp <level> flag sets the level of floating point optimisation performed by the compiler. The <level> variable ranges from 0 to 3 and specifies the level of allowable optimisation. Level 3 gives the maximum freedom to the compiler to optimise floating point operations. At this level however, it also is necessary to ensure the results are correct as the floating point values may not conform to the IEEE standard.

The -Oscalar\_3 flag allows the compiler to perform an aggressive level of optimisation of code which will be run on the scalar processor of the vector system.

The -Ovector\_3 flag specifies the most aggressive level for vectorisation.

The final compiler flag experimented with was the -Oipa<level> flag. This flag sets the level of inter-procedural analysis to allow optimisations such as inlining, data alignment and argument removal between subroutines. There are five ipa levels corresponding to the amount of inlining each optimisation will perform. For example, -Oipa5 specifies to inline to a depth of 4 procedure calls.

Table 5.1 shows the execution time for different combinations of the above compiler flags. For each of these flag combinations the output was compared against a set of original results.

Both the -O2 -Ocache\_3 -Ovector\_3 -Oipa\_3 and -O3 -Ocache\_3 -Oipa\_5 compiler flag combinations show the best performance improvement for the code. A full performance run on the X2 for both of these set of flag combinations is shown in Figure 5.13. For most processor counts the two optimised versions have similar performance. At 16 X2 processors both optimised versions are the same and perform better than the original version of the code. For 32 processors the -O3 -Ocache\_3 -Oipa\_5 set of compiler flags performs better than the -O2 -Ocache\_3 -Ovector\_3 -Oipa\_3 flags. However, both versions perform worse than the original version at this processor count. The -O2 -Ocache\_3 -Ovector\_3 -Oipa\_3 compiled version of the code performs better than both the -O3 -Ocache\_3 -Oipa\_5 and original versions when 64 processors are used.

### 5.2.2 Code optimisation

This section details some of the changes made to the Incompact3D in an attempt to further reduce the runtime of the code.
Table 5.1: The execution time of Incompact3D on 64 X2 processor using different combinations of compiler flags.

<table>
<thead>
<tr>
<th>Compiler flags used</th>
<th>Execution time (seconds)</th>
<th>% improvement from original</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O1</td>
<td>66.09</td>
<td>-180.0</td>
</tr>
<tr>
<td>-O2</td>
<td>23.64</td>
<td>0</td>
</tr>
<tr>
<td>-O3</td>
<td>21.37</td>
<td>9.6</td>
</tr>
<tr>
<td>-O2 -Ocache3</td>
<td>21.046</td>
<td>10.9</td>
</tr>
<tr>
<td>-O3 -Ocache3</td>
<td>20.97</td>
<td>11.3</td>
</tr>
<tr>
<td>-O2 -Ocache3 -Ofp3</td>
<td>24.57</td>
<td>-3.93</td>
</tr>
<tr>
<td>-O3 -Ocache3 -Ofp3</td>
<td>24.13</td>
<td>-2.0</td>
</tr>
<tr>
<td>-O2 -Ocache3 -Oscalar3</td>
<td>21.0141</td>
<td>11.1</td>
</tr>
<tr>
<td>-O2 -Ocache3 -Ovector3</td>
<td>20.97</td>
<td>11.3</td>
</tr>
<tr>
<td>-O2 -Ocache3 -Ovector3 -Oipa3</td>
<td>20.96</td>
<td>11.4</td>
</tr>
<tr>
<td>-O3 -Ocache3 -Oipa5</td>
<td>20.96</td>
<td>11.4</td>
</tr>
</tbody>
</table>

Why was the slfft3d_shift subroutine modified?

The slfft3d_shift subroutine is a significant contributor to the execution time of Incompact3D. The subroutine contains calls to the FFTW create plan, destroy plan, and local sizes routines. These calls are executed multiple times throughout the program execution, leading to a significant portion of the total execution time.

The Change:

To reduce the execution time spent in the slfft3d_shift subroutine, the create and destroy plan routines were moved to the start and end of the program. Additionally, a module was created to store all global variables needed for the FFTW. This allowed for easier access to the variables between subroutines. A rule was added to the makefile to build this module.

Analysis of Change:

Following the modifications, the execution time has been reduced to 20.13 seconds. The time spent in the slfft3d_shift subroutine has been reduced to 6.95 seconds.
Reduce the time spent in AlltoAll calls

Theory:
From the profile of the original Incompact3D code it was seen that MPI_Alltoallv calls account for 4.1% of the execution time. Alltoallv allows a process to exchange a different number of elements between processes by specifying the send and receive counts for each process. In Incompact3D however, the calls to the MPI_Alltoallv routine are only used to exchange the same number of elements between processes. A call to the MPI_Alltoall routine could be used to perform this operation. The MPI_Alltoall routine is expected to have a lower overhead than MPI_Alltoallv as it requires fewer checks on send and receive counts and array displacements.

The Change:
Two subroutines in the code, swap_to_xy and swap_xy_to, contain calls to the MPI_Alltoallv routine. Both of these subroutines contain a DO loop. This sets each element in the sendcount, receivecount and displacement arrays which are then passed to the MPI_Alltoallv call.

Each element in the sendcount and receivecount arrays are set to nx*nyb*nzb. As nx, nyb and nzy are pre-defined constants the DO can be removed and the sendcount and receivecount arrays can be replaced with a scalar variable which can be set to nx*nyb*nzb once. The call to MPI_Alltoallv can then be replaced with a call to MPI_Alltoall.

Analysis of change:
A sample run of the code with this optimisation on 64 X2 processors verified that the output was still correct. The optimisation reduced the 50 time step execution time slightly from 20.13 seconds to 19.80 seconds.

Optimise MPI collective routines

Theory:
The MPI programming environment contains a MPICH_COLL_OPT_ON environment variable. This environment variable enables optimisation of MPI collective routines. It uses non-default, architecture specific algorithms for collective operations such as MPI_Alltoall, MPI_Allreduce, MPI_Barrier and MPI_Bcast.

The Change:
To apply this optimisation the line export MPICH_COLL_OPT_ON is added to the PBS submission script.

Analysis of change:
By enabling the optimisation of MPI collective routines, the runtime for a 50 time step simulation on 64 processors was reduced from 19.80 seconds to 19.28 seconds.

Figure 5.13 shows a full performance run of Incompact3D with all the above code optimisations applied. The code was also compiled with the -O2 -Ocache3 -Ovector3
Optimised versions of Incompact3D on HECToR X2

Figure 5.13: A comparison of the performance of the optimised versions of Incompact3D on the Cray X2.

-Oipa3 flags. The code performs better than the original version of the code at 64 processors. For all other processor counts the optimised version code is performing slightly worse than the original. It is unknown why this is the case.

However, the performance improvement at 64 processors highlights that optimisations using compiler flags can make difference to the performance of the code. They provide a quick and easy means to get a reasonably good speedup from the code. Also, it was seen from the improvement achieved through modifying the code that profiling is a useful tool in identifying potential optimisation areas in the code.
Chapter 6

Conclusions and Future Work

The main aim of this project was to investigate and improve the performance of a number of application codes on two Cray vector systems (the Cray X1E and X2). By comparing the performance of these codes to their performance on the Cray XT4 scalar supercomputer, the characteristics of applications best suited to vector systems were identified.

Four codes were used for the performance investigation. Three of these are computational fluid dynamics (CFD) codes, BenchC, Incompact3D and PDNS3D. The fourth code, LAMMPS, is a molecular dynamics (MD) simulation code.

Initially, each code was ported to the three target systems. It was found that codes with few external dependencies were relatively easy to port to the vector systems. For example, BenchC is written in C and only uses the MPI library. Thus, it was relatively straightforward to port to both the X1E and X2 systems. Some problems were encountered with code that contained system specific variables or library calls. The PDNS3D code called the mclock() routine, which is not supported on the vector systems. This call was replaced by a call to the more portable MPI_Wtime() function.

A detailed performance analysis of each code was then undertaken for each system. In general, the CFD codes performed at least a factor of four times better on the vector systems than on the scalar XT4 system. Each of these codes also achieved close to linear scaling on the vector systems. These codes are characterised by calculations involving large arrays and many loops, which are easily vectorised (and multi-streamed on the X1E) by the compiler of the vector system.

On the other hand, LAMMPS performed better on the scalar system. On the scalar XT4, it performed at least five times better than the X2 and at least eight times better than on the X1E. LAMMPS has an irregular memory access pattern. This characteristic is typical of MD codes and makes them more favourable to high performance scalar processor systems with low latency memory access.

To understand the differences in performance across the systems and to identify areas of the code that are potential candidates for optimisation, each code was profiled. The
most computationally intensive routines in the CFD codes were vectorised, explaining the good performance of these codes on the vector systems. However, the main class methods of LAMMPS are not vectorised due to the irregular memory access pattern of the code.

Also, from the performance analysis of a UPC and MPI version of BenchC, it was found that UPC performance was slightly worse at high processor counts. Profiling of BenchC’s main communication routines found that the UPC broadcast and reduction routines exhibit poor scaling.

Based on the profiling results, optimisations were carried out on BenchC and Incompact3D. For BenchC, an attempt to improve the UPC broadcast and reduction routines on the Cray X1E was made by implementing a number of alternative algorithms. The best modification to the UPC broadcast routine used a 2-level tree algorithm. This algorithm allowed for the work of the broadcast to be split among two processes and gave fast vector loads and stores across the system. As a result, the time spent in the broadcast routine was reduced for all processor counts tested. However, using a mixed mode of programming, where MPI calls replaced the UPC broadcast and reduction routines, was found to give a better overall performance than both the original UPC and 2-level tree versions of the code.

The overall runtime of Incompact3D on the Cray X2 system was reduced by using compiler and code optimisation techniques. Compiler optimisation involved testing different combinations of compiler flags. It was found that the best improvement was obtained with the -O2 -Ocache3 -Ovector3 -Oipa3 flags. Code optimisations involved removing unnecessary calls to the FFTW create and destroy plan routines and changing MPI_alltoallV calls to MPI_alltoall. As a result of these optimisations, the execution time of the code using 64 X2 processors was reduced by 18%.

A discussion of the changes made to the original work plan is presented in Appendix A.

**Future work**

There are a number of possible areas where further work could be carried out.

During optimisation of UPC communication routines in the BenchC code, it was found that the usability of the UPC library was good. It was easy to implement different UPC communication algorithms. Converting the other codes to use UPC or CAF for inter-process communication would enable a further analysis of the usability of these libraries. It would also for a more detailed performance comparison between these libraries and MPI.

Also, the porting of the PDNS3D code to the Cray X2 system could be continued when the Totalview debugger is correctly working.

It was seen from profiling of Incompact3D (Section 4.3) that the FFTW routines consume a high percentage of execution time on the vector system. An investigation into which parts of the FFTW library are vectorised could take place in order to try to explain this. Also, some of the codes could be rewritten to use the Cray LibSci FFT routine. These are known to perform well on Cray vector systems.
Bibliography


Appendix A

Work Plan

Figure A.1 shows the changes to the work plan proposed during the Project Preparation module [8].

![Diagram showing work plan]  

**Figure A.1:** Work plan showing the changes to the initial project schedule given in the Project Preparation report. The red arrows indicate the final schedule.

Overall, the project was on schedule with just a few changes to the initial work plan. Less time was spent in the background phase of the project. The reason for this was that much of the research for the project was carried out during the Project Preparation module.

Instead, the porting phase was started straight away. It was decided to overlap the porting and performance analysis phases, since benchmarking of a code could be automated. Thus, benchmarking began as soon as the first code was ported. As part of the performance analysis, profiling of each code was also carried out.
Seven weeks of optimisation took place based on the profiling and performance results. For all optimisations, the performance optimisation cycle was employed in order to evaluate the optimisation. This cycle contains the following stages:

1. **Theory**: This involves proposing a reason as to why a code section is performing poorly.

2. **Propose and make change**: Based on the theory, this stage makes a change to the code in an attempt to obtain a better performance.

3. **Verify results**: This involves checking if the results are still correct.

4. **Iterate cycle again or exit**: This involves deciding whether to execute another iteration of the optimisation cycle or to exit. The decision is based on an analysis of the likely improvement in changing the code versus the cost to implement another change.

In the initial work plan, four weeks were scheduled for the dissertation write-up. In the end, this was changed to three weeks, without causing any problem as writing was ongoing throughout the project. Also, an interim report was produced after seven weeks. This contained a background chapter and some initial performance analysis.