UPC Collective Optimization

Savvas Petrou

August 20, 2009

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2009
Abstract

Efficient collective operations are an important feature for new parallel programming languages. The size of massively parallel machines is increasing fast and the performance of these operations are crucial to the scalability of many applications. The Unified Parallel C (UPC) language is one of the new parallel programming languages based on the Partitioned Global Address Space (PGAS) model. The purpose of this project is to investigate the performance and scaling of the current implementations of collective operations and also develop new implementations based on tree structure algorithms.

For the benchmarks two codes for Computational Fluid Dynamics (CFD) are used. This type of applications extensively use functions to gather and scatter information for various parameters of the simulations, something that makes them ideal for benchmarking collective operations. The benchmarks are executed on the United Kingdom National Supercomputing Service, HECToR. The system is a hybrid supercomputer consisting of a massively parallel scalar system and a vector system. Both systems are going to be tested to observe the behaviour of the algorithms on different architectures.
# Contents

1 Introduction .......................................................... 1

2 Background ............................................................ 3
   2.1 Unified Parallel C ................................................. 3
      2.1.1 UPC Compilers ................................................. 5
      2.1.2 Collectives implementations .............................. 7
   2.2 The system ....................................................... 8
      2.2.1 HECToR Cray XT4 ........................................... 8
      2.2.2 HECToR Cray X2 ........................................... 9
   2.3 Applications .................................................... 10
      2.3.1 BenchC ....................................................... 10
      2.3.2 XFlow ...................................................... 12

3 Porting ................................................................. 14
   3.1 BenchC ............................................................ 16
      3.1.1 Preprocessor flags .......................................... 16
      3.1.2 Timers ....................................................... 16
      3.1.3 Header files issue ......................................... 18
      3.1.4 Broadcasts within all-reductions ......................... 19
   3.2 XFlow ............................................................ 19
      3.2.1 Collective operations file ................................ 19
      3.2.2 Timers ....................................................... 20
      3.2.3 Header files issue ......................................... 22
      3.2.4 Broadcasts within all-reductions ......................... 22
   3.3 Tests ............................................................ 22

4 Optimizations ........................................................ 23
   4.1 Broadcast ........................................................ 23
      4.1.1 Default CFD code implementation ......................... 24
      4.1.2 Implementation 1 ........................................... 25
      4.1.3 Implementation 2 ........................................... 27
      4.1.4 Implementation 3 ........................................... 28
      4.1.5 GASNet implementation .................................... 29
      4.1.6 MTU implementation ....................................... 30
   4.2 Reduction ........................................................ 31
4.2.1 Default CFD code implementation ........................................ 31
4.2.2 Implementation 1 .......................................................... 32
4.2.3 Implementation 2 .......................................................... 33
4.2.4 MTU implementation ...................................................... 34

5 Benchmarks and Analysis ...................................................... 36
  5.1 Input parameters ............................................................ 36
     5.1.1 BenchC ............................................................... 36
     5.1.2 XFlow ............................................................... 39
  5.2 Benchmarks ................................................................. 41
     5.2.1 Cray XT ............................................................... 41
     5.2.2 Cray X2 ............................................................... 51

6 Conclusions and Future Work .............................................. 60
  6.1 Conclusions ................................................................. 60
  6.2 Future work ............................................................... 61

A HECToR modules ............................................................. 64
  A.1 Cray XT4 ............................................................... 64

B Graphs ............................................................................ 66
  B.1 Cray XT ............................................................... 66
     B.1.1 Broadcast ........................................................... 66
     B.1.2 Reduction .......................................................... 70
  B.2 Cray X2 ............................................................... 73
     B.2.1 Broadcast ........................................................... 73
     B.2.2 Reduction .......................................................... 77

C Work Plan ......................................................................... 80
## List of Tables

3.1 Timers used in BenchC depending on programming standard, architecture and compiler ........................................... 17

4.1 Quad tree statistics for first broadcast implementation ................................................................. 26
4.2 Quad tree statistics for second broadcast implementation ............................................................... 27

5.1 Input parameters for BenchC ........................................................................................................... 37
5.2 Percent of time spent in collective operations [BenchC, XT4] ....................................................... 38
5.3 Percent of time spent in collective operations [BenchC, X2] ........................................................... 38
5.4 Input parameters for XFlow ............................................................................................................ 39
5.5 Percent of time spent in collective operations [XFlow, XT4] ......................................................... 40
5.6 Percent of time spent in collective operations [XFlow, X2] ........................................................... 40
## List of Figures

2.1 UPC shared variables example ................................................. 4
2.2 AMD Opteron Quad-core architecture schematic .......................... 9
2.3 BenchC execution diagram ...................................................... 11
4.1 User broadcast implementation ............................................... 25
4.2 First implementation of broadcast .......................................... 26
4.3 Second implementation of broadcast ....................................... 27
4.4 Third implementation of broadcast ......................................... 28
4.5 Binomial tree of order 4 ......................................................... 29
4.6 MTU GET implementation of broadcast .................................... 30
4.7 User implementation of reduction .......................................... 32
4.8 First implementation of reduction .......................................... 33
4.9 Second implementation of reduction ....................................... 34
5.1 Broadcast tree rank benchmarks [BenchC, XT4] ............................ 43
5.2 All broadcast implementations comparison [BenchC, XT4] .............. 45
5.3 All broadcast implementations comparison [XFlow, XT4] .............. 46
5.4 Reduction tree rank benchmarks [BenchC, XT4] ........................... 48
5.5 All reduction implementations comparison [BenchC, XT4] ............. 49
5.6 All reduction implementations comparison [XFlow, XT4] .............. 50
5.7 Broadcast tree rank benchmarks [BenchC, X2] ............................. 53
5.8 All broadcast implementations comparison [BenchC, X2] .............. 54
5.9 All broadcast implementations comparison [XFlow, X2] ............... 55
5.10 Reduction tree rank benchmarks [BenchC, X2] ............................. 57
5.11 All reduction implementations comparison [BenchC, X2] .............. 58
5.12 All reduction implementations comparison [XFlow, X2] ............... 59
B.1 All broadcast implementations comparison (tree rank 4) [BenchC, XT4] 66
B.2 All broadcast implementations comparison (tree rank 8) [BenchC, XT4] 67
B.3 Broadcast tree rank benchmarks [XFlow, XT4] ............................. 68
B.4 All broadcast implementations comparison (tree rank 4) [XFlow, XT4] 69
B.5 All broadcast implementations comparison (tree rank 8) [XFlow, XT4] 69
B.6 All reduction implementations comparison (tree rank 4) [BenchC, XT4] 70
B.7 All reduction implementations comparison (tree rank 16) [BenchC, XT4] 70
B.8 Reduction tree rank benchmarks [XFlow, XT4] ............................. 71
Acknowledgements

First of all, I would like to thank my supervisors, Dr Adam Carter and Jason Beech-Brandt for their continued support and supervision throughout this project.

Also, I want to express my appreciation to my colleague and friend Christopher Charles Evi-Parker for his help during the proofreading stage.

Last but not least, I thank my parents for making this Masters degree possible.
Chapter 1

Introduction

In parallel programming there are many factors that need to be considered in order to be able to develop an application that has good parallel characteristics. One of these factors is the efficient communication between the processes for performing collective operations. Depending on the data dependencies and the algorithms used, different operations are important for every application. For portability and performance reasons these operations are usually part of the parallel programming language specifications. This creates an abstraction layer and gives the opportunity to developers to write their own implementations in cases where the default versions have poor performance. Moreover the codes are able to execute on a variety of systems and architectures without the need of porting procedures.

The Unified Parallel C (UPC) is one of the new parallel programming languages based on the Partitioned Global Address Space (PGAS) model. The main feature of the model is the use of single sided communications for data exchange between processes. This offers great flexibility to the developer and makes it possible to parallelise applications that have dynamic communication patterns.

The United Kingdom National Supercomputing Service, known by the name HECToR, has recently installed a UPC compiler for both architectures of the system. HECToR is a Cray XT5h hybrid architecture system. It consists of a Cray XT4 massively parallel scalar system and a Cray X2 vector system. In addition to the installation of the UPC compiler the XT system was upgraded from dual core processors to quad cores. The multi-core architecture introduced new potential optimisations for collective functions. The purpose of the project is to benchmark the currently available implementations of collective operations and develop new versions in an effort to optimise them.

For the benchmarks two applications for Computation at Fluid Dynamics (CFD) are going to be used. They are both developed by the same software development group, the Army High Performance Research Centre (AHPCRC) of the United States. The first application is a hybrid code written using MPI and UPC called BenchC. The second application, called XFlow, is entirely developed in UPC. The extensive use of collective
operations like the broadcast and the reduction functions makes these codes a good choice for benchmarking tools.

The structure of the rest of the report follows the same order of stages as the actual work was carried out. Chapter 2 begins with more details about the UPC language, the PGAS programming model and the single sided communications. Afterwards follows a brief description of the UPC compilers that are going to be used and their default implementations of collective routines. Next is a description of the two architectures of the system and finally a few details about the two CFD codes.

In Chapter 3 the porting process for the two CFD codes is described. All the changes that had to be made are explained and also the additions to the timers in order to be able to time the collective functions individually.

In Chapter 4 the new implementations for the collective operations and also the currently available versions are described.

Chapter 5 presents all the graphs taken from the benchmarks. For each graph there are comments and explanations about the behaviour of the different implementations.

Finally, Chapter 6 includes conclusions about the project and some ideas about future work.
Chapter 2

Background

2.1 Unified Parallel C

The main topic of the project is focused on a new programming language standard, Unified Parallel C (UPC) [1]. As the name suggests, UPC is an extension of ISO C. The functionality of the language is implemented using the Partitioned Global Address Space (PGAS) model. One of the main characteristics of the model is the use of Single Sided communications for exchanging data between processes. Most of the communications are implied and the programmer does not need to explicitly perform any data exchange. Expressions trigger Remote Memory Access (RMA) calls when a shared variable is read or written.

PGAS model

PGAS is a programming model designed for parallel languages. It considers all available memory as a unified global memory address space, where all threads can read and write data. This convention simplifies the data exchange and makes it easier to program. The affinity of the data is specified on declaration. Accessing data with local affinity benefits from local reference and it does not trigger remote memory calls.

In UPC this functionality is implemented using shared variables. Variables which are not declared as shared are private to the thread and not visible to the others. This distinction between data types also holds for general types like arrays and structures. An example of a shared integer array is:

```
```

The shared keyword must be set at the beginning of the declaration. After that the affinity of the data is set. In the example the block size is set to 3. The elements of the array are distributed in round robin order and in blocks of three elements. Figure 2.1
shows a graphical representation of the example for a thread count of 5 [2].

![UPC shared variables example](image)

The blocking factor is optional and if ignored it defaults to 1. Moreover, empty brackets set the blocking factor to 0. All data are placed in the shared memory address space of thread 0. Scalar variables are also allocated in thread 0.

**Single-sided communications**

In single-sided communications there are two operations for exchanging data. A process can **PUT** data into another process’ memory address space or **GET** data from it. In both cases the target process is not aware of the memory access, thus is not participating in the communication. This freedom of data exchange favours applications with dynamically changing communication patterns. When the code is translated, references on shared variables are translated to remote memory accesses depending on the affinity of the data.

Despite the fact the model offers ease of use to the programmer, it comes with a price. Remote Memory Access (RMA) calls do not imply synchronization. The lack of synchronization between communicating threads creates a demand for explicit synchronization by the programmer. All data needed by the origin process must be placed in the shared variables and the target process must be told that it can start accessing them. Similarly for putting data, the target process must ensure the variables used do not contain any important data because they will be overridden.

UPC has three methods for synchronization.

- Barriers
- Fences
- Locks

Barriers are the most commonly used method. It synchronizes all threads and guarantees that they reached a specific line of code. It is useful for synchronizing after a specific block of code is executed, so that the threads can start reading data from their shared address space. **Split phase** barriers also exist in UPC for overlapping communication with computation if possible.
Fences are used to synchronize memory accesses. All writes issued before the fence are made visible to all threads. After the fence memory reads are guaranteed to see the new values assigned to the shared variables.

Locks are used for securing critical sections and avoiding race conditions. They are used to isolate specific blocks of code and ensure that they are only executed by one thread at a time. These blocks usually manipulate values on shared variables, and having two or more threads executing the same code could lead to data corruption.

### 2.1.1 UPC Compilers

Currently there are many established UPC compilers. Since the language is relatively new, features are added and the functionality extended quite often. During the development of the project two compilers were used. The Berkeley UPC compiler, which is a joint project of Lawrence Berkeley National Laboratory (LBNL) and Berkeley California University, and Cray UPC compiler. The Berkeley compiler was used for the XT part of the system, and Cray for the X2 part.

Although the system where the codes were executed is a Cray supercomputer, the Berkeley compiler was used on the XT part. It is one of the currently most well written UPC compilers and it has features that were useful for optimizing the collective operations. It supports the full UPC Specification Standard (v1.2) [1] and installs with the GASNet [3] network library as the default network layer. This makes the compiler a very good and portable solution for translating and compiling UPC codes. Moreover, the compiler is open source and fully configurable. Particularly for the Cray XT system a cross compiler configuration tool exists, which makes the installation process very easy. It enables all XT specific features and also enables GASNet to use the Portals conduit as the default network protocol.

The Cray UPC compiler on the other hand is not an open source compiler. Is a proprietary compiler developed by Cray for compiling codes for Cray specific systems. For compiling codes on the X2 part of the system the use of the compiler is mandatory. Due to the unique architecture of the system, only the Cray specific compiler exist for compiling codes. This compiler is very well written and offers a variety of options for compiling and optimizing codes [4]. An important feature is its vectorization capabilities. Taking advantage of the vector processors benefits the execution time and scalability of the codes a lot. The implementation of the communications is not publicly available, but the algorithms used for the collective operations are available. The compiler currently uses the MTU collectives implementation [5]. More details about the algorithms are explained in the next paragraphs.
GASNet and Portals conduit

An essential factor in the performance and scalability of the collective operations, is the use of the high performance network support available on each system. On the XT part of the system the network is utilized using a SeaStar2 network interconnect. Portals is a lower level network protocol, developed and designed by Sandia National Laboratories, and currently used on the Cray XT and X2 systems [6]. Primary objective of the protocol is to solve the scalability problems in massively parallel systems. As supercomputers keep growing in processing nodes, the scaling of the network becomes a crucial factor.

As it is mentioned in the Project Preparation report [7], the GASNet [3] network layer is a language independent, high performance library. The primary goal is to provide network support for various networks and operating systems, for global address space languages like UPC, Titanium and Co-Array Fortran. The latest version, which came out on the 3rd of October 2008, supports a variety of network protocols like Cray XT3/XT4 Portals, Cray X1/SGI Altix SHMEM, IBM BlueGene/P, Infiniband and many more. It also supports a large range of operating systems, architectures and compilers. Along with various bug fixes and added platform support the latest version has a new improved performance implementation of the Portals network interface.

Berkeley UPC

The Berkeley compiler was chosen for two basic reasons: (a) because it is one of the most well implemented UPC compilers and (b) because of the collective operations implementation.

From 2003 Berkeley UPC has released updated versions of both the translator and the runtime. As the language was changing, all the new features and new specifications were incorporated in new versions. Using a bug server, the software development team is trying to improve the quality of the final translated code and fix the bugs found by the users.

Starting from release 2.5.10 the compiler supports the pthreads option for Cray systems. This option enables the compiler to use POSIX threads to spawn UPC threads rather than creating new processes. Threads created by the same process can communicate through the main memory without the need of RMA calls. This significantly reduces the use of the network for shared variable accesses. Data with affinity to any of the threads can be accessed by the rest as if it was local.

For collective operations in particular this can be very beneficial. Information within the same cluster can be broadcast or reduced with less communication cost. Only one of the threads needs to perform an RMA call and then scatter or gather the information within the cluster using shared global variables.
2.1.2 Collectives implementations

One of the main aspects of a parallel language standard is the collective operations. The collectives specifications [8] were officially introduced in 2003 and integrated in the language specification after two years. Prior to the specification, users were implementing their own collectives using generic code. A lot of simple, functional implementations are available, although they are not optimized as far as scaling is concerned. Usual approaches consist of explicit point-to-point remote memory accesses with only a single master process performing the work. By doing so, the entire work load is executed in serial order and thus not taking advantage of any parallel features.

Since their official incorporation in the language specification, two major implementations of the collectives were developed. The first implementation is by Berkeley UPC Group (BUPC) and the second by the Michigan Technological University (MTU). Their distinctive difference is in the communication pattern.

The MTU code implements a flat algorithmic approach in which the communications are performed by either only PUTs or GETs. Although the work load is divided in most of the collective functions, the scaling factor is not optimal. In some cases, for example the broadcast, implementation using PUT operations will serialize the procedure. Thus the GET version makes the procedure completely parallel, performing a vast amount of operations at the same target process causes bottlenecks and poor performance. The remote memory accesses are implemented using native memory copy functions. The choice between the two implementations is made using compiler directives during compilation. All collective calls are implemented with the same approach. Altering the implementation of specific collectives is made possible by altering the appropriate flags in the source files.

The BUPC implements a binomial tree with variants of PUTs and GETs. Tree structures are the traditional way of implementing collective operations. The data are propagated very fast down the tree and parallelism can be exploited in many ways. Various implementations exist and most of them have minimum needs for synchronization. The implementation of the binomial tree is developed in low level using the GASNet library function calls [3].

Most of the hardware of today’s supercomputers are commodity components. Something that tends to remain system specific is the interconnect. Each vendor has its own optimized interconnect and utilizes its own high performance protocol. The GASNet low-level network layer creates an abstraction layer between the programmer and the underline network. With the use of GASNet the code can be easily reused on any system just by changing the GASNet network protocol option.
2.2 The system

For the purposes of the project the United Kingdom National Supercomputing Service was used. The system is a hybrid architecture machine, which includes scalar and vector processor nodes.

The basic building block of the system, a compute blade, hosts four quad-core AMD processors. HECToR has currently 1416 compute blades, which totals to 22,656 active cores. Each quad-core processor has 8 GB of main memory, giving a total of 45.3 TB of distributed memory. Additionally, it has 24 service blades, each hosting 2 dual-core processors. Moreover, it has 934 TB of high performance RAID disks. The disks are controlled using 3 controllers through 12 I/O nodes [9]. Every compute node can access the storage through the Lustre distributed parallel file system.

2.2.1 HECToR Cray XT4

The XT part of the system is a Cray XT4 massively parallel processing (MPP) system [10]. The main processing unit is commodity AMD Opteron processor at 2.3 GHz clock speed. Each processor is a quad core architecture with three levels of cache. The first level (L1) has 64 KB data cache and 64 KB instruction cache. The second level (L2) consists of 512 KB. Each core has dedicated caches for the first and second levels. Both levels can store eight double precision words at each line, which sums to 64 byte cache lines. The third cache level (L3) has 2 MB and it is shared between all four cores. Level 2 and 3 caches behave like victim caches for their upper levels. Data dismissed from L1 cache are stored in L2 cache and similarly data dismissed from L2 are stored in L3. Figure 2.2 shows the inner processor architecture.

For main memory the system has 8 GB of Double Data Rate (DDR2) memory modules. Main memory is available to the cores through the HyperTransport technology. The memory has two modes of operation. Single node (SN) and virtual node (VN). In single node all memory is dedicated to a single compute task. In virtual node the memory is split equally to all processing cores, giving 2 GB on each core.

The communication between the processors is implemented using a proprietary Application Specific Integrated Circuit (ASIC) chip, the Cray SeaStar2. Every chip is dedicated to one quad-core processor. SeaStar has 6 links to neighbouring processors and implements a 3D-torus topology with dimensions 20×12×24. It has point-to-point bandwidth of 2.17 GB/s and minimum bi-section bandwidth of 4.1 TB/s. The latency between nodes is estimated to be around 6µs.

For the lower level communication the SeaStar2 uses the Portals low-level message passing interface. Part of the protocol functionality is implemented inside the chip’s firmware, thus offloading the communication load from the processor. Also, it enables the interface to directly transfer data to and from the user’s memory using the Hyper-
Transport link, without the operating system taking part in the procedure.

### 2.2.2 HECToR Cray X2

The second part of the system is implemented with Cray X2 vector processors. It consists of 28 vector compute nodes. Each node has 4 vector processing units, implemented as a 4-way SMP. This totals to 112 active vector processors. There are 8 vector pipes per node and each pipe can perform addition and multiplication of floating point numbers. Each pipe is coupled with 16 elements of a vector register, giving a vector length of 128 elements. The theoretical performance of each core is 25.6 Gflops, which totals a peak performance of 2.87 Tflops.

The processors have a clock rate of 1.6 GHz, on-chip Level 1 and Level 2 caches and 4 bi-directional Network Interface (NIF) ports. The cache sizes are 16 KB for Level 1, 512 KB for Level 2 and for Level 3 it has 8 MB of shared memory between the 4 processors on the compute node. The local main memory is divided into 16 Memory Daughter Cards (MDC). The memory cards are controlled by Weaver memory controllers [11]. The memory available for every compute node (4 processors) is 30 GB [9] and each processing unit has uniform memory access to all local memory through a high-speed narrow channel to every daughter card.

Communications between the compute nodes are implemented using a proprietary interconnect, the YARC chip. The topology used in this part of the system is a fat-tree,
which allows the system to scale well to many thousands of processors. The YARC routing chip hosts 64 ports. Each port can support bidirectional traffic with a bandwidth of 1.9 GB/s in each direction. The point-to-point bandwidth of the system is 16 GB/s and the bi-section bandwidth 254 GB/s.

Cray X2 and XT systems are tightly connected using a high-speed network bridge called StarGate. The StarGate chip is a protocol bridge chip that provides the conversion between the Cray XT SeaStar 3D network and Cray X2 YARC fat-tree network. User login nodes are housed in the Cray XT cabinets and can launch applications on the XT compute blades or the X2 compute blades. All X2 input/output packets are directed to a StarGate. The StarGate chip provides the necessary protocol conversion between the YARC network protocol and the SeaStar network protocol. The StarGate chip enables I/O and network traffic to seamlessly travel between the two systems.

2.3 Applications

2.3.1 BenchC

The first of the two applications used for benchmarking the collectives is BenchC. It is a Computation at Fluid Dynamics (CFD) kernel developed by the Army High Performance Research Centre (AHPCRC) in the United States [12].

Since 1991 AHPCRC has been developing parallel flow solvers, with the latest one being BenchC. This application solves unstructured meshes applications with the use of Finite Element Methods. The first version of the code was implemented using C*, an extension of C programming language designed for data parallel systems. The system was a distributed memory, MIMD architecture vector system, the Thinking Machines CM-5. Over the years, and as the parallel programming standards evolved, the code was reimplemented using the Message Passing Interface (MPI). This new programming model gave the application the ability to run on large supercomputers taking advantage of the massively parallel architectures.

After the UPC language was developed the code was modified in order to incorporate some of the features provided by this PGAS model. The new code incorporates an option for choosing between a version entirely written with MPI, or a hybrid version with MPI and UPC. The main core of the application uses MPI calls for all communications. Only four functions were converted. The gather, scatter, broadcast and reduction functions are implemented in both versions. The choice between implementations is possible with the use of compiler preprocessor directives.

BenchC calculates the velocity and pressure variables using the incompressible Navier-Stokes equations. Due to the non-linear nature of the Navier-Stokes equations, an equation system is formed and solved using a GMRES-based iterative equation solver.
There are four iteration levels in the code. The first, outermost level, is the time step loop. The second level is the non-linear iteration loop. At each iteration a new equation system is formed and solved. The other two levels are inside the GMRES solver, and are referred to as inner and outer loops. The inner iterations specify the Krylov space size. The default value is set to 20, which is in general a good value for most datasets. The outer iteration, also called restart iteration, enables the option of executing the GMRES solver successive times. The default value is set to 1 and almost never set to a different value.

Throughout the execution, timers and performance monitoring variables store information about all aspects of the application. As the execution progresses information about the current time step is printed out to let the user know which step of the algorithm is currently calculated and what the current values of different variables are. The scatter and gather functions take up roughly two thirds of the communication time, while broadcast and reductions spend the remaining third of communication time.

The execution pattern of the application can be split into two stages. The preprocessing stage and the iterative stage (time marching algorithm). Figure 2.3 shows a diagram of the execution.

Figure 2.3: BenchC execution diagram
During the preprocessing stage, the mesh is read from the file and initially scattered to the processes. After that, a Recursive Coordinate Bisection (RCB) partitioning algorithm splits the mesh amongst threads. Now the mesh is properly distributed, a redistribution step occurs to place the tetrahedral on the correct process. The next two steps perform the communication setup between neighbouring processes and set the boundary conditions. Moreover, during the preprocessing stage all the dynamic memory allocation takes place.

The time marching step consists of the four iteration levels mentioned. Depending on the values given in the input file, variable number of time steps and non linear iterations are performed. Apart from the iteration options, the input file can define a number of different constant values. More details on the input files, values and options available are mentioned in Section 5.1.1. The code is well structured and had been successfully tested with extremely large problems containing over 2 billion tetrahedral elements.

2.3.2 XFlow

The second application used for benchmarking the collectives is XFlow. This is similar to the first application mentioned and is another CFD kernel developed by the the Army High Performance Research Centre (AHPCRC).

These two applications share a lot of features, but also have some very distinctive differences. The most important difference is the mesh structure. BenchC uses a static unstructured mesh while XFlow uses a dynamic unstructured mesh [13]. This enables XFlow to simulate more complex CFD applications. Domains where the shape is changing or has moving objects, like flapping wings, engines or pumps, the dynamic nature of the application can adapt to these changes and refine the mesh in order to achieve better simulation. The common approach in mesh construction is to create a good first mesh, and keep it static throughout the simulation. The outcome is to introduce a dependency between the mesh and the final result. Altering the primary mesh can change the quality and accuracy of the whole simulation. Using the dynamic method, the mesh is changing and adapting to the domain as the simulation progresses. In contrast to the previous dependency, the mesh depends on the desired solution quality and accuracy. The mesh can be refined at each time step or at a multiple of time steps. The current application has the option to define the refinement policy as an input parameter at the start of the execution. More details about the input parameters are discussed in Section 5.1.2.

The automatic mesh generation (AMG) technology in XFlow is coupled with a traditional CFD simulation code developed by the AHPCRC [14]. The AMG technique implemented is a Delaunay based method to generate and manipulate the unstructured mesh.
Taking into account the dynamic character of the application, the development of the code had a very high complexity factor. One of the difficult parts was the communication pattern between neighbouring processes. As the mesh changes, the elements on the process boundaries change, creating a need to adjust the communications. Along with this issue comes the dynamic implementation of the lists storing the nodes, faces and elements at each process. Refining can mean both removing existing nodes or creating new ones.

Furthermore, the actual refinement algorithm performs various steps in order to identify the area of refinement and split or merge depending on the refinement variable. All these steps create the need for referencing elements, faces or nodes arbitrarily across the entire domain. For keeping the complexity of the refinement manageable, XFlow allows only one process to change the mesh at each step.

In consideration of the previous issues the parallel programming standard used for this kind of algorithm had to fulfill some certain characteristics. The use of Message Passing Interface (MPI) was unsuccessful due to complexity. The process boundaries are very restrictive and processes need to communicate explicitly to obtain information about elements, faces or nodes. Since the domain is dynamic and the information at each process always changing, the implementation of the steps mentioned above are very difficult to implement.

Having finished with the literature review on the parallel language standard, the systems and the tools used for the project, the next stage is to port the CFD codes for the benchmarks and ensure their correct functionality. The next chapter will explain the steps performed in order to port the codes and the necessary changes made for obtaining the timing information needed.
Chapter 3

Porting

The primary target was to port the codes for the systems that were going to be used and ensure their correct functionality. The use of two different compilers introduced extra difficulty to the porting procedure of the timers and extensive testing was required to ensure that the timings taken were correct.

Cray compilers were preinstalled for both the XT and X2 systems. The Berkeley UPC compiler had to be configured and installed on the XT system. The installation was straightforward due to the installation script file supplied by the Berkeley UPC compiler, for enabling all features supported by the system. As mentioned in Section 2.1.1 the script also enables support for the Portals conduit, which is the high performance protocol used on the XT system.

The Portland Group compiler is part of the default programming environment loaded on login and was used for compiling the Berkeley UPC compiler. It was also used for the MPI version of BenchC. The complete list of default modules loaded on login can be found in Appendix A.

The Berkeley UPC compiler version information is as follows:

<table>
<thead>
<tr>
<th>UPC Runtime</th>
<th>v. 2.8.0, built on Jul 1 2009</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPC-to-C translator</td>
<td>v. 2.8.0, built on Nov 5 2008</td>
</tr>
<tr>
<td></td>
<td>host aphid linux-x86_64/64</td>
</tr>
<tr>
<td></td>
<td>gcc v4.2.4 (Ubuntu 4.2.4-1ubuntu3)</td>
</tr>
<tr>
<td>Translator location</td>
<td><a href="http://upc-translator.lbl.gov/upcc-2.8.0.cgi">http://upc-translator.lbl.gov/upcc-2.8.0.cgi</a></td>
</tr>
<tr>
<td>Default network</td>
<td>portals</td>
</tr>
</tbody>
</table>

The module package is responsible for loading the various modules available on the Cray systems. It can be also used to swap programming environments and load the X2 default modules. The list with the default modules for the X2 system can be found in Appendix A.
Compilation flags

For the XT system the Berkeley compiler supports a number of different compilation options. As far as the UPC optimizations are concerned the compiler has the following options:

1. **split-phase**: Enables split-phase communications to overlap communication with computation
2. **pre-add**: Eliminate redundant pointer arithmetic in shared pointers. These operations are expensive and their elimination will benefit the execution time
3. **ptr-coalesce**: Performs network transfers to contiguous memory addresses in one operation rather than many
4. **ptr-locality**: Intra-procedural analysis to replace use of shared pointers with normal C pointers to achieve faster pointer arithmetic and data access
5. **forall-opt**: Optimizes the `forall` statement by generating serial code with fewer affinity checks
6. **msg-vect**: Loop optimizations. Still in an experimental stage

The `-opt` compiler flag enables the default optimizations which includes split-phase, pre-add and ptr-coalesce. The compiler’s manual clearly states that all optimizations are experimental. Test executions were performed to observe the effect of the default optimizations and ensure the correctness of the code. The primary results taken were not clear and more tests were needed to clarify the effect of the optimizations. In some cases the code failed to execute and in other cases the codes were taking longer to finish. Due to limitations in both time and resources it was decided not to use the optimizations for the benchmarks.

For the X2 system the compiler offers a different set of compilation options. The Cray vector compiler is a well written, system specific compiler with many optimization options. For compiling both CFD codes the options below were used [4]:

1. **-h fp3**: Floating point optimizations. Level 3 is the highest level but the least conformable with the IEEE standard
2. **-O3**: Enables Level 3 for scalar, vector, stream and cache optimizations
3. **-h ipa4**: Automatic inlining of small functions
4. **-h aggress**: Aggressive optimizations for loops

These optimizations may have an impact on the performance of the collective operations on the vector system. Great benefit can be gained from vectorizing blocks of code because it implies parallelism.
3.1 BenchC

3.1.1 Preprocessor flags

As mentioned in Section 2.3.1 the code uses MPI for the main part of the code, but supports an option for choosing between MPI and UPC for the collective operations. This option is implemented using a preprocessor flag. The collective operations BenchC uses are broadcast, reduction (only global sum), scatter and gather. The flag also modifies parts of the final output to print UPC related timings rather than MPI.

Another two very useful preprocessor flags are __craynv and __crayx2. Both flags are defined by the Cray specific compiler and are usually used to enable code that optimizes and performs better on Cray supercomputers. The second flag is only available on the X2 system and in many cases is used to enable code that the compiler can vectorize.

In more detail, the functions altered by the Cray compiler specific flags are:

- All collective operations
- The code implements a Quicksort algorithm. The use of restrictive pointers enables the compiler to perform optimizations because it guarantees that the pointers passed as arguments have restricted access to the actual data
- A Binary search algorithm is also implemented and the use of restrictive pointers enables optimizations

A combination of the above flags was used in order to choose the correct timers for each architecture. Due to the fact the code uses the Cray specific timer routines, the use of a different compiler, Berkeley UPC, created the need for different timers as well. More about the timers will be explained in the next section.

3.1.2 Timers

The code implements its own logic about timing the various sections and monitoring the performance. A global clock starts ticking at the beginning of the code. When an event that needs to be timed independently starts, the tstamp function is called. The parameter of the function defines the timer variable at which the previous time slice will be added. Time slice is defined as the time passed between successive calls to the function. Summing the various timer variables gives the total run time. The use of a global clock and time slices guarantees that the overall time is counted and added to various timers. This makes performance monitoring more efficient and accurate.
The next step was to investigate the function calls the code uses to get the real time clock. There are two different functions, depending on the compiler used. When compiling with a Cray compiler, the intrinsics `_rtc` function is called. For all other compilers the `MPI_Wtime` function is called. These two functions return a different metric of time. Cray’s `_rtc` returns clock ticks as a `long long` integer value. On the other hand, `MPI_Wtime` returns time in seconds as a `double` precision value.

For choosing between timer functions the code uses the **UPC** preprocessor flag. When the code is compiled using the MPI standard, the timer provided by the MPI library is used. When compiled using UPC, the functions differ between architectures. Table 3.1 shows the functions depending on different compilation options. On the X2 part of the system, the Cray compiler intrinsics are available and thus using `_rtc` is not a problem. On the XT part, because of the Berkeley compiler, the intrinsics are not available. To overcome this problem, an `_rtc` function was implemented using the `gettimeofday` timer structures. By using the `__craynv` flag, it can be identified when the Cray compiler is not used. To overcome the problem the new `_rtc` implementation is used and the code is able to compile successfully. The appropriate changes to the timer variables were also made, because the new function returns time in seconds, in contrast to the intrinsics that returns clock ticks.

<table>
<thead>
<tr>
<th></th>
<th>MPI</th>
<th>UPC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PGI on XT</td>
<td>Cray on X2</td>
</tr>
<tr>
<td><code>MPI_Wtime()</code></td>
<td><code>MPI_Wtime()</code></td>
<td><code>_rtc()</code></td>
</tr>
<tr>
<td><code>double</code></td>
<td><code>double</code></td>
<td><code>double</code></td>
</tr>
</tbody>
</table>

Table 3.1: Timers used in BenchC depending on programming standard, architecture and compiler

The timings for the various sections of the code are stored in an array of timer variables. Each position of the array defines a different timer. Below are the 12 sections timed:

1. Input - output
2. Setup of the application (initialize timers, allocate memory)
3. Partitioning the domain
4. Other partitioning
5. Scatter
6. Gather
7. Reduction/Broadcast
8. Matrix transformation functions (diagonalization, RHS formation)
9. GMRES solver
10. Matrix-vector multiplication
11. Mesh moving
12. Other
As it can be seen the reduction and broadcast operations are monitored using the same timer. This makes it impossible to separate the time spend by each operation at the end of the simulation. To be able to monitor each operation independently, different timers are used for each operation. The array structure of the timers makes it easy to expand and add more timers. An additional timer was added and used for the reduction operations. Changes to the code were also made in order to call the \texttt{timestamp} function with the correct timer identifier. The output was also altered to print out the new information.

For enhancing the information gathered by the timers, variables were declared to store information about the call count of the collective operations. Every time the function gets called, the counter is incremented and it is printed out in the end together with the timing information.

### 3.1.3 Header files issue

The procedure for compiling UPC codes includes a stage where the code is translated from UPC to C, and then compiled with a normal C language compiler. The Berkeley UPC-to-C translator handles header files with a special way. There are two supported types. The regular files having the "\texttt{.h}" extension, and the UPC header files having the "\texttt{.uph}". The difference is in the code allowed to exist in each file. The first type is not allowed to have any UPC specific constructs. The translator is ignoring the regular "\texttt{.h}" C header files during the translation step. Files with no UPC code do not need to be translated. In a later step where the code is compiled these header files will be taken into consideration.

This behaviour creates issues with header files that have dependencies in source files. Preprocessor macros defined in source files are only expanded the first time, when the code is translated. During the compilation step the macros are not expanded again thus cause compilation problems.

In BenchC the above issue is present. Source files include header files with external variables. Depending on the source file these variables must be either instantiated or imported as \texttt{extern} variables. Failure to do so declares all variables in all files as \texttt{extern} and compilation fails with errors with \texttt{undefined variables}.

There are two possible workarounds to this problem. The first is to explicitly define the macros when executing the compilation command. This way they are going to be present when the code is translated and compiled. This can produce long compilation lines and it is not easy to maintain. The macros have to be hard coded inside the \texttt{Makefile}. The second solution is to rename the "\texttt{.h}" files to "\texttt{.uph}". The files will then not be ignored during the translation and the dependencies are going to be taken into consideration.
Two files had to be renamed for the problem to be resolved. These are the "sf340.h" and "info.h" header files. Both files contain code which is not UPC, but their dependencies with the source files which includes them are important for both translation and compilation stage.

### 3.1.4 Broadcasts within all-reductions

The default implementation of all-reduce operation has the final broadcast step hard coded inside its code. Part of the optimization procedure was to optimize both steps of the operation. Since the broadcast was already optimized it could be used to broadcast the final reduced value to all threads. Instead of copying the broadcast code at the end of the reduction, like it is in the default version, the broadcast is going to be executed as a function call to the normal broadcast operation.

An issue with the timings arises from this approach. When the broadcast function gets called is timing how much time the call needed, and is adding this value to the broadcast timer at the end. A mechanism was implemented in order to be able to suspend the timings when the broadcast is called from within a reduction. The reduction is timing the time spend throughout all steps of the operation and adding this time to the reduction timer. The time spent in the broadcast is part of the reduction not the individual broadcast operation.

For implementing this idea the `timeit` variable was used. This variable is used by the application to enable or disable all timings. Normally it is set at the begining of the execution, and the code never alters its value. For executing a broadcast from within a reduction, this variable is set to 0 in order to suspend the timers, and restored to 1 after the broadcast.

### 3.2 XFlow

The porting procedure for XFlow had similar steps as BenchC. Both codes were developed by the same center, with XFlow been the newest, and also solve problems which involve flows in fluids (CFD codes). The structure of the codes is almost the same and is based in similar programming ideas.

#### 3.2.1 Collective operations file

The collective operations used by this application are implemented in the same source file. These collectives are:
The user implementations are flat GET or PUT versions. The entire work load is executed by the master thread, while the rest of the threads wait until it finishes. These implementations are straightforward and do not expand into many lines of code. Having them in the same file has no negative effect on maintenance. The new versions are much bigger in size. For better readability and maintenance the code was split into two files. The names used for the new files are the same as in BenchC. All broadcasts are implemented in `my_broadcast.c` and all reductions in `my_reduction.c`. The necessary changes to the Makefile were made as well.

### 3.2.2 Timers

The timers logic is the same as BenchC (Section 3.1.2). The code measures times slices between successive calls of the `tstamp` function and adds them to the specified timers. Due to the dynamic nature of this code there are more timers declared to monitor the behaviour of the application. These timers are:

1. Initialize timings
2. Refinement (Laplace solver)
3. CG solver
4. Mesh movement block (Linear Elasticity)
5. Mesh movement GMRES solver
6. Fluid block-R
7. Fluid block-V
8. Fluid GMRES solver
9. Purge mesh
10. Communication setup
11. Mesh output
12. Regularize 1 (Delaunay quality)
13. Regularize 2 (Aspect ratio quality)
14. Refine mesh
15. Coarsen mesh
16. Repartitioning
17. All other times
Despite the fact the timers are a lot in number, they are not specific enough to be used for the needs of the project. Collective operations must have individual timers in order to monitor their performance without interference from other parts of the code. To solve this issue new timers were declared to store execution times of all collective operations implemented.

For implementing the timers a new struct was declared. It contains a variable for storing the time consumed by the collective operation and also a call counter. The use of two different timer variable data types are going to be explained in the next paragraphs. The call counter can be used to observe how many times the operation is called and adjust the time steps needed for a good estimate. Moreover, it can be used to calculate an approximation of how much time is spend per operation call. Using the \texttt{tstamp} function with the new timers was not possible. Because of the time slices concept every time the function is called the previous slice has to be added to a timer. Calling the function from inside the collective operations suggests changes to the function prototypes of all collectives. A separated timer was used for the new timings which offers flexibility and makes the implementation easier.

\begin{verbatim}
typedef struct {
      unsigned int call_count;
      #ifdef __crayx2
      long long time;
      #else
      bupc_tick_t time;
      #endif
    } extended_timers_s;
\end{verbatim}

Every time a collective operation is called two variables are declared, \texttt{time\_new} and \texttt{time\_old}. These variables store the time when the operation begins and when it finishes. The result from subtracting \texttt{time\_new} from \texttt{time\_old} gives the time consumed. A call to function \texttt{extended\_tstamp} will add this time to the appropriate timer. The timers are stored in an array of structs, and the position of the timer denotes the collective operation monitored. The array can easily be expanded to as many timers needed by increasing the value of a variable.

For similar reason as in Section 3.1.2, the struct is declared with two different data types for the timer variable. For the XT system the code uses the Berkeley compiler’s timer function that counts time in clock ticks, and returns a value of type \texttt{bupc\_tick\_t}. For the X2 system the code uses the intrinsics \_\texttt{rtc} function that again counts time in clock ticks but returns the value as a \texttt{long long} value.
3.2.3 Header files issue

The issue explained in Section 3.1.3 also exists in XFlow. The way the application declares its global variables is similar to BenchC. All source files sharing variables include the same header file. Only one file, `main.c`, instantiates the variables and the rest declare them `extern`. The files that had to be renamed for resolving the problem are: `infoFL.h`, `infoRG.h` and `motions.h`.

3.2.4 Broadcasts within all-reductions

The same idea as in Section 3.1.4 is implemented for XFlow as well. The difference in this application is that there is no variable declared for suspending timings. The timings are always enabled by default. A global variable was declared, `active_timer`, and used the same way as in BenchC. When the value is set to 0 it suspends the timers and when is set to 1 it enables them. This variable is only used for the collective operations timings.

3.3 Tests

Both CFD codes were tested with different process counts to ensure that they are working correct, and also that they return the timing information needed for the project. Through these tests information about the approximate time needed for each execution was gathered and also the call count information for the different collective operations. Depending on this information and the available resources, the value of the time step variable for each code was decided. More details about the input parameters of the codes are stated in Sections 5.1.1 and 5.1.2.

With the CFD codes working, the next stage of the project was to investigate in depth the current implementations, and develop the new implementations of the collective operations. The next chapter describes all the details about the implementations benchmarked.
Chapter 4

Optimizations

The collective operations used by the applications vary in number and their functionality is application specific. Both codes were developed prior to the collective operations specifications manual [8] and they implemented their operations depending on the needs of the algorithm. For the purposes of the project only the most commonly used were chosen to be implemented in order to be tested thoroughly and with different parameter values.

More specific, six different implementations were tested for the broadcast operation and four different implementations for the reductions. Three of the six implementations are new versions of the broadcast developed for the purposes of the project. The other three versions are the currently available implementations from GASNet, MTU and the default version written by the developers of the CFD codes. For the reductions two new versions were developed. The other versions are from the MTU implementation and the default version from the developers of the CFD codes. In the next paragraphs the different versions are explained.

4.1 Broadcast

For the broadcast operation a tree based approach was chosen as the main structure of the algorithms. Each node in the tree represents a single process. In broadcasts the value exists only in one thread at the beginning of the operation. The root node of the tree represents the primary source of the broadcast. A good algorithm will try to propagate this value as fast as possible to more levels so that more sources of data will be created. The propagation of the data from a level to the next will be referred to as a step. For synchronizing the process at each step only one level of the tree is active and able to perform remote memory calls. In tree structures deeper levels have more nodes, thus more threads can execute send and receive operations in parallel as the active level depth moves deeper.
Before being able to perform the algorithms each thread must calculate the values of 3 parameters. These parameters are:

- **Thread depth**: each thread calculates its own depth according to the tree rank and other parameters depending on the implementation
- **Tree depth**: all threads must know how deep the whole tree is
- **Barriers**: how many barriers are needed for the whole operation to finish, depends on the tree depth and other parameter of the implementation

These values do not change during the application execution and do not need to be calculated for every broadcast call. They are depended only to the thread count, and since all threads take part to collective operations the values remain the same. They are stored in global static variables and used for all calls. A flag is used to determine the first call and initialize the parameter variables. The rest of the calls will not execute the initialization process. They will continue to the main broadcast algorithm and use the precomputed values.

Implementations that perform the broadcast or reduction operations with algorithms that do not change the communication pattern depending on the process count will be referred to as *flat* implementations.

**Broadcasts data types**

In BenchC the broadcast is always called with the same data type, a scalar double precision value. The source of the value is always thread 0.

In XFlow the broadcast function is called with three different data types. A scalar integer, a scalar double and two integers. Each broadcast has its own function call but they all implement the same algorithm. This convention was kept for the new implementations as well. The same code was adjusted for all different data types.

### 4.1.1 Default CFD code implementation

The developers of the CFD code implemented their own broadcast operation as a flat **PUT** algorithm. Thread 0 is executing all the work load and copies the value into a shared variable in every thread. The algorithm is quite simple but has poor performance. While the process count is growing the work load is increasing as well. Because the load is placed only in one thread, leaving the rest in idle waiting, it scales very poorly.
In XFlow the approach is similar. The difference is in the primary source thread. Any thread can be the source of the data depending on the demands of the code. The work load in this application is placed on the thread having the value to be broadcast. The algorithm is again a flat PUT algorithm.

Figure 4.1 shows the algorithm in a tree structure of order 4 (quad-tree). Each node in a quad-tree has 4 child nodes and 1 parent. Exceptions are the root node that has no parent and leaf nodes that have no child nodes. The red arrows in the figure represent PUT operations. The arrow starts from the source process of the remote call and ends at the target. All active levels are highlighted using a light blue box. The same conventions are used for all figures in this chapter.

The remote memory accesses needed are equal to the thread count minus one. There are explicit barriers at the beginning and at the end of the broadcast for synchronization. The idea is implemented with a single for loop statement which copies the local value of the source thread in all elements of the shared array. After the last barrier the threads are safe to copy the value from the shared array to their local variable. Copying the value from the shared array is not considered to be a remote memory call as long as the thread accesses an element of the array with local affinity.

For the X2 architecture the code can be fully vectorized. The assignments in the for loop can be performed with a single vector operation because a scalar value is copied to all elements of a shared array. There are no dependencies between the iterations of the loop, they are all done in a serial order and there are no conditional statements. Because of the simple structure of the code the compiler can perform optimizations and improve the final code with vector operations.

4.1.2 Implementation 1

The first approach uses variants of PUTs and GETs. The active level performs a GET operation to copy the value from its parent node. After that it performs a number of PUT operations to copy the value into its child nodes. The number of PUT operations is defined by the rank of the tree. Figure 4.2 shows the algorithm in a quad-tree structure.
The blue arrows represent GET operations. The arrow starts from the source process and ends at the target.

![Diagram](image)

Figure 4.2: First implementation of broadcast

Main reason for using variants of remote calls is to reduce the barriers needed for completing the operation. Only odd tree levels are active in this implementation. Threads on the even levels remain idle and execute a number of barriers until the operation is complete. The trade-off in this approach is between the number of barriers and the active threads. Moreover, a small serial part of code exists when a thread is copying the value to its child nodes. As long as the rank of the tree is sufficiently small, this will not become a hot spot.

Table 4.1 shows the parameter values when a quad-tree structure is used. The tree root node is considered to be on depth 0, which means a tree with depth 3 has 4 levels, including depth 0. The minimum number of barriers is always 2 in all implementations. Threads always synchronize on entry and exit points of the operation.

<table>
<thead>
<tr>
<th>Processes</th>
<th>Depth of tree</th>
<th>Barriers</th>
<th>PUTs</th>
<th>GETs</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>3</td>
<td>3</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>64</td>
<td>3</td>
<td>3</td>
<td>16</td>
<td>47</td>
</tr>
<tr>
<td>128</td>
<td>4</td>
<td>3</td>
<td>59</td>
<td>68</td>
</tr>
<tr>
<td>256</td>
<td>4</td>
<td>3</td>
<td>187</td>
<td>68</td>
</tr>
<tr>
<td>512</td>
<td>5</td>
<td>4</td>
<td>272</td>
<td>239</td>
</tr>
<tr>
<td>1024</td>
<td>5</td>
<td>4</td>
<td>272</td>
<td>751</td>
</tr>
</tbody>
</table>

Table 4.1: Quad tree statistics for first broadcast implementation

For this algorithm the Cray compiler can only vectorize a small part. The PUT operations to copy the value to the child nodes are implemented as a for loop copying the value to contiguous elements of the shared array. These copies can be executed as vector operations. The rest of the remote calls are executed by single operations and thus cannot be optimized by the compiler.
4.1.3 Implementation 2

The second approach implements a method where all threads have a part of the workload to perform. Rather than having the parent threads copy the value to their child nodes, the child nodes are going to copy it themselves.

![Figure 4.3: Second implementation of broadcast](image)

In this implementation the barriers needed are almost double in number comparing to the previous one. This was necessary in order to make all levels active. Before each level executes its work it must first ensure that the previous level is finished. There is a data dependency between successive depth levels and synchronization is needed.

Table 4.2 shows the parameter values for a quad-tree structure.

<table>
<thead>
<tr>
<th>Processes</th>
<th>Depth of tree</th>
<th>Barriers</th>
<th>GETs</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>3</td>
<td>4</td>
<td>31</td>
</tr>
<tr>
<td>64</td>
<td>3</td>
<td>4</td>
<td>63</td>
</tr>
<tr>
<td>128</td>
<td>4</td>
<td>5</td>
<td>127</td>
</tr>
<tr>
<td>256</td>
<td>4</td>
<td>5</td>
<td>255</td>
</tr>
<tr>
<td>512</td>
<td>5</td>
<td>6</td>
<td>511</td>
</tr>
<tr>
<td>1024</td>
<td>5</td>
<td>6</td>
<td>1023</td>
</tr>
</tbody>
</table>

Table 4.2: Quad tree statistics for second broadcast implementation

The vectorization optimizations are not applicable to this implementation. Each thread is executing a single GET operation to copy the value from its parent node. There is nothing that can be vectorized.
### 4.1.4 Implementation 3

The last implementation takes into account the multi-core and vector architecture of the systems used. Goal is to group together threads which can communicate efficiently and optimally not even use the network.

For the XT system the architecture is currently using quad core AMD processors. The batch submission system [15] supports the option of choosing how many UPC threads (processes) to be placed at each processor. With one thread per core the system can be seen as an SMP cluster. Each quad core is considered an SMP node with 4 processing units. These units share common main memory and can perform data exchange operations locally.

The X2 vector system is using compute nodes with 4 processing units implemented as a 4-way SMP (see Section 2.2.2). These units share common main memory and also can perform vectorized loads and stores. Taking advantage of the compiler the code can be vectorized and perform the broadcast operations in parallel for more than one UPC thread.

Figure 4.4 shows the algorithm in a quad-tree structure.

![Figure 4.4: Third implementation of broadcast](image)

The group factor is declared as a preprocessor defined value and can be set at compile time. The idea of propagating the value between levels is the same as in the second implementation. Each group has a thread which is considered to be the group master. This thread will perform a \texttt{GET RMA} to copy the value from its parent node. Immediately after then the synchronization will take place. The copying of the value into the rest of the threads in the group will happen during the next time slot. By doing so the copying will overlap with the propagation of the value to the next level. Exception to this rule
is when the value reaches the final level. The value is then copied to the group threads before the last barrier is executed.

Although the idea is the same with the second implementation the barriers needed are less. Having groups of four rather than single process groups is reducing the depth of the tree. More specific the nodes of the tree are now four times less than the previous versions.

The compiler on the X2 system can perform some vector optimizations for this implementation. The GET operation to copy the value from the parent node is a single operation and it cannot be vectorized. Copying the value to the threads in the same group is a for loop implementation and this helps the compiler to vectorize the code.

4.1.5 GASNet implementation

The GASNet implementation is the most recent implementation of collective operations. The latest version of the Portals conduit is developed entirely over Portals [6]. Previous versions were using the MPI conduit for the active message layer and performing the PUT and GET operations with Portals. This dependency is now removed and the implementation is completely independent.

The communication structure used is a binomial tree. The root node of a binomial tree of order $k$ has as child nodes other binomial trees of orders $k-1, k-2, ..., 1, 0$. There are $2^k$ nodes and the maximum depth is $k$. Figure 4.5 shows a binomial tree of order 4. Similarly to the previous trees each node in the tree represents a UPC thread. Nodes have variable number of child nodes but only one parent node.

![Figure 4.5: Binomial tree of order 4](image)
The implementation uses variants of \texttt{PUT} and \texttt{GET} operations. The communication pattern is not widely known. This implementation was not tested on the X2 system because GASNet was not available.

### 4.1.6 MTU implementation

As mentioned in section 2.1.2 the MTU implementations are flat algorithms. The user’s implementation is the same idea as the \texttt{PUT} version of MTU. The master thread is executing all the workload. For comparison reasons the \texttt{GET} version of MTU was chosen as the last version of the broadcast. This version is also the default implementation on the X2 vector system.

Figure 4.6 shows how the algorithm works.

![Figure 4.6: MTU GET implementation of broadcast](image)

Two barriers are needed for this algorithm, one before the remote accesses and one after. All threads are performing their copy almost simultaneously. This version has all threads working at the same time which is the maximum parallelism level possible. On the other hand all threads are trying to copy the same value from the same target. This could create a lot of traffic on the target node at the process count grows bigger thus causing poor scaling.

All the remote calls in this algorithm are single \texttt{GET} operations and the compiler cannot vectorize any part of the broadcast code. For the X2 vector system this version is precompiled and linked to the final executable at compile time.
4.2 Reduction

The same approach as in broadcast was used for the reduction operations. The difference in reductions is the direction of data movement in the tree structure. Values from all threads must be combined to produce the final result and then broadcast back to the threads. The first step implements a bottom-up algorithm to reduce the values. The second step is a top-down algorithm for broadcasting the result. Since the second step is the same as a normal broadcast the new reduction implementations call the broadcast function to perform this step. A mechanism was implemented in order to take correct timings when calling the broadcast within a reduction (Section 3.1.4). The user implementation is using its own broadcast step. The MTU implementation has no broadcast step and thus is modified to use the normal broadcast function call.

Similarly to the broadcast a number of variables need to be calculated before the reduction can take place. These values are also saved in global static variables and are used for all calls.

Reduction operations

BenchC uses one reduction operation. It sums a scalar double precision value from all threads and also broadcasting the result back. The kind of operation where the result is broadcast to all threads at the end is known as all-reduce operation.

XFlow uses two different data types for its reductions: scalar integer and double precision values. For each data type there is an all-reduce function and also a normal version where the final value is only reduced in one thread. Moreover there are three different operations available for each one of the versions. Summing the values, finding the minimum and finding the maximum value. A function parameter specifies the operation selected. Each versions has its own function call but they all implement the same algorithm.

4.2.1 Default CFD code implementation

The user’s implementation is a simple flat algorithm. The master thread reduces the values in the shared array and calculates the final result. This step translates to GET operations from the master thread. The next step is implemented as a flat PUT broadcast version. The master thread is copying the result in all elements of the shared array.
The serial implementation of both the reduction and broadcast steps may lead to poor scaling. Increase of the process count will increase the workload on the master thread resulting to slower execution.

Figure 4.7 shows both steps of the user’s implementation.

Since only one thread is executing all the workload there is no need for synchronization. Threads only synchronize at the beginning and at the end of the call. After the second barrier they are free to copy the final value from their local element of the shared array to their local variables. This operation is not a remote call because the element of the shared array accessed has local affinity to each thread.

For both CFD codes the implemented algorithm is the same. Both steps are implemented using for loops. The Cray compiler on the X2 system can perform vectorization optimizations in both loops. They have no dependencies between iterations and no conditional statements to break the loop. This specific implementation, despite its simple and non-parallel nature, can be optimized by the compiler and gives better performance. On the XT system vectorization options were not available and the code could not be optimized by the compiler.

4.2.2 Implementation 1

The first reduction implementation is based on the same idea as the second broadcast. The direction of the data movement is now bottom-up. Going from the last level up, child nodes push their values to their parent nodes. The parent performs the reduction with these values and its own, and then pushes the new value to its parent node.

For the algorithm to work each thread needs to have a small shared buffer for storing the values from its child nodes. The buffer has size equal to the rank of the tree. The data structure used is a single shared array for all threads. The block factor is set in such
a way that every thread has its part of the buffer local. Reducing the values from child
nodes should not trigger remote memory accesses. Each child uses its rank identifier
and the tree rank to calculate its father’s rank, and also where in the shared array it
should push its value. The formula gives a unique position in the shared array for each
child.

The shared buffer is initialized with appropriate values before the reduction is executed.
For BenchC the buffer is always set to zero for the global sum to give the correct result.
For XFlow the values differ depending on the operation (SUM, MIN or MAX).

![Figure 4.8: First implementation of reduction](image)

The code performing the reduction of values from the child nodes can be vectorized
for the X2 system. The part of the shared buffer has local affinity to each thread and
the reduction can be performed in a single vector iteration. The rest of the algorithm is
made from single remote calls that cannot be optimized in any other way.

### 4.2.3 Implementation 2

The second implementation is a multi-core aware approach. The threads are grouped
together in groups of four. The group factor is declared as a preprocessor defined value
and can be changed during compilation. Each group has a thread set as group master.
This thread is responsible for performing the reduction within the group and then copy
the value to the next level. For the group master to be able to reduce the group values,
the rest of the threads in the group must place there values in the shared buffer.

The algorithm has three steps. In the first step the group master GETs the values from
the threads in the same group and reduces them. This will happen in all levels of the
tree. In the second step, starting from the last level going up, the group master will place
its value into its parent group master thread shared buffer. The last step is for the parent
to reduce the values from its child nodes. It will also reduce the value it has already
reduced from within the group. Steps two and three will be repeated as many times as
needed until the final result reaches the root node.
Figure 4.9: Second implementation of reduction

The shared buffer is again implemented as a common shared array with block factor equal to the tree rank. Each thread’s buffer has local affinity. The group masters can calculate their father’s shared buffer using their rank identifier and the tree rank. A formula for calculating a unique position in the shared buffer is also implemented using the thread and tree ranks.

4.2.4 MTU implementation

This implementation has two versions. Both are based on PUT operations but different as far as synchronization is concerned.

The first version allocates a shared buffer of size equal to the thread count. Each thread will put its local result to this buffer and the destination thread will reduce the values. The result will remain only in the destination thread. The MTU reductions are not all-reduce implementations.

For the second version a shared lock is allocated and all threads are using it to get access to a shared variable which holds the final reduction result. When a thread gets the lock it reduces its value and then returns the lock. All threads are active in this approach and are all competing to gain access to the shared variable. The final result is again only in the destination thread.

The reductions used in both CFD codes are always using scalar values. Shared scalar values in UPC are always allocated in thread 0. The code was modified to execute a final broadcast step of the result. The broadcast used was chosen to be the MTU version. In both the XT and X2 systems the current implementation used is the MTU. Using the same version for both the reduction and broadcast steps will give a better comparison. The first version is the default reduction and the one used in the benchmarks. The default broadcast is also used (GET version).
Since the Berkeley UPC compiler was used on the XT system this version was not available for benchmarks and it was only used on the X2 vector system.

The next stage after explaining the different implementations that are going to be used is to perform the benchmarks. The next chapter describes the input parameters of the two CFD codes and presents details about the execution times. Finally it comments on the graphs from the data taken.
Chapter 5

Benchmarks and Analysis

5.1 Input parameters

In the next subsections the input parameters used for the benchmarks are discussed. The two CFD codes have some similar options because they were both developed by the same research center.

5.1.1 BenchC

The parameters used in both the XT and X2 systems are the same except from one. The time step parameter (nts) has a different value. On the X2 system each time step needs less time to execute. Depending on the available allocation units (AUs) [16] the time steps performed for each system were adjusted in order to be able to run all needed executions for a good time estimate. For the XT system the value was set to 4 and for the X2 system was set to 16.

Depending on the number of time steps used the broadcast function is executed 2624 times on the X2 system and 656 times on the XT. The reduction function is executed 14848 times on the X2 system and 3712 times on the XT.

Table 5.1 shows all the options specified in the input file with their values and a small description. The same parameters were used for all implementations and process counts.
<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>turb</td>
<td></td>
<td>Enables Smagorinsky LES turbulence mode</td>
</tr>
<tr>
<td>quiet</td>
<td></td>
<td>Print to screen only necessary information</td>
</tr>
<tr>
<td>nts</td>
<td>16</td>
<td>Number of time steps</td>
</tr>
<tr>
<td>timeit</td>
<td></td>
<td>Enable timings for various parts of the code</td>
</tr>
<tr>
<td>nodata</td>
<td></td>
<td>Do not output any data during the marching algorithm. A final output file is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>always created at the end of the simulation with the velocity and pressure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>variables</td>
</tr>
<tr>
<td>rcb</td>
<td></td>
<td>Enable the RCB mesh partitioning algorithm after the mesh is read from the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>file</td>
</tr>
<tr>
<td>nit</td>
<td>4</td>
<td>Number of non linear iterations at each time step</td>
</tr>
<tr>
<td>dt</td>
<td>0.1</td>
<td>The size of the time step. Is application depended and related to mesh</td>
</tr>
<tr>
<td></td>
<td></td>
<td>characteristics</td>
</tr>
<tr>
<td>ntsbout</td>
<td>4</td>
<td>Abbreviation for (n)umber of (t)ime (s)teps (b)efore (out)put. The solution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>will be written to file after the number of time steps set by this variable</td>
</tr>
<tr>
<td>alpha</td>
<td>0.5</td>
<td>The time-marching ’trapezoidal-rule stabilization’ parameter</td>
</tr>
<tr>
<td>viscosity</td>
<td>0.0001</td>
<td>The viscosity of the fluid</td>
</tr>
<tr>
<td>noslip</td>
<td>1</td>
<td>Apply zero-velocity (no-slip) boundary condition on boundary 1</td>
</tr>
<tr>
<td>noslip</td>
<td>2</td>
<td>Apply zero-velocity (no-slip) boundary condition on boundary 2</td>
</tr>
<tr>
<td>noslip</td>
<td>3</td>
<td>Apply zero-velocity (no-slip) boundary condition on boundary 3</td>
</tr>
<tr>
<td>noslip</td>
<td>1.0</td>
<td>Specifies the x-component of velocity on inflow boundary 1</td>
</tr>
<tr>
<td>done</td>
<td></td>
<td>Indicates the end of the input parameters</td>
</tr>
</tbody>
</table>

Table 5.1: Input parameters for BenchC
The time spent for collective operations compared to the total time needed by the CFD code is displayed in Tables 5.2 and 5.3. The collectives time in the tables below is measured as the time spent in the collective operations optimized, the broadcast and the reduction operations.

For the XT system the percent of time is significant and optimizing the collective operations could benefit the total execution time when executing with a high process count. The X2 vector system has only 112 processing units but for a bigger system the percent is expected to increase. On both systems the total overall time decreases as the process count grows but the time spent in collective operations increases.

<table>
<thead>
<tr>
<th>Process count</th>
<th>Collectives time (sec)</th>
<th>Total time (sec)</th>
<th>Percentage of total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>5</td>
<td>1310</td>
<td>0.38 %</td>
</tr>
<tr>
<td>64</td>
<td>9</td>
<td>933</td>
<td>0.96 %</td>
</tr>
<tr>
<td>128</td>
<td>17</td>
<td>653</td>
<td>2.60 %</td>
</tr>
<tr>
<td>256</td>
<td>34</td>
<td>460</td>
<td>7.39 %</td>
</tr>
<tr>
<td>512</td>
<td>68</td>
<td>341</td>
<td>19.94 %</td>
</tr>
<tr>
<td>1024</td>
<td>130</td>
<td>309</td>
<td>42.07 %</td>
</tr>
</tbody>
</table>

Table 5.2: Percent of time spent in collective operations [BenchC, XT4]

<table>
<thead>
<tr>
<th>Process count</th>
<th>Collectives time (sec)</th>
<th>Total time (sec)</th>
<th>Percentage of total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>321</td>
<td>0.31 %</td>
</tr>
<tr>
<td>16</td>
<td>1.1</td>
<td>159</td>
<td>0.69 %</td>
</tr>
<tr>
<td>32</td>
<td>1.6</td>
<td>81</td>
<td>1.97 %</td>
</tr>
<tr>
<td>64</td>
<td>2.2</td>
<td>42</td>
<td>5.23 %</td>
</tr>
<tr>
<td>112</td>
<td>3.6</td>
<td>25</td>
<td>14.40 %</td>
</tr>
</tbody>
</table>

Table 5.3: Percent of time spent in collective operations [BenchC, X2]
5.1.2 XFlow

The values of the input parameters are the same for both systems. Similarly to the BenchC code the number of time steps executed for each system were adjusted depending on the available resources [16]. For the XT system the value of time steps was set to 4 and for the X2 system was set to 16.

Depending on the number of time steps used the broadcast function is executed 7301 times on the X2 system and 1699 times on the XT. The reduction function is executed 47631 times on the X2 system and 11883 times on the XT.

Table 5.4 shows all the options specified in the input file with their values and a small description. The same parameters were used for all implementations and process counts.

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kspace_mm</td>
<td>40</td>
<td>The GMRES Krylov space size</td>
</tr>
<tr>
<td>nodata</td>
<td></td>
<td>Do not output any data during the marching algorithm. A final output file is always</td>
</tr>
<tr>
<td>ramp</td>
<td>300</td>
<td>Set the ramp-up range for the inflow velocity</td>
</tr>
<tr>
<td>nts</td>
<td>16</td>
<td>Number of time steps</td>
</tr>
<tr>
<td>ntsbout</td>
<td>1</td>
<td>Abbreviation for (n)umber of (t)ime (s)teps (b)efore (out)put. The solution will be written to file after the number of time steps set by this variable</td>
</tr>
<tr>
<td>ntsbrem</td>
<td>2</td>
<td>Abbreviation for (n)umber of (t)ime (s)teps (b)efore (rem)esh</td>
</tr>
<tr>
<td>numfor</td>
<td>2</td>
<td>The number of forces to compute during each run</td>
</tr>
<tr>
<td>spin1</td>
<td>1 Z -0.65 0.0 -1.0</td>
<td>Parameters for spinning Z-axis around X and Y</td>
</tr>
<tr>
<td>spin2</td>
<td>2 Z 0.65 0.0 1.0</td>
<td>Parameters for spinning Z-axis around X and Y</td>
</tr>
<tr>
<td>xfix</td>
<td>3</td>
<td>Set boundary type for X velocity</td>
</tr>
<tr>
<td>yfix</td>
<td>4</td>
<td>Set boundary type for Y velocity</td>
</tr>
<tr>
<td>zfix</td>
<td>5</td>
<td>Set boundary type for Z velocity</td>
</tr>
<tr>
<td>mem_buff_pct</td>
<td>100.0</td>
<td>Percent of extra memory to allocate</td>
</tr>
<tr>
<td>dt</td>
<td>0.0174</td>
<td>The size of the time step. Is application depended and related to mesh characteristics</td>
</tr>
<tr>
<td>viscosity</td>
<td>0.004</td>
<td>The viscosity of the fluid</td>
</tr>
<tr>
<td>done</td>
<td></td>
<td>Indicates the end of the input parameters</td>
</tr>
</tbody>
</table>

Table 5.4: Input parameters for XFlow
The time spent in collective operations for this code starts from a small percent and slowly increases. Due to the small data set the benchmarks were executed with relatively small process counts. Even so, the percent of time for collective operations reached a value of around 40% for 256 processes on the XT system and is expected to increase for more processes. For the X2 system a similar behaviour is observed.

<table>
<thead>
<tr>
<th>Process count</th>
<th>Collectives time (sec)</th>
<th>Total time (sec)</th>
<th>Percentage of total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>16</td>
<td>377</td>
<td>4.24 %</td>
</tr>
<tr>
<td>64</td>
<td>28</td>
<td>342</td>
<td>8.18 %</td>
</tr>
<tr>
<td>128</td>
<td>68</td>
<td>271</td>
<td>25.09 %</td>
</tr>
<tr>
<td>256</td>
<td>115</td>
<td>293</td>
<td>39.24 %</td>
</tr>
</tbody>
</table>

Table 5.5: Percent of time spent in collective operations [XFlow, XT4]

<table>
<thead>
<tr>
<th>Process count</th>
<th>Collectives time (sec)</th>
<th>Total time (sec)</th>
<th>Percentage of total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1.8</td>
<td>62</td>
<td>2.90 %</td>
</tr>
<tr>
<td>16</td>
<td>1.5</td>
<td>30</td>
<td>5.00 %</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>17</td>
<td>11.76 %</td>
</tr>
<tr>
<td>64</td>
<td>2.5</td>
<td>13</td>
<td>19.23 %</td>
</tr>
<tr>
<td>112</td>
<td>3</td>
<td>12</td>
<td>25.00 %</td>
</tr>
</tbody>
</table>

Table 5.6: Percent of time spent in collective operations [XFlow, X2]
5.2 Benchmarks

In this section graphs from both CFD application are plotted and discussed. Each subsection is focused on one system architecture.

The benchmarks are separated in two stages. In the first stage tests are performed to observe the effect of different tree orders (rank) for the new implementations. The second stage is a comparison between existing and new implementations.

Each point in the graphs is an average of three executions. The physical topology of the processors requested on the XT and X2 systems is dynamic and not known in advance. Each execution was performed with a different physical topology for realistic results. The performance of the collective operations is topology dependent and timing executions from a variety of topologies gives a better estimate of the performance.

The graphs are going to be commented on using two measures. The performance of the functions and their scaling factor. The performance of each implementation is the time spent in all broadcast calls and is represented by the Y-axis of the graph. The scaling factor is not measured in units. It is how well the implementation scales as the number of processors grows.

5.2.1 Cray XT

Broadcast

The graphs in Figure 5.1 show how the new implementations perform for process counts of 32 to 1024. These graphs are for the BenchC CFD code, which uses only a scalar double value for broadcast.

Figure 5.1(a) shows the first implementation’s results for tree rank values of 4, 8 and 16. The different ranks do not have a clear effect on the performance of this version. For rank 4 there is a big step in performance between process counts 256 and 512. This step is due to the need of an additional barrier. The same behaviour is observed for tree rank 16. For tree rank 8 this increase in processes is not creating an additional tree level and thus the time spent in the broadcast is increased by a smaller factor. The big steps for rank 8 are observed between 64 and 128 processes and also between 512 and 1024. For the first step the same reason as above applies, the additional barrier. The second big step is due to the large increase of PUT operations. For 512 processes the algorithm performs 64 PUTs and 448 GETs. For 1024 processes the GET operations increase to 520 and the PUT operations to 504. The PUT operations are executed in a serial order. The parent thread is copying the value to each one of its child nodes. For tree rank 8 the parent executes a for loop of size 8. The execution of this part of the code is increased for 1024 processes and it causes poor scaling.
The second broadcast implementation, Figure 5.1(b), has similar behaviour. For tree ranks 4 and 16 there is a big step in performance between 256 and 512 processes. This step is not observed for rank 8 because the tree structure remains the same. For the other two ranks an additional barrier is needed because of the new tree level created. A new tree level is created for rank 8 between 64 and 128 processes and also between 512 and 1024. The additional barrier introduces more synchronization and thus affects the execution time of the broadcast. As the tree continues to grow, the same behaviour is expected when a new level is added.

The last graph, Figure 5.1(c), shows the third implementation’s results for the same tree ranks. For this implementation the different ranks have an obvious effect on the performance of the algorithm. As the rank gets bigger the broadcast takes less time to complete. Similarly to the previous implementations, the additional barriers added due to the growth of the tree have an effect on the performance. For tree ranks 4 and 8 an additional barrier is needed between 256 and 512 processes. For tree rank 16 a barrier is added when going from 64 processes to 128. A big increase in the time is also observed for tree rank 16 when executing with 1024 processes. No additional barrier is needed but the big increase in processes causes the total execution time of the broadcast to increase by a bigger factor.
For all implementations the same negative effect is observed when adding a barrier in the process. The barrier is a costly synchronization mechanism for the XT system and it should be used only when needed. More barriers are going to be needed as the process count gets bigger, and thus more levels will be added to the tree structure.
Although this is a bad characteristic for the new tree structures its occurrence will become less frequent as the tree grows. The number of nodes in a new level is equal to the number of nodes in the previous level multiplied by the tree rank. This creates a geometric increase in nodes every time a new level is created and thus the need of new levels will become less frequent as the tree gets bigger.

A comparison between all the different implementations of the broadcast can be seen in Figure 5.2. The third implementation has the best scaling factor and also the lower timings for the broadcast. For this implementation the best results are achieved using a rank value of 16. For this reason the comparison is made using the 16 rank tree structures for all new implementations. The comparison graphs with ranks 4 and 8 are also available in Appendix B.

The graphs are plotted with both axis in logarithmic scale for better view of the results. The default implementation has the worse scaling factor due to its serial implementation. As the process count is growing the load in the master thread is increasing and more time is needed for the broadcast to execute. The MTU flat GET implementation has a better scaling factor but is again scaling poorly as the process count is growing. The tree based implementations have a better scaling factor compared to the default and MTU versions. As the process count doubles, the time needed by the broadcast is increasing by a small factor. The third implementation has the best scaling factor and also very low execution times. The GASNet implementation has a steady and good scaling factor for all process counts. As mentioned in Section 2.1.1 the new GASNet Portals conduit has new improved versions of the collective operations and it was expected to perform and scale well on the XT system. The last implementation plotted is the MPI version of the broadcast. The Cray XT system is assembled for message passing programming and the MPI library used is specially developed by Cray for this architecture. As it can be seen from the graph the scaling factor is very good and the time spent on the broadcast very low.

The speedup gained from using the third implementation over the default implementation starts at a value of 3 and increases to approximately 16 as the process counts grows. The GASNet implementation starts with no benefit over the default code but as the processes increase the speedup reaches a value of approximately 25.
The XFlow graphs for testing the different tree ranks for the new implementations have the same results as in BenchC. These graphs are not shown here but they can be found in Appendix B. Figure 5.3 shows the comparison of all broadcast implementations using the XFlow CFD code. Due to the small size of the data set the code could only be executed for process counts of 32 to 256. The code is entirely written in UPC and there is no MPI version available for comparison. The scaling factors of the new implementations are the same as in BenchC.

The speedup gained by using the third implementation over the default is around 4 for small process counts and increases to 7.5 for 256 processes. The GASNet implementation has similar behaviour as in BenchC. It starts with a worse speedup gain but outperforms the third implementation after 128 processes.
Reductions

The graphs in Figure 5.4 show the tree rank benchmarks for the two reduction implementations. These graphs are for the BenchC CFD code and for process counts of 32 to 1024.

The first graph, Figure 5.4(a), shows the first implementation’s results. While the effect of the rank is not clear, using a rank value of 8 has a steady behaviour and good scaling factor. A big difference in time spent is observed between 64 to 128 processes due to the additional barrier needed, and also between 512 and 1024 processes. For tree rank 16 an additional barrier is needed between 256 and 512 processes. Despite the fact that the increase in processes from 512 to 1024 is much bigger than from 256 to 512, the scaling of the implementation is much better. The additional synchronization needed due to a new level in the tree structure seems to cost a lot. For tree rank 4 the graph is very smooth in all parts and the effect of the additional barriers is not easily detected. An additional barrier is needed when going from 64 to 128 processes and also from 256 to 512.

The second implementation’s results can be seen in Figure 5.4(b). Similar observations as for the previous implementation apply to this as well. The big steps are due to the additional barriers. For tree rank 4 these steps are between 64 and 128 processes and also between 256 and 512. For tree rank 8 between 32 and 64 and also between 256 to 1024.
and 512. Finally for tree rank 16 one big step exists between 64 and 128 processes. A comparison between all ranks shows that rank 4 has the worst scaling factor. The graphs for ranks 8 and 16 cross each other at several points and it is not clear which one is considered to perform better.

It is worth mentioning that in reductions the order of the operations can affect the final result. Both CFD codes are using a global sum function. The rounding errors when adding double precision numbers differ for each implementation depending on the order the numbers are added. For all implementation the order of the addition is predefined and the results are reproducible.

![First implementation of reduction (BenchC, Cray XT4)](image)

(a) First implementation
Since the difference between ranks is not clear, for comparing the reduction implementations rank 8 was chosen. The comparison graphs using ranks 4 and 16 can be found in Appendix B. Figure 5.5 shows the graphs for all implementations using a tree rank of 8. The graph with the worst scaling factor (red line) represents the default, flat PUT algorithm implementation. As the number of processes double, the work load on the master thread also increases and the algorithm needs more time to execute. The scaling factor of the new tree based implementations is almost the same, with the second implementation performing better. As the number of processes is growing both implementations need a small amount of extra time to execute the reductions. The last graph represents the MPI version. As mentioned before the MPI library is developed specially for the XT system and performs and scales very well. Although the time needed by the MPI version is much less than the new implementations, the scaling behaviour is similar.
The graphs for XFlow have again similar behaviour as in BenchC on the XT system. These graphs can be found in Appendix B. Figure 5.6 shows the comparison of all reduction implementations using the XFlow CFD code. The MPI version is not available for this code and the only comparison possible is between the default implementation and the new algorithms.
The architecture of the Cray XT system is developed for message passing programming. The implementation of single sided communications is developed in software with no hardware support. This may introduce an overhead for RMA calls that causes poor performance on the XT system. The new implementations have similar scaling behaviour with the MPI versions but need more time to execute.
5.2.2 Cray X2

The same benchmarks were executed for the X2 system as well. The system has only 112 processing units available (see Section 2.2.2) and both CFD codes were tested with 8 to 112 processes.

Broadcast

For the first broadcast implementation, Figure 5.7(a), the rank is not affecting a lot the scaling factor. The graphs for all ranks are almost identical, except for process counts of 32 and 112 that have small differences. Another observation is that the barriers are not easy to detect when looking at the graphs. The different ranks need additional barriers for different process counts but their graphs have the same behaviour. As the process count changes the additional time needed by all ranks is almost the same.

Figure 5.7(b) shows the results for the second implementation. For this implementation as the rank is growing the algorithm performs and scales better. Looking at the graph for rank 4 it can be seen when the algorithm needs an additional barrier. Going from process count 16 to 32 and also from 64 to 112 are the two cases where the tree structure grows by one level and more synchronization is needed. For rank 8 the same behaviour is detected between process counts of 64 to 112. For rank 16 the process count is not large enough to cause an additional synchronization point and thus the graph is smoother.

The third implementation of the broadcast, Figure 5.7(c), has similar behaviour to the second implementation. The rank of the tree has an effect on the scaling of the algorithm. For process counts of 8 and 16 the graphs for all ranks are identical. After that they split and each rank has a different scaling. For rank 4 the two cases where an additional barrier is added are observed, as before, between process counts of 16 and 32, and 64 and 112. For tree rank 8 the additional barrier is added when using 112 processes. Although there is no additional barrier between 32 and 64 processes the scaling factor is not so good.
First implementation of broadcast [BenchC, Cray X2]

(b) Second implementation
The first observation is that the worst performance is given by the MPI and MTU implementations. In contrast to the XT system, the X2 vector system is build towards shared global address space languages (see Section 2.1). This makes the system a very good choice for executing codes written in UPC. The Cray UPC compiler offers a variety of optimizations and produces a very good final code [4]. The MTU flat GET implementation has a very good scaling factor for process counts lower than 64 processes. For 112 processes it performs very poorly and the time needed to execute the broadcast is increased almost by a factor of 10. The number of GET operations with the same target, the master process, is increased significantly and this might create a lot of traffic in a single node, something that causes poor performance. As far as the MPI version is concerned the implementation of the broadcast is not publicly known. The only available information is that a version specially for the X2 system is still under development and the one currently used has no specific optimizations for the vector system.

Comparing the remaining implementations it can be seen that the new algorithms are not performing better than the default flat implementation. The compiler is able to vectorise the default implementation and thus parallelise the procedure. Since the BenchC code only uses a single broadcast data type, a scalar double precision number, the code is not very complex and easy to optimize. All processes in the vector system have a view of the whole memory available and by vectorising the broadcast the master process can
perform the \texttt{PUT} operation in all threads by using a single vector operation.

The new implementations match the performance and scaling of the default implementation when compiled using a rank value of 16, but perform worst with lower values (see Appendix B). Moreover the graphs for all three implementations have similar performance and scaling, with the first implementation having a small advantage over the other.

![All broadcasts (tree rank = 16) [BenchC, Cray X2]](image)

**Figure 5.8:** All broadcast implementations comparison [BenchC, X2]

The graphs for the rank benchmarks using the XFlow CFD code show the same results and thus are not shown here. They can be found in Appendix B along with the other graphs.

For the XFlow code, Figure 5.9, the comparison favours the new implementations. The MTU version on the other hand has similar behaviour as before. It scales and performs steadily for process counts lower than 64 processes but scales poorly for 112 processes. Having the same poor scaling in both CFD codes verifies the abnormal results of the implementation.

The default implementation of the broadcast has a slightly better performance than the MTU version and it also scales well for all process counts. Due to the three different broadcast data types used in XFlow, the comparison between the new and the default implementations has changed. The new codes are performing better for this CFD code, with a similar scaling factor to the default implementation. The performance of the new versions is again similar to each other.
On the X2 system it has no distinctive difference which version of the broadcast it is used. The parameter affecting all implementations is the tree rank value, something that did not give clear results in the XT system.

Figure 5.9: All broadcast implementations comparison [XFlow, X2]
Reductions

The following graphs show the benchmarks for the reduction functions on the X2 system for both CFD codes used.

The first graph, Figure 5.10(a), shows the first implementation’s tree rank benchmarks. The main observation is that, similarly to the broadcast function, the rank of the tree makes a very small difference. All ranks have similar graph behaviour and scaling factor as the process count is growing. For each rank the barriers needed for synchronization are different for every process count but it does not affect the overall performance.

For the second implementation, Figure 5.10(b), the same observations are made. The tree rank has a small effect on the performance and scaling, and the overall behaviour is the same for all process counts.

Because it was not possible to run the benchmarks with a bigger process counts it cannot be seen if the rank makes a significant difference. For 112 processes both implementations’ graphs are starting to have a bigger performance gap. On the XT system they are more cores available and thus the benchmarks can give more information about the performance and scaling.
The comparison between all implementations, Figure 5.11, shows that the default implementation has a better performance than the others. The simple implementation of the reduction in the BenchC code helps the compiler to perform good optimizations and parallelise the code. The complex structure of the new implementations has an algorithmic advantage over the flat version, but the compiler produces a better final executable for the simple code. Although the performance is not as good as the default version the scaling factor of the new implementations is similar to the default.

The MTU implementation is performing poorly due to two reasons. The first is because it is using the MTU broadcast as the second step of the reduction to broadcast the reduced value to all threads. The second reason is that it dynamically allocates memory in order to collect the values from all threads and reduce them locally. Memory allocation is a costly operation and affects the overall time needed by the reduction. The new implementations are using static arrays as buffers and do not need to allocate and free memory for every function call.

The implementations with the worse performance are the MPI and MTU versions. For the MPI version the same explanation as for the broadcast applies. The current library is a standard implementation of the message passing interface and offers no optimizations for the vector system. This code might not be able to take advantage of the architecture of the system and thus performs poorly.
All implementations have similar scaling factors but their performance is quite different. The best implementation is approximately 20 times faster than the worst for small process counts and has an even bigger difference for large counts. Taking into account that the graphs have logarithmic scale in both axis, the difference between the various implementations is increasing very fast for the last process counts.

Figure 5.11: All reduction implementations comparison [BenchC, Cray X2]
The comparison graph for XFlow, Figure 5.12, shows a very different behaviour of the default implementation. As the process counts grows the time needed by the reduction functions is reducing rather than increasing. The code was executed multiple times to make sure that the timings were accurate and reproducible. In section 4.2.1 was mentioned that the compiler is vectorising both the reduction and broadcast loop for this implementation. As the process counts grows a bigger part of the machine is dedicated to the execution of the code, which means less interference from other codes executing on the system that might use the interconnect. For 112 processes the code is running on the entire system. The vectorised instructions may then be able to execute faster and thus give better results than for smaller counts.

The graphs for the MTU and the new implementations have similar behaviour as in the BenchC code. The new codes show the best scaling and performance, with the default version matching their performance for 112 processes. The reduction operations in XFlow (see Section 3.2) have a more complex functionality than in BenchC. Even though the implementation of the default version is identical the results from the benchmarks are quite different.

Figure 5.12: All reduction implementations comparison [XFlow, X2]
Chapter 6

Conclusions and Future Work

6.1 Conclusions

The results taken from the benchmarks are very promising for the two collective operations examined. All implementations have good performance and very good scaling behaviour. On both systems the new codes outperformed the default user’s version, and the system’s current default implementation.

On the XT system the current version of the collective operations is a flat algorithm which needs very high execution times and has linear scaling as the process count grows. The GASNet implementation is the only available version which supports the full collectives specifications manual [8] and offers optimized code for the XT system. The code is developed using the GASNet high performance communication system [3], which needs to be installed on the system in order for the implementation to work.

The X2 system uses the same implementation as the XT. Although it is a flat algorithm the compiler can vectorise the code and thus introduce parallelism. Moreover the system is geared towards the shared global address space programming model which benefits programming languages like UPC. The XT system on the other hand is geared towards message passing programming. Comparing the MPI against the UPC versions of the collectives showed how well the MPI library performs on the XT system, and also the potential gain from using languages like UPC on the vector system.

The BenchC broadcast and reduction code has a very good performance and scaling on the X2 system. Because of the simple structure of the code and the data sent, only one version for a scalar double precision value, the compiler can perform vectorization optimizations and make the procedure parallel. Despite the fact the new implementations have a big speedup gain over the default user’s implementation on the XT system they are not able to perform better on the vector system. The XFlow code on the other hand has an identical user implementation for the collective operations but fails to outperform the new implementations. The differences between the two CFD codes are in the
data used and in the reduction operations. XFlow supports more reduction operations and has additional broadcast functions for 2 integers and a scalar integer value. The collective operations library could support special optimized versions of the functions just for scalar values. Simple codes can be optimized using the compiler and offer better performance than general versions that support variable length send buffers.

### 6.2 Future work

In order to keep the project within specific time limits only some of the collective operations were chosen to be implemented and benchmarked. The two CFD codes used offer the possibility of benchmarking the *scatter* and *gather* operations. These functions have complex code structures and they require more development time in order to implement their tree based versions. Another limiting factor is the resources available for testing and benchmarking the new codes. Allocations units for supercomputers have a cost and there are budget limitations that have to be taken into consideration.

Another future plan is to test the codes for bigger process counts. The maximum number of processes used in the project was 1024 using a big BenchC data set. The data set used in XFlow was a limiting factor for the benchmarks on the XT system and executions for only up to 256 processes were possible. The HECToR Cray XT supercomputer currently supports queues from 8 to 16384 processing cores. Benchmarking and analysing the codes for such process counts will give better information about the true performance and scaling.

A last suggestion for future work is to alter the synchronization mechanism of the tree based algorithms. The current implementations are using barriers for synchronizing all threads. In tree structures synchronization is only needed between successive levels and not between all threads. Moreover, threads can exit the broadcast function as soon as they have the data and successfully propagate them to the next level. Due to the use of barriers all threads need to wait and perform barriers until all the threads finish.
Bibliography


Appendix A

HECToR modules

A.1 Cray XT4

On login, the system loads the default modules for the XT system. Below is the list of modules:

1. modules/3.1.6  
2. pbs/8.1.4  
3. packages  
4. pgi/8.0.2  
5. ase-opt/2.1.56HD  
6. xt-libsci/10.3.2  
7. xt-mpt/3.1.0

8. xt-pe/2.1.56HD  
9. xt-asyncpe/3.0  
10. PrgEnv-pgi/2.1.56HD  
11. xt-service/2.1.56HD  
12. xt-libc/2.1.56HD  
13. xt-os/2.1.56HD  
14. xtpe-target-cnl

The command for swapping the XT default modules with the X2 modules is:

module add x2-env
When executed the list of modules changes to the one below:

1. CC/6.0.0.7
2. cal/2.1.0.4
3. PrgEnv-x2/6.0.1
4. cftn/6.0.0.7
5. x2-mpt/1.0.9
6. craylibs/6.0.0.8
7. libsci/6.0.0.3.chip21
8. x1x2-gcc/1.0.1
9. x2-sysroot/1.1.0
10. x1x2-pe/6.0.0.5
11. pbs/8.1.4
12. x2-env
Appendix B

Graphs

B.1 Cray XT

B.1.1 Broadcast

![Graph: All broadcasts (tree rank = 4) [BenchC, Cray XT4]](image)

Figure B.1: All broadcast implementations comparison (tree rank 4) [BenchC, XT4]
Figure B.2: All broadcast implementations comparison (tree rank 8) [BenchC, XT4]
Figure B.3: Broadcast tree rank benchmarks [XFlow, XT4]
Figure B.4: All broadcast implementations comparison (tree rank 4) [XFlow, Cray XT4]

Figure B.5: All broadcast implementations comparison (tree rank 8) [XFlow, Cray XT4]
B.1.2 Reduction

All reductions (tree rank = 4) [BenchC, Cray XT4]

Figure B.6: All reduction implementations comparison (tree rank 4) [BenchC, XT4]

All reductions (tree rank = 16) [BenchC, Cray XT4]

Figure B.7: All reduction implementations comparison (tree rank 16) [BenchC, XT4]
Figure B.8: Reduction tree rank benchmarks [XFlow, XT4]
Figure B.9: All reduction implementations comparison (tree rank 4) [XFlow, Cray XT4]

Figure B.10: All reduction implementations comparison (tree rank 16) [XFlow, Cray XT4]
B.2 Cray X2

B.2.1 Broadcast

Figure B.11: All broadcast implementations comparison (tree rank 4) [BenchC, X2]
Figure B.12: All broadcast implementations comparison (tree rank 8) [BenchC, X2]
Figure B.13: Broadcast tree rank benchmarks [XFlow, X2]
All broadcasts (tree rank = 4) [XFlow, Cray X2]

Figure B.14: All broadcast implementations comparison (tree rank 4) [XFlow, X2]

All broadcasts (tree rank = 8) [XFlow, Cray X2]

Figure B.15: All broadcast implementations comparison (tree rank 8) [XFlow, X2]
B.2.2 Reduction

Figure B.16: All reduction implementations comparison (tree rank 4) [BenchC, Cray X2]

Figure B.17: All reduction implementations comparison (tree rank 8) [BenchC, X2]
Figure B.18: Reduction tree rank benchmarks [XFlow, X2]
Figure B.19: All reduction implementations comparison (tree rank 4) [XFlow, X2]

Figure B.20: All reduction implementations comparison (tree rank 8) [XFlow, X2]
Appendix C

Work Plan

Figure C.1 shows the work plan of the project. The original schedule proposed in the Project Preparation [7] course was kept the same until the end of the project. The report was written throughout the whole duration of the project.

Figure C.1: Gantt diagram