# An Implementation of the Coarray Programming Model in C++

Don M. J. Browne

August 2013

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2013

## Abstract

The Coarray model is an extension to Fortran which implements a Partitioned Global Address Space programming model on top of standard Fortran constructs by providing a simple notation for data decomposition and single sided data transfers. This project investigates the feasibility of implementing this model as an object oriented library on top of C++ - providing a similar ease of use without having to modify the language or any of its tools. The library will make use of the OpenSHMEM API for single-sided communications, which will allow the code to be ported to a variety of High Performance Computing systems. The performance of this library will also be investigated on the UK national supercomputing system HECToR.

# Contents

# List of Figures

# Acknowledgements

I'd like to thank James Perry for supervising my project. I'd also like to thank David Henty for the advice and information he provided me for this project.

I'd like to thank my friends, family and classmates for their support and encourage. Particular mention is due to my ISC competition team mates - Swapnil Gaikwad, Nikos Koutsikos and Paolo Martino.

Thanks are also in order to the guys at Boston and Viglen, including (but not limited to) Dave, Mike, Jon, Matt and Roshan for letting the four of us use their hardware and really going out of their way to support us.

# Chapter 1

# Introduction

High Performance Computing (HPC) systems continue to provide more and more raw computing power to academia and industry, and the sort of computing power that was formerly the domain of large and extremely expensive supercomputer systems is becoming readily available to small businesses and individual research groups. There is a great deal of legitimate concern that it will be difficult, if not impossible, to keep producing faster and more affordable systems at the rate at which they have been developed so far.[17] However, a potentially more serious problem is that current HPC systems are often severely underutilized, with only a handful of applications available to exploit the full performance of today's most capable machines.[1] The root cause of this problem is that the sort of machines that are being built are different to those that have gone before them. There has been a move away from small numbers of specialized processors towards large numbers of low cost and low power commodity processors. This requires applications to scale to processor counts that were inconceivable at the time of their creation. Another issues is the increasing complexity of memory systems required for these large scale parallel machines. They have complicated hierarchical memory structures where certain portions of memory within a shared memory node are quicker to access than others, and network limitations cause certain remote memory regions to be slower to access than others. Many pre-existing programming models do not adequately address these issues, and thus make it difficult for the programmer to write efficient code that takes them into account. There are two main issues that limit this scalability - one is algorithmic, some software is not designed in such a way that exploits large-scale parallelism. The other lies in the programming models used themselves - some of them incur too much overhead when dealing with large numbers of processors. Some are too awkward or difficult to use, and waste the programmer's time on trying to solve the problems of the programming model instead of the problem that their application is trying to solve.

In the search for better ways of programming HPC systems, much research has gone into so-called Partitioned Global Address Space languages (PGAS) which combine the relative simplicity of shared memory programming, with the support for distributed and hierarchical memory that traditional message passing programming provides. While the exact design varies from language to language, these languages make the notion of remote and hierarchical memory into first class constructs in the language, and allow the programmer to access remote data in much the same way that local data is accessed. There is also an emphasis on single-sided communications, which allow one processor to modify another processor's data without co-ordination between the two processors. In order to make these languages as easy to use as possible, they have either been designed from scratch, or more commonly, by modifiying and extending a language that has already found favour in the HPC community. In modifying the base languages, the language designers have allowed the PGAS constructs to feel like a proper part of the language, in constrast to some of the unwieldy APIs that have been used for HPC programming, such as MPI. It has also allowed them to create compilers that are aware of the PGAS constructs and that can optimize the PGAS code to exploit the underlying hardware as efficiently as possible. All this comes at a cost - the implementers must maintain separate compilers and debuggers for these PGAS languages, and the compatibility between these languages and legacy languages (and between each other) is often limited. Often the limiting factor in the performance and usability of these development tools is how much time and money has been expended in their development. With a relatively large amount of different languages serving a relatively niche field, it can be difficult for vendors to create efficient tools, and potential users can be put off using the languages by doubts as to whether the language will ever gain commercial traction.

In this context, a library that implemented PGAS semantics on top of an unmodified language could prove to be quite useful. While it be difficult to provide automatic optimization compared with a solution that used a special compiler, it would potentially allow fewer disruptive changes to existing software toolchains, allow a more straightforward means of adding PGAS semantics to a piece of software, or create means of prototyping a port to a dedicated PGAS language from a legacy language.

In order to provide a compelling alternative to existing HPC programming libraries, the library should work in way that is as unobtrusive as possible, and if possible, feels like an inbuilt part of the language. The C++ language provides certain facilities which are useful in this regard:

- Object Orientated Programming: This allows an encapsulation of concerns which hides the inner workings of a library from the programmer who uses it.
- Operator Overloading: This allows the behaviour of certain inbuilt language operations to be redefined by the programmer for data types that they have defined. This reduces some of the syntactic burden associated with third party libraries.
- Template Metaprogramming: This allows programmer-defined container types, such as arrays, to be defined in general terms that are not tied down to any specific types. In other words, programmer-defined data structures can be made to make with arbitrary data types without any need for duplicated code.

This project sets out to create a library in C++ that offers similar behaviour and functionality to Coarray Fortran - one of the original and most popular PGAS programming languages. Coarray Fortran was created with the explicit aim of making as being as simple and unobtrusive while still being useful. This aspect of its design allows a large portion of its behaviour to be imitated as a library, and the feasibility of implementing its feature set as a C++ library has been validated elsewhere. This project differs from previous efforts not only by specific implementation details, but by the underlying communications library - previous efforts either used a custom MPI-based single sided communications layer that required dedicated threads for each image[5], or a proprietary single sided library[6]. This project uses the standard OpenSHMEM library, which provides straightforward PGAS-like semantics, and has implementations that will run on a wide variety of machines, from SMP machines to commodity clusters linked by Ethernet or Infiniband, to highly specialized HPC machines with specialized networks.

Chapter 2 of this paper will provide background detail on parallel programming models, PGAS languages, and communications libraries. Chapter 3 will describe certain details of the C++ programming language that are pertinent to the project. Chapter 4 describes the implementation of the library and its features. Chapter 5 will discuss how the library was tested and benchmarked. Chapter 6 will discuss the conclusions from this work, and the opportunities for future work.

# Chapter 2

# Background - Programming Models

## 2.1 Traditional Parallel Programming Models

Parallel computer architectures are split into two categories depending on the organization of their main memory[19] - Shared Memory and Distributed Memory. Shared memory machines have a single address space that is shared between all processors in the parallel system. This category can be further subdivided into symmetric multiprocessor (SMP) systems, where all processors can the entirety of the main memory at the same rate. SMP systems do not scale well as contention for the single memory bus causes the average latency for a memory access to become unacceptably high. Consequently, large scale shared memory systems use a so-called Non-Uniform Memory Access (NUMA) model where there is a notion of memory locality, whereby a given processor can access certain memory addresses quicker than others. In hardware terms, this is due to the fact that the memory is split into several portions, and certain processors are attached directly to teach. In order for a processor to access memory outside of its local region, it must go through one or more additional buses to reach it. Thus, accessing remote memory is slower than accessing local memory.

Distributed Memory differs by the lack of a common address space between all processors in the parallel system. Removing the requirement for a common address space allows the avoidance of problems associated with large-scale Shared Memory systems, such as maintaining acceptable latency for accessing local memory, and implementing cache coherency. This allows for scaling that is limited only by the interconnect and the ability of the system's applications to make use of the large number of processors available. While the largest Shared Memory systems only allow a few thousand processors, distributed memory machines with over a million processors have been built, and consequently these machines dominate the Top 500 list of fastest supercomputers. The downside of Distributed Memory systems is that processors cannot share data without the use of explicit transfers over the network initiated by software.

Generally, Shared Memory architectures are associated with a threaded (or fork/join) programming model, and Distributed Memory architectures are associated with the Single Program, Multiple Data (SPMD) execution model, and either a message passing or single-sided data transfer models.

In a threaded model, the parallel program is started as a single process, which spawns multiple threads of execution, which are executed asynchronously (i.e. there are no guarantees that they will progress towards completion at the same rate), and scheduled on the available processors by the operating system. As the threads are spawned within the context of a single process, they all share the same address space, and thus data can be passed between them through the use of pointers, or global data structures. This leads to low overhead of execution. Thus the threaded model describes both the means by which code is executed in parallel, and how data is shared between the processors executing the parallel programming. The disadvantages of this model are that the asynchronous nature of exectuion, and the visibility of the same memory to all processors can lead to race conditions, and that there tend not to be any provisions for modelling locality of access in NUMA systems.

Due to the fact that Distributed Memory machines do not share an address space (and consequently do not share other things such as operating system instances) it is not possible to have a single process controlling multiple threads of execution. Instead, in the SPMD model, a process is started on each processor taking part in the execution of the parallel program. As with threaded programming, each process runs asynchronously, unless explicit attempts are made to synchronize them. Each process runs a full copy of the program, which requires a fundamentally different approach to software design compared with the threaded model - in the threaded model, the process starts out with a single thread which at some point spawns more. Thus, at least part of a threaded program is serial, whereas in SPMD the program is entirely parallel. This means that the code must be written in a way to make sure each processor does different work. This is usually accomplished by using the process's ID to control the operations it carries out.

The SPMD model does not state anything about how the programs communicate with each other. There are two models - the first is message passing, which is the commonly used method in HPC codes, primarily in the form of the standard Message Passing Interface (MPI) API. The message passing model views the transfer of data between two processors as a send-and-receive operation, where both the sender and receiver need to involved in the data transfer at the application level. This model has several advantages - it neatly maps on to the semantics of most computer networking protocols. If synchronous messages are used, there is implicit synchronization between the two processes, which minimizes the chances of creating race conditions in the code. The main disadvantage is that the requirement for both processors to be actively involved in the data transfer makes it difficult to model certain data transferral patterns, particularly in the case where the transfer patterns are only known at runtime (such as the case where dynamic load balancing is being carried out). MPI implementations also face certain issues - such as matching up corresponding send/receive pairs when there are many messages being sent around the network at once, and the reliance on buffering messages until the receiving process is ready to receive them, which is likely to be problematic as the amount of memory per core decreases on future machines due to power and cost limitations. There is also a reliance on internal data structures whose size grows proportionate to the number of processes available, in particular, MPI communication groups require each processor in a group to have local information on all the other processors in the group.[12]

The other option is to use single sided transfers, where one processor can place data into a remote processor's memory, or copy a piece of remote data into its local memory. This allows for more flexible code at the expense of requiring more explicit synchronization operations to prevent race conditions from occurring. The performance of single sided transfers depends on how well their semantics map on to certain types of network better than others. They originated on special HPC machines where the interconnects were capable of Remote Direct Memory Access (RDMA) operations, where I/O devices (such as the interconnect controller in this context) could copy data from local memory straight into remote memory without the operating systems at either end being involved. On interconnects that do not support this, this has to be emulated on top of a conventional two-sided messaging model, and thus the performance benefits are negated (however, the programmer can still program in a single-sided manner, which may make it easier for them to implement their application.) However, the availability of RDMA support in both Infiniband and ten-gigabit Ethernet (10GBE) means that this sort of capability is more commonly available than it was before.

The prevalence of multi-core processors has led to HPC machines that feature both Shared Memory parallelism, and Distributed Memory parallelism, and are sometimes called Shared Memory clusters. It has become common to make use of a hybrid model for programming these systems, where threads are used within nodes, and message passing is used across nodes. This can make the code quite complicated due to the mixture of different models that were not designed to interoperate, and is one of the main motivations behind the PGAS model - to provide a single model for all situations.

## 2.2 Coarray Fortran

*"We designed Co-Array Fortran to answer the question 'What is the smallest change required to convert Fortran 95 into a robust, efficient parallel language?'. Our answer is a simple syntactic extension to Fortran 95. It looks and feels like Fortran and requires Fortran programmers to learn only a few new rules"* - Numrich and Reid.

Coarray Fortran (CAF) originated as an extension to Fortran (specifically the Fortran 95 standard) but was later incorporated into the Fortran 2008 as an official part of the language. It uses the SPMD model, where each copy of the program (called an image) is executed asynchronously. Each image has its own set of variables and data structures which independent of those on other images, even if they share the same name.[9]

Data sharing between images is carried by use of Coarrays, a data structure that behaves like a standard Fortran array, except that the programmer is allowed to access another image's copy of the given Coarray by specifying the remote image's identifier.

A Coarray is required to have the same dimensions and type on each image in the Coarray program. If the programmer wishes to have Coarrarys of different size on different images, they are permitted to have Coarrays of pointers to local standard arrays.

Coarrays can make use of Fortran array syntax. Certain implementations may optimize Coarray expressions with array syntax to make use the bulk or strided transfers of the underlying network library.

All synchronization between images must be carried out explicitly by the programmer, and the language specification makes no guarantees about timing between synchronization points - this is to simplify optimization of the code by the compiler. Coarray Fortran provides several synchronization intrinsics - the most fundamental one is sync_memory, which is used to flush changes to remote processors, and to retrieve any changes made to the local data from memory. sync_images is used to synchronize a set of specified images, and sync_all is used to synchronize all images in the program.

The language also provides intrinsics for finding the current image number, the number of images in the program, and for synchronizing file I/O between images.

## 2.3 Unified Parallel C

Unified Parallel C (UPC) is an extension to C99 which originated at the University of California Berkeley and the Lawrence Berkeley National Laboratory. Like Coarray Fortran, it uses the SPMD programming model (it uses the term ‘threads’ instead of ‘images’, even though the underlying implementation may not use threads to implement the parallel execution).[8]

As with Coarray Fortran, data sharing between threads is accomplished by declaring certain variables and data structures to be shared. However, the way in which the sharing takes place differs from that in Coarray Fortran:

- When declared as shared, scalar variables are only allocated on the first thread. In Coarray Fortran, they exist on each image of the program.
- A shared array in UPC has its elements split across all images in a round robin manner. This is in contrast with Coarray Fortran, where the array is allocated separately on each program image. By default the elements are distributed one at a time, but it is possible for the programmer to specify a ‘blocking factor’ so that that more than one element is distributed at once. It is possible to specify blocking factors such that the array is either located entirely on the first thread, or that the array is distributed in evenly sized contiguous blocks. If the latter is used to split an array of size N by the number of threads, for some N that is a multiple of the number of threads, it is equivalent to declaring a Coarray of size N in Coarray Fortran.

As such, UPC's arrays provide a global view of the distributed data which provides an abstraction over the distribution of the data, whereas CAF's Coarrays require the programmer to manually distribute the data across images. While UPC arrays are potentially easier to use, they also make it easier for the programmer to write code that performs poorly due to excessive remote memory accesses.

In order to minimize the number of remote accesses, UPC provides a special for loop construct (upc_forall) where operations on a given element are only carried out by the thread where that element is local to.

UPC also provides barrier and locking facilities, as well as a split phase barrier, where the barrier is split into a notify and wait phase. This allows threads to notify eachother and carry out further computation before waiting. If given enough work to do between the two stages, it is effectively possible to ensure that no thread will ever wait, thus allowing non blocking synchronization.

UPC also provides special pointers that allow data on remote threads to be accessed and dereferenced.

## 2.4 Asynchronous PGAS languages - X10

A separate category of PGAS languages, named asynchronous PGAS (APGAS) languages have also been developed. These are also known as High Productivity Computing Systems (HPCS) languages, name for a DARPA project and tender to build a next generation supercomputing system at Oak Ridge National Laboratory, for which the APGAS languages were developed. The best known APGAS languages are X10 from IBM, and Chapel from Cray. Another language that is often mentioned in this context is Fortress from Oracle (originally Sun Microsystems) but this language is no longer developed.[4]

The PGAS languages described so far use the SPMD execution model. This model is familiar to those who have worked with legacy HPC programming libraries such as MPI, but imposes certain problems - for one it tends to assume that there is one process per available processor in the system, and does not elegantly handle the situation where parallelism is dynamically added or removed from the program. It assumes that all processes in the program are heterogenous, and does not handle the case where different processes are running on different hardware (and if it did, the languages would have no features to allow the programmer to reason about the differences). Finally, while the langauges (to varying extents) support data parallelism, and some global notion of distributed data, there is no global view of the flow of the program, which can make task parallelism difficult to implement.

APGAS languages differ by offering a programming model that is more similar to the fork-join model commonly associated with shared memory programming (in practise, the runtime may be implemented in an SPMD manner, but this is hidden from the programmer). The APGAS languages are based around two concepts that are represented as first class objects in the language - sites of execution, and tasks that can be executed on them. Instead of writing code that is executed on all processes, an APGAS contains a description from a high level viewpoint of how tasks are mapped onto places, and how synchronization is carried out. In the X10 languages, the tasks are called 'asyncs' and the execution sites are called 'places'. When an 'async' is assigned to a place, it is carried out asynchronously with respect to the place where the task assignment was carried out. The programmer may choose to wait on its completion with the finish keyword, or specify that the task is to be carried out in an atomic manner.

In constrast with the conventional PGAS languages which generally assume one thread per process, an APGAS site of execution may contain one or more threads, allowing for multiple asynchronous tasks to be executed at once.

There is no requirement for the places to be of homogeneous type, and the developers of both X10 and Chapel have explored different types of place to represent accelerators, and have proposed hierarchical places to represent execution sites with complicated memory or execution structures (such as the Cell Broadband Engine processor.)

## 2.5 OpenSHMEM

SHMEM (Symmetric Hierarchical MEMory) is a library for C, C++ and Fortran which allows the programmer to carrying out single-sided remote memory accesses. Whereas MPI uses a two-sided send-receive model of communication which features implicit synchronization, SHMEM allows the transfer to or from remote memory to be carried out by the calling processor without the involvement of the remote one. This is useful where the implicit synchronization of MPI is unrequired, or in situations where two-sided communcations make the given algorithm difficult to express (specifically algorithms that require load balancing or dynamic distribution of workload, and thus the communication patterns are not known in advance.) This comes at the cost of having to explicitly synchronize communications to prevent race conditions from happening.[20]

SHMEM was originally developed by Cray for their T3D architecture, and was further developed by SGI when they acquired Cray and incorporated SHMEM into their Message Passing Toolkit. As it was a proprietary tool designed specifically for Cray and SGI hardware, other vendors and academic institutions developed their own implementations. OpenSHMEM was created as an open definition of the SHMEM interface by a group of vendors and users, and is based on the SGI SHMEM implementation.

SHMEM implements many of the PGAS constructs seen in the previously described languages - it uses an SPMD programming model where each processing element (PEs - equivalent to the CAF image or UPC thread) has a set of private and shared variables. It differs from UPC and CAF by the fact that it does not modify or extend the underlying language, it is simply a library for standard versions of the aforementioned languages that it supports. In this manner, it does not require any special tools, but the compiler is unaware of what the library is doing, and thus cannot optimize the programmer's use of it. It also uses function calls to transfer data, and thus accessing remote data requires more programmer effort than accessing local data, which is something that the dedicated PGAS languages have strived to avoid.

As with CAF and UPC, certain rules exist about the sort of data that can be shared under SHMEM. Data that lives in the global or static sections of program memory can be shared, as the addresses that these items will occupy in memory is known at compile time. Variables residing in either the stack or regular program heap cannot be shared. However, SHMEM provides special versions of the standard C memory allocation functions that allow data to be allocated on a 'symmetric heap' which is a portion of the program heap which the SHMEM library allows transfers in and out of. Certain rules exist regarding allocation on the symmetric heap - any time memory is allocated from the symmetric heap, it must be done by all processing elements at the same time, and they are all required to allocate the same amount of memory. Similarly, if a block of memory is freed from the symmetric heap, it must be done by all the PEs at the same time. The specification for SHMEM states that failure to follow these rules results in undefined behaviour. One consequence of these restrictions is that a given data item will have the same address on the symmetric heap of each processing element.

SHMEM provides a variety of barrier and synchronization functions. There are barrier functions equivalent to the sync_all and sync_images from CAF. There are wait and wait_until functions which cause the calling processing element to stall until a specified shared memory location on a specified processing element meets a certain condition (which is specified by the programmer.) The is a fence function which guarantees that all writes to remote memory locations will be completed before those initiated after the fence (but it does not guarantee when they will be completed.) The quiet function does something similar, except that it forces the completion of all outstanding writes to remote processors (this differs from a barrier in that it is not a collective operation, and thus there are no guarantees that outstanding writes to the calling processing element have been completed by the end of the function call.) As with other PGAS languages, SHMEM also provides collective and locking functions, and like UPC, it supports atomic operations on remote memory, with the condition that the operations are only guaranteed to be atomic if the targeted memory locations are only written to using atomic operations. If there is a mixture of non atomic and atomic operations used to write to the memory location at the same time, the atomicity is not guaranteed.[13]

## 2.6 GASNet

The PGAS languages are dependent on a high performance interconnect, preferably one capable of remote memory accesses. Many of the runtimes and libraries are built on top of GASNet (Global Address Space Networking), a library that supports single-sided semantics on top of a variety of different network types (referred to as 'conduits'). The library consists of two parts, the GASNet Core API, which is implemented directly on top of the underlying conduit API, and the GASNet Extended API, which is implemented purely in terms of the Core API. This allows new networks to be supported by implementing the Core API on top of the network, which is similar to the Abstract Device Interfaces used by MPICH2. Some of the GASnet conduits available include shared memory, MPI (which has poor performance, but is highly portable) various Infiniband implementations, as well as the proprietary IBM and Cray networks - including the Gemini network used by HECToR and the Aries network that will be used by ARCHER. However, on the Cray systems, many of the PGAS libraries and runtimes (including Cray SHMEM) are implemented on top of their proprietary Distributed Shared Memory Application (DMAPP) API which interacts with the hardware directly[7]. The GASNet Extended API is based around a notion of Active Messages - messages which execute when they are received instead of simply carrying data. This interface is quite low level, and consequently is intended for compiler writers and library implementers as opposed to application developers.[18]

# Chapter 3

# Background - Relevant features of the C++ language

## 3.1 C++ Templates

Templates are a mechanism in C++ that facilitate generic programming, a style of programming where algorithms, data structures and/or libraries are written in a manner that does not tie their use to a particular type of data.[16] Consider the following C++ class declaration which represents an array of integers:

```
class Array
{
private:
int * startOfArray;
size_t lengthOfArray;
public:
Array(size_t length);
~Array();
int & operator[] (size_t index);
};
```

Note that in this declaration the behaviour of C++'s array subscription operator (i.e. []) has been overloaded. This means that if the [] operator is applied to an instance of this object, it calls the method defined by the programmer. In this context, that method would be as follows:

```
int & operator[] (size_t index)
{
return startOfArray[index];
}
```

In absence of this overloading, the application of the [] operator to an instance of this array would be undefined, and would result in a type error at compile time if the programmer attempted to use this in their code.

It should be reasonably clear that the only difference between this declaration, and an equivalent declaration of an array of double-precision floating-point values is that all instances of 'int' would be replaced with 'double' (this would also apply to the code that implements the class' methods). Instead of duplicating the code for each type, C++ templates can be used which allow the data structure to be described in terms of a type that will be declared later.

```
template <typename T>
class Array
{
private:
T * startOfArray;
size_t lengthOfArray;
public:
Array(size_t length);
~Array();
T operator[] (size_t index);
};
```

When instantiating an instance of this class, the programmer would use the following statement to create an array of ten integers:

```
Array<int> array(10);
```

The type is said to be parameterized, which means that it is effectively passed as an argument to the template-using code. This is carried out at compile time, and the compiler effectively creates an internal class which looks like the previous declaration with the hard coded type. It is also possible to have a template function which is not a member of a class, for example:

```
<typename T>
void sort(T * array, int length);
```

It is also possible to alter the behaviour or even the entire definition of a template class for a manually specified type. To give a somewhat contrived example, imagine if a programmer wanted to specify that for character arrays, the class should have a method that returns a string, but not have this behaviour available for other types. The behaviour for type 'char' would be defined as follows.

```
class Array<char>
{
private:
```

```
T * startOfArray;
size_t lengthOfArray;
public:
Array(size_t length);
Array();
T operator[] (size_t index);
string toString();
};
```

This is called template specialization. No code is shared between specialized templates and the non specialized class.

## 3.2 Arrays in C and C++

In considering the design of a library that mimics the behaviour of arrays in C and C++, it is useful to consider the semantics and behaviour of the native arrays of those languages. First, consider the most basic case - a single dimensional array in C (and C++). A single dimensional C array consists of a contiguous block of bytes in memory. The array is represented as a pointer to the first byte of the array, and the array subscript operator (i.e. []) is an operation which adds the specified index to the base address to get the address of the desired item. The portion of program memory in which the array is stored is dependent on how the array is allocated - it can either be allocated in the stack frame of the function that declares it, on the heap through the use of a library call, or in the read-only static section of memory if the contents of the array are initialized at run time. For simple single dimensional arrays, these largely behave the same except for the statically initialized arrays whose contents cannot be changed at run time.[10]

Multi-dimensional arrays are slightly more subtle. When dealing with statically initialized and statically declared arrays, the arrays are laid out in row-major order in memory - in other words, if we have a 4x4 array of a single-byte type, the compiler will create space for sixteen bytes - the first four of which will represent the first row, the next four the second row and so on. In effect, the two-dimensional array is an array of rows. This pattern can be applied recursively to deal with N dimensional arrays, which is treated as an array of N-1 dimensional arrays, which are treated as an array of N-2 dimensional arrays and so on. Ultimately, for any number of dimensions, the array is represented in memory as an array of rows. When such an array is indexed, the compiler generates object code which indexes into the array from the address of the first element using information about the extents of the dimensions that are known at compile time.

In order to index into such an array, it stands to reason that we need to know about the extents of the array's dimensions, as well as the index. Consider the previous 4x4 array. If we wish to access the 2nd element of the 3rd row, we would need to know where the 3rd row starts, which we can get by multiplying the number of elements per row by three. We would then add two to get the starting addressing of element [3, 2]. In more general terms, a two dimensional array is indexed by:

```
(Row index * length of a row) + column index
```

As mentioned above, this process can be generalized to n-dimensional arrays by treating an N dimensional array as an array of N-1 dimensional arrays, and thus applying this formula recursively till it becomes a one dimensional array.

The ability to index arrays using the above method is dependent on the compiler knowing the extents of the array dimensions at compile time. In C, this information in encoded in the type of the array, that is to say if declare an array foo as follows -

```
int foo[4][4]
```

Foo's type is

```
int[4][4]
```

An a pointer named bar to foo would be declared as -

```
int (*bar)[4][4] = &foo;
```

An interesting consequence of this is that a pointer to an array of size can be used in C and C++ to treat a block of memory allocated on the heap like a statically defined array.

```
int (*foo)[4][4] = (int (*)[4][4])malloc(4 * 4 * sizeof(int))
```

At no point does the compiler need to know the extent of the inner-most dimension (i.e. The number of columns per row). This means that the extent of the first dimension can be left out to provide a degree of flexibility, for example, given an integer variable x equal to four, the following statement is equivalent to the previous one (in that it creates a 4x4 array)

```
int (*foo)[4][] = (int (*)[4][])malloc(4 * x * sizeof(int))
```

If the extents of the outer dimensions of the array are not known at compile time, neither C++ nor ANSI C provide any mechanism to allow compiler-assisted indexing into a multidimensional array. The standard method to create arrays of arrays (or arrays of arrays of arrays and so on for greater numbers of dimensions). This is accomplished with code as follows:

```
int * foo = (int *)malloc(sizeof(int *) * 4);
for (int i = 0; i < 4; i++)
{
foo[i] = malloc(sizeof(int) * 4);
}
```

The above implementation is not ideal as there are no guarantees that all the rows are located contiguously in memory, which is bad for cache performance. It also means that several free operations are required to deallocate the array. This can be solved by allocating all the required memory at once, and iterating over the resulting memory block.

Even doing this, the requirement to have arrays of pointers causes several problems:

As a certain portion of the allocated memory is required for pointers, there is a storage overhead associated with arrays of arrays, that increases exponentially with the number of dimensions.

The address of the start of the array of arrays is pointer, not a data element, in other words `&foo` is not the same as `&foo[0][0]`. This means the arrays are not interchangeable with statically defined arrays.

C99 introduces so called Variable Length Arrays (VLAs) which allows the declaration of statically-declared arrays whose extents are not known at compile time.[11] In other words, it is possible to declare the following for variables x and y whose values are set at run time:

```
int foo[x][y];
```

As statically defined arrays are allocated on the stack frame of the current function call, the compiler implements this construct by resizing the size of the stack frame at run time, an operation equivalent to the C standard library `alloca` function. Use of VLAs (and similarly use of alloca) is considered by some to be bad practise as it can lead to stack overflows during the middle of a function call. However, it is possible to create pointers with VLA types, which allows multidimensional arrays whose size is not known at compile time to be allocated on the heap while retaining the ability to index into the array without having arrays of arrays:

```
int (*foo)[x][y] = (int (*)[x][y])malloc(sizeof(int) * x *
y);
```

These pointers can be passed to functions as long as the extents are declared in the function parameter list before the pointer:

```
void some_function(int x, int
y, int (*array)[x][y]); //Allowed
void some_other_function(int (*array)[x][y], int x, int y);
//Will not compile
```

One advantage of arrays of arrays over pointers to VLA types is that arrays of arrays allow the programmer to create arrays where the number of dimensions is not known at compile time.

VLAs are not supported in the current C++11 standard. There are proposals to support VLAs in the next C++ standard, C++14. However, pointers to VLAs will not be supported, nor is it possible to apply the sizeof operator.

# Chapter 4

# Implementation

## 4.1 Array Semantics

To create a library that implements the Coarray model, we need to provide a data structure that lets the programmer specify which image's copy of the array they wish to access. It needs to behave consistently irrespective of whether the programmer accesses local data, or remote data. It must also provide synchronization routines as well.

The issue arises of how to implement array indexing in a C++ library, knowing that the same method must be suitable for indexing both a local array, or one that is located on a remote processor. Indexing a local array is a relatively straightforward procedure, so the choice should be made around accommodating the remote case.

C++ allows the programmer to overload the array subscript operator. The operator is represented as a function that takes a single argument (usually an unsigned integer type) and returns a C++-style reference to the item that has been indexed, for example, for a class MyArray that represents an array of integers, the programmer may overload the subscript operator by declaring (and implementing) the following function:

```
int& MyArray::operator[] (size_t index);
```

There are two things to note here - first is that the function returns a C++ reference instead of returning the item by value, or returning a pointer. A C++ reference (which is declared like a pointer, but using & instead of *) is a named alias for a piece of data (somewhat similar to Fortran's notion of pointers). It is not possible to apply pointer arithmetic to a reference, nor is it possible to have a pointer or a reference to a reference (as the reference does not exist as a separate entity in memory.) It is also not possible to dereference a reference, instead, all operations carried out on the reference act as if they are carried out on the original variable. In other words, the following code:

```
 int x = 5; //Standard scalar variable
int & y = x; //Declare a reference y which refers to x
```

```
y = y + 1;
cout « x « endl;
```

Will print ‘6’, as any time y is used, it is treated as if x has been used. The purpose of returning a reference from the subscript operator is to allow the subscripted object to be used on both the left and right hand side of an assignment statement (in C++ parlance, it can be used as an ‘lvalue’ or an ‘rvalue’[15]). Consider the below code where x is an instance of the previously described MyArray class which holds at least two elements:

```
x[0] = 3;
int y = x[1];
```

The above code would not work if the operator returned a value or a pointer.

However, there is a problem with this system - there is no notion of overloading multiple array subscripts that have been chained together. In other words, we can’t define a single function that handles the following case:

```
x[4][3] = 9;
```

Ultimately, the subscript operator is designed with an ‘array of arrays’ mentality - in order to make the above statement work, we would have to break the subscription of x into two - first we have x[4] which returns an intermediate object (or a pointer) to which we apply [3], giving a reference to the desired memory location.

Assume that the process of accessing data on remote processors has been solved. There are several ways in which we might decide to index an array located on a remote processor. The most preferable way would be that we could specify the indices using the standard C/C++ syntax, and that the local object would calculate the address of the remote data item, fetch that item, and return it to the user. For example, the statement -

```
x[2][1] = 5;
```

Would first create a object that represents the third row of the array, to which application of the [] operator would return the second element of that row. This allows the library to use the familiar array indexing syntax, but requires the creation of intermediate objects, which is potentially wasteful of memory of processing effort. Nonetheless, the library uses this approach so that the Coarray behaves like a normal array as much as is possible

Another option which is taken by certain libraries which implement multidimensional array objects, including the Coarray on C++ implementation described in[5], is to simply abandon any attempt to create an object that behaves like a standard C++ array, and devise separate syntax for array indexing which is more convenient for the implementer. One possible way of doing this is to overload the function call operator (i.e. () ) and use it to pass a series of indices (using either C-style varags, or C++11-style parameter packs, which will be discussed later) calculate the location of the specified item in memory, and return an index to it. For example -

```
(1, 2, 3) = 5;
x(1, 1, 1) = 4;
```

(One consequence of this approach which is interesting in the context of this project is that it mimics the array indexing syntax of Fortran, which has the advantage of familiarity for Fortran programmers, but the disadvantage of unfamiliarity for the C++ programmers who actually have to use it...)

Given the choice of array indexing, we can think of the basic array object semantics for use in the library as a constructor that takes a list of extents for each dimension and allocates a contiguous block of memory which can hold all these items, the array indexing operator, and a destructor which frees the allocated memory block. The type used to define a three dimensional integer array would be -

```
Array<int, 3>
```

While the number of dimensions could be inferred from the number of indices passed to the constructor, having the the number of dimensions as part of the type means that this information is available at compile time, and that it is possible to carry out certain desirable compile time checks, such as ensuring that the programmer does not pass too many or too few indices when the array is indexed.

## 4.2 Handling Remote Arrays and Elements

To create an object that behaves like a CAF coarray, we define an object that behaves like the above array, but differs first by allocating the memory on the SHMEM symmetric heap. It must also provide means of specify which SHMEM processing element's address space we wish to index. This is facilitated by overloading the function call operator as a means of specifying the PE index required (this function will be referred to as the 'PE selector' operation). While the programmer is free to specify the index of the local processor, if they omit the process selector, it is implicitly assumed that they are referring to the local array.

```
Coarray<int, 2> x(4, 4); //Declare a 4x4 integer Coarray
x[2][2] = 5; //Update index [2, 2] on the local array
x(4)[2][2] = 4; //Update index [2, 2] on the array of PE 4
```

In the case where we use option 1, the array behaves as normal when dealing with the local array. When specifying a remote array, the PE selector returns a RemoteArray object. This is an object that encapsulates the address of the start of the array, and has a single method - the overloaded () operator which serves as the array subscript operator. When the array is indexed, the subscript operator returns a RemoteReference object. This object encapsulates the address of the remote data that is to be accessed, and is intended to behave like a standard C++ reference, except that it refers to a value on a remote processing element.

In order to simulate the behaviour of a reference, we need to overload two operators. The first is the assignment operator, so the reference can be used as a C++11 lvalue. The assignment operator takes the value to be assigned as an argument, and returns the same value at the end of the function (in C and C++, assignment statements are treated as expressions that evaluate to the new value of the assigned variable. This allows, amongst other things, assignment statements to be chained, e.g. x = y = 5). The remote reference uses the shmem_put function to write the value to the remote data item.

In order to let the remote reference behave as an rvalue (in other words, so that we can use it in an expression, and have it evaluate to the value of the remote data) we rely on the implicit conversion operator of C++. This operator, specified for a certain type, tells the compiler how the object can be converted to a specified type (or perhaps more descriptively, how the object behaves when located in an expression that expects the object to be of the specified type.) For example, if the programmer defines a class named Integer that wraps around an int, and provides certain methods, they might want to be able to use instances of that class in places that expect a regular int. This is possible in C++ if the Integer class has overloaded the implicit conversion operator for int. The programmer may implement a method as follows:

```
int Integer::operator int()
{
//Where value is class attribute that stores the wrapped integer
return value;
}
```

This allows the following code to work:

```
 Integer x(5);
int y = 4;
//x behaves like an int in this context.
int z = x + y;
```

Without overloading the int() operator, this code would fail with a type error, as x is an Integer and not an int.

Using this operator, we can allow the remote reference to fetch the remote data in the background by overloading the implicit conversion operator for the type of the Coarray that we are accessing (this type information is passed by templates from the Coarray to the RemoteArray down to the RemoteReference) to carry out a shmem_get on the reference's memory address, and returning the value that it has returned.

For option 2, the array behaves in much the same manner as before, but when referring to a remote array, there are RemoteArrayIndex objects that are otherwise identical to the regular ArrayIndex objects, except that they store the PE identifier provided to the PE selector operation, and when it reaches the inner-most dimension, it returns a RemoteReference object as described above.

## 4.3 Synchronization

Regarding other intrinsics of Coarray Fortran, certain ones map on directly onto functions of the SHMEM API - for example, the collective barrier for all images, finding the number of images, and finding the image number of the calling image. One important one that does not sit so well is the CAF sync_images. SHMEM provides an barrier for a subset of images, but it is a somewhat awkward function, requiring the programmer to specify the number of processes in the subset, a 'symmetric work array', the first image in the subset of processes to be synchronized, and the stride between the processes we want to synchronize. While this works for situations where the programmer wants to synchronize every second image for example, it doesn't have the same flexibility as CAF's sync_images, which allows the programmer to specify an arbitrary list of processes to be synchronized. The other issue is that it requires the programmer to declare a symmetric work array for the given subset of processors that are about to be synchronized, that the array be initialized with a special value, and the array must be reinitialized if it used to carry out synchronization of different subsets of processors. Ultimately, finding a layer of abstraction over these details did not appear trivial, and it would likely involve the programmer instantiating special wrapper objects, would result in a level of ungainliness that would defeat the point of the abstraction, and so it was left out.[13]

An attempt to create a point-to-point barrier using SHMEM's atomic fetch and increment operations along with the shmem_wait_until operation (which stalls the processing element until a local value at a given address satisifies some condition) to create a semaphore-based barrier. However, as there were no collective atomic operations, the performance of this function was far slower than that of the global barrier, and thus it was removed from the code.

## 4.4 Emulating Fortran Array Syntax

The Fortran 90 standard treats arrays as first class objects in the language, and provides a convenient syntax (so-called Array Syntax) to simplify common array operations. If two arrays have the same type, same number of dimensions, and the same extents, it is possible to write single statements which operate on all elements of the arrays as if they were scalar values.[14] For example, given the arrays A, B, C:

```
A = B + C
```

Adds the corresponding elements in B and C, and assigns the results to the corresponding elements in A.

```
A = B
```

Simply copies all the elements of B into C.

It is also possible to carry out manipulations on subsets of arrays with 'slice' notation. The following statement, for a single-dimensional array X:

```
X(3:  7) = 4
```

Sets elements 3 through 7 to be equal to 4. Given the two dimensional arrays A and B of same type:

```
A(1:2, 1:2) = B(1:2, 1:2)
```

Is equivalent to the following iterative code:

```
do i=1, 2
do j = 1, 2
A(j, i) = B(j, i)
end do
end do
```

Neither C nor C++ have any equivalent construct. Cilk Plus, a extension to C++ for parallel programming developed by Intel has an equivalent syntax which maps onto SIMD vector operations, but this not part of either the C or C++ standards, and is only officially supported on Intel's C++ compiler, and non-standard forks GCC and Clang.[21]

While a full implementation of Fortran's Array Syntax is beyond the scope of this project, a limited subset of this functionality is particularly desirable in the context of this project as not only does it provide convenient syntax for transferring items to and from remote Coarray objects, it allows the library to represent whole-array and slice transfers in terms of one or more SHMEM put/get operations that transfer multiple elements of the array at once. Even without array syntax, the compilers for languages like CAF and UPC can spot instances where the programmer is transferring elements one by one to a remote process in a loop, and optimize it into a single RMA operation. This is important for performance reasons, as any operation across the network has a minimum setup cost and latency, and thus it is possible to send multiple elements in the same time taken to send a single one, especially when dealing with small data types.

The library implements an Array Syntax for the case of assigning between local arrays, local Coarrays, and remote Coarrays, either for whole arrays, or slices of arrays. The whole-array transfer is relatively straightforward. Given a coarray x, the programmer can write the following statement:

```
x = x[1];
```

Which fetches all elements of the Coarray x on processing element 1, and assigns them to the equivalent elements on the local Coarray x. This is implemented as a single SHMEM get operation which copies N * T bytes from the address of the first element of the array, where N is the number of array elements in total, and T is the size in bytes of the datatype stored by the Coarray. The opposite case (i.e. if the order of the assignment was flipped around) is not defined in the library as it would lead to a set of mutually-dependent template classes, which are not allowed in C++ (to be more specific, the definition of the overloaded assignment operator would require the definition of the other class for both classes). This could potentially be solved by redesigning the API somewhat, but there was insufficient time for this.

Array slicing is a somewhat more complicated affair. There was no obvious way of overloading the array subscript operator to take the slice range specifiers, so a 'slice' method on the Coarray object is used instead. For an array of N dimensions, the programmer passes N tuples of two unsigned integers, the first specifies the lower bound for the given dimension, and the second the upper bound of the given dimension. The slice method returns an array slice object, and the assignment is implemented by overloading the assignment operator for the slice type. Keeping with the convention of C and C++ arrays, the last tuple in the parameter list represents the slice of the rows, and the first tuple (if when there are N dimensions for N > 1) represents the slice of the Nth dimension. As with Fortran, the slice is carried out from lower to upper, inclusive of both. However, unlike Fortran, it is not possible to specify a stride length. While this could have been implemented, it would have added further complexity to the code and was left for future work due to time constraints.

The means by which the slices are mapped on to SHMEM RMA operations depends on the number of dimensions in the array. The simple case is when the array has a single dimension. Given the following statement:

```
x.slice(1, 5) = x[1].slice(2, 6);
```

The library simply uses SHMEM get to transfer elements 2 through 6 inclusive of the remote array into the local array, starting at the address of element 1.

The other case that the library covers explicitly is when the array has two dimensions. Consider a 4x4 coarray x:

```
x.slice(1,3,1,3) = x[1].slice(1,3,1,3);
```

An efficient way of representing this operation would be as a strided transfer. Unfortunately, SHMEM's strided transfers are somewhat limited compared with what is possible with MPI - the programmer specifies (for both the source and destination arrays) the number of elements to be copied, and the gap between each consecutive element. It is not possible to specify that a certain number of contiguous elements should be copied before the stride, or to specify multiple stride lengths per array. For the two dimensional case, we can get around the former limitation by treating each row slice as a single data item. Unfortunately, SHMEM does not allow the programmer to specify the size of the individual elements in a strided, and must select from predefined sizes. This does not fit in with the library's focus on generic operations, and so it is not used.

Instead, the slice operation defines all data transfers in terms of transfers of single rows. N dimensional arrays are split into a (conceptual) set of one dimensional arrays, and the library them one by one. In other words, given the 3x3x3 coarray x:

```
x.slice(1,2,1,2,1,2) = x[1].slice(1,2,1,2,1,2);
```

The library treats x as an array of three two-dimensional arrays. It calculates the starting address of the two-dimensional arrays corresponding to each index i in the inclusive range of the outermost array slice with the following calculation: start = (address of first element in the array) + (i) x (number of columns per row) x (number of rows in the 3D array)

It then recursively calls the internal function used by the overloaded assignment operator, passing the address calculated above as the start of a two dimensional array, to which the same recursive operation is applied to, yielding a set of one dimensional rows, which are then sent as if the programmer had specified a slice of a single dimensional array. As consequence, for N dimensional arrays of N > 1, the number of SHMEM transfers required to send a slice is a product of both the number of dimensions beyond two, and the length of their array slices.

The slicing operation is subject to certain limitations, which are not dissimilar to those of the Fortran standard implementation. One that is specific to this library is that is it only possible to copy from one array slice to another. There is no direct equivalent of the following Fortran statement (for arrays A that stores four elements):

```
A = B(3:6)
```

Other limitations are more obvious ones - the elements specified by the stride must be within the extent of the given dimension for that array, and the corresponding stride lengths for the two arrays for a given dimension must be the same (although they are refer to different upper and lower bound elements). The arrays must also have the same number of dimensions and have the same extents (certain modifications to the current code would remove this limit, but it was not implemented in order to avoid introducing additional complexity into the code.)

# Chapter 5

# Benchmarking

In benchmarking the Coarray library, we set out to profile its performance not only for certain patterns of data transfer, but also its performance on different types of hardware, both across shared and distributed memory programming scenarios. The library was benchmarked on HECToR. Its Gemini interconnect has been specially optimized for single-sided RMA operations, and as discussed earlier, its implementation of OpenSHMEM is built on top of their proprietary DMAPP hardware interface. Previous benchmarking efforts for Coarray Fortran served as the basis of some of the tests described here.[2] [3]

To build the library on HECToR, the cray-shmem module must loaded. Version 5.6.5 was used, which was the latest version available at the time of testing. Furthermore, at the time of writing, the Cray C++ compiler does not support the C++11 standard, certain features of which are required to build the library. Thus, the GNU C++ compiler, version 4.7.2 was used to build the library and benchmark code. In order to use the GNU compiler, the default Cray Programming Environment (PrgEnv, version 4.0.46 at time of writing) module which uses the Cray compiler suite must be unloaded, and the corresponding PrgEnv module for the GNU compiler suite is loaded. Once the GNU PrgEnv is loaded, C++ source can be executed using the CC command (as distinct from the cc command used to compile ANSI C and C99). The CC command is a wrapper which wraps the compiler specified by the PrgEnv module, and include the appropriate include and linker flags for the various Cray libraries, including Cray SHMEM, which is compatible with OpenSHMEM. All these requirements are completed with the following three commands, which was placed into a shell script run at login to automate them:

```
module unload PrgEnv-cray/4.0.46
module load PrgEnv-gnu/4.0.46
module load cray-shmem/5.6.5
```

Through the benchmarks, we wish to examine the bandwidth and latency associated with the single sided SHMEM transfers, the cost of synchronization, and the efficiency of block transfers compared with single element transfers.

The process of running an OpenSHMEM program on HECToR is largely similar to that of running an MPI program. A script for the PBS job scheduler is prepared, specifying the expected run time of the job, the total number of processing cores required, the number of processing cores to be allocated from each node, as well as the path and name of the executable, and any command line arguments. The script then executes the job by passing this information to the aprun utility which submits a job to the job queue. The script is executed using the qsub command, and progress of the job on the queue can be monitored with qstat -u $USER, where the -u flag limits jobs listed to those launched by the specified username, and the $USER being a Unix environment variable which contains the name of the user logged in on the shell (or more specifically, the name of the user to whom the current process belongs to). The output of the program is displayed in a file in the same directory, whose name consists of the name of the PBS, '.o' and followed by the job number. By specifying the number of cores to be allocated on each node, it is possible to create distributed and shared memory versions of jobs that use a small number of cores. For example, to create a shared memory ping pong test, the script requests two cores, and specifies that there are two cores per node. To create a distributed memory run, the script requests two cores, but only one core per node, so that the program is run across two separate nodes.

## 5.1 Array Transfer

In this test, arrays of increasing size are transferred by a single node on HECToR to another core on another node. Times are in tenths of microseconds, and arrays use 64-bit double precision floating point values.

First shown for the case where elements are transferred one by one in a loop.

Second when the transfer is carried out with the whole array assignment syntax, thus using a block transfer.

In the iterative transfers, the cost of sending starts low and then increases at a rate proportional to the number of elements sent, this is not entirely unexpected given that each remote assignment corresponds to a single remote get operation. The block transfers are far more efficient, but transferring a single element seems to take an unusually long amount of time, a phenomenom observed across multiple runs, and across some of the other benchmarks. It is unclear what the cause of this is.

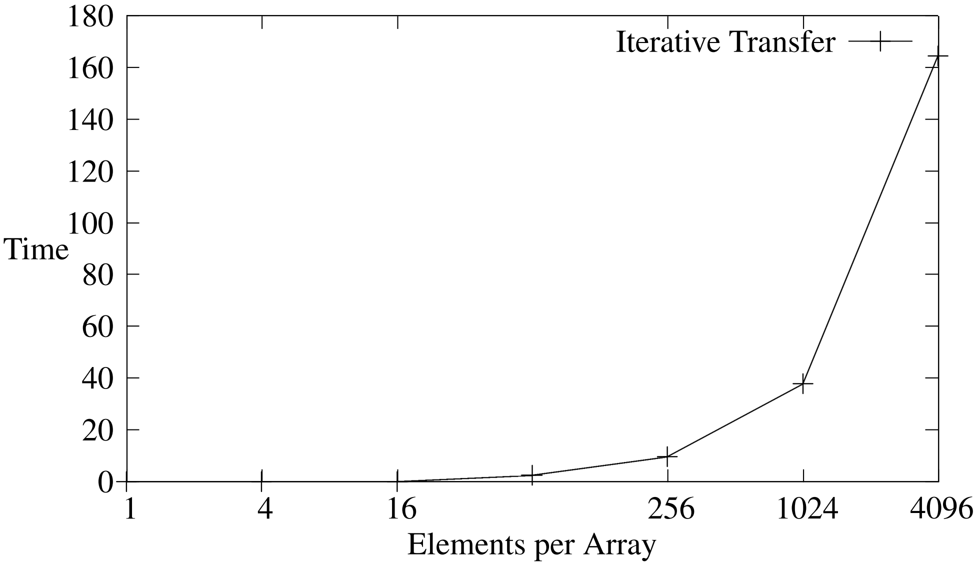

Figure 5.1: Time (microseconds) against Array Size (in doubles) for one-by-one whole array transfer

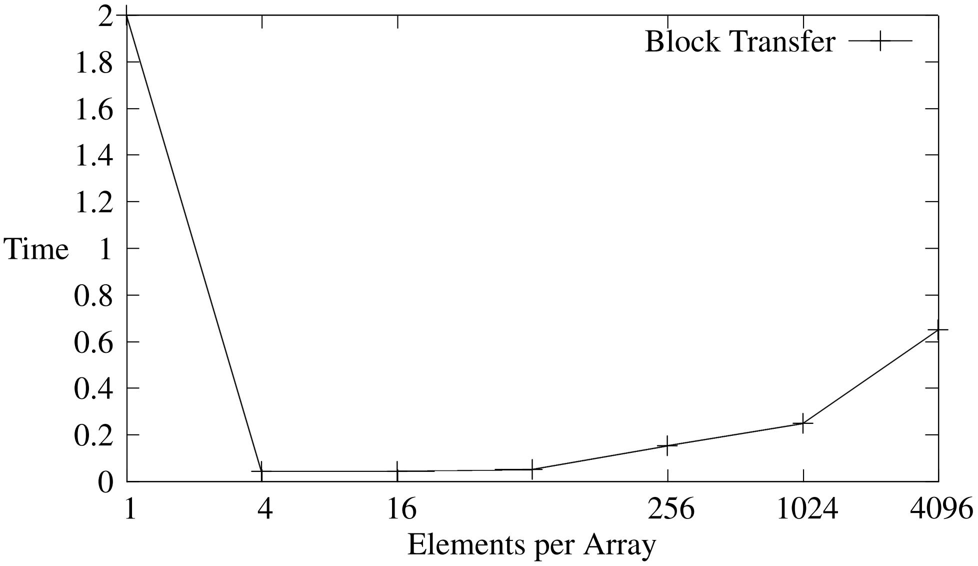

Figure 5.2: Time (microseconds) against Array Size (in doubles) for whole array block transfer

## 5.2 Group array transfer

In this test, arrays of increasing size are transferred by every core on a single node of HECToR to the corresponding core on a second node. Times are in microseconds. For the 4,096 element array, the time taken is much greater for the group of processors compared with a single pair, suggesting saturation of the network.

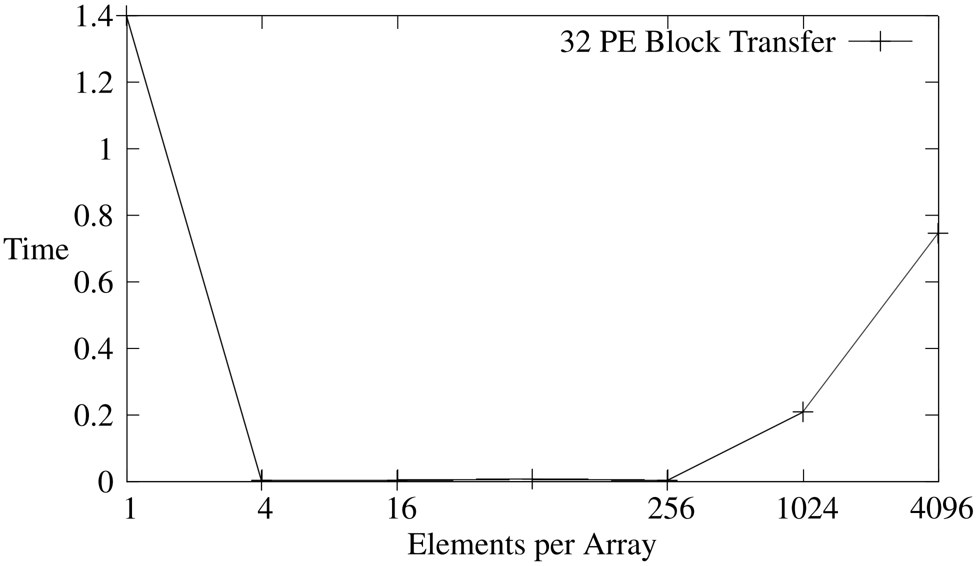

Figure 5.3: Time (microseconds) against Array Size (in doubles) for group array transfer

## 5.3 Synchronization

Global synchronization is shown for HECToR, expanding across four nodes. Times are in microseconds. Synchronization time effectively grows proportionately to the number of processors to be synchronized.

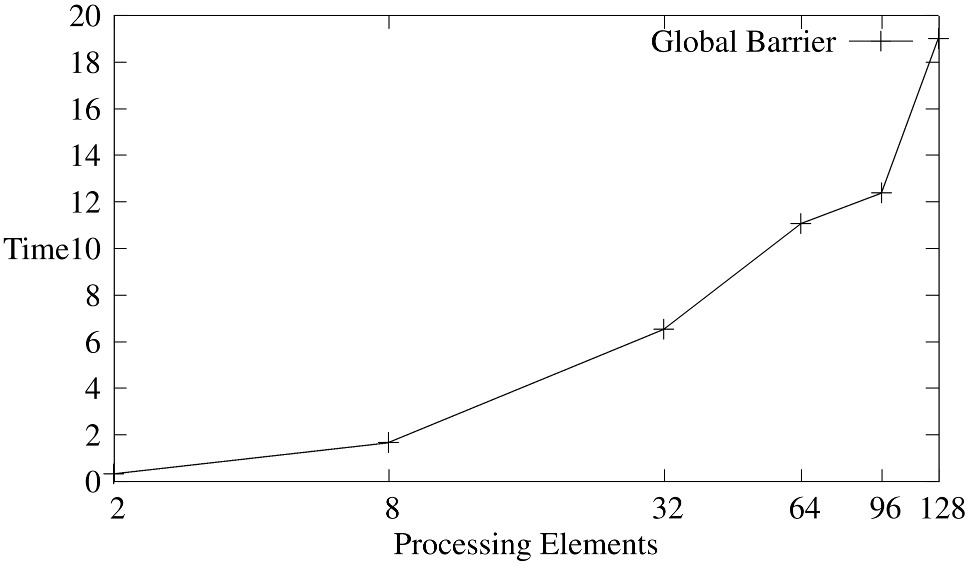

Figure 5.4: Time (microseconds) against Processing Elements for global barrier

## 5.4 Ping Pong

This is similar to the array transfer, except that the arrays are sent back and forth between the two processors, and a barrier takes place at the end of each transfer. The times are shown for the array being bounced back and forth one-thousand times. Times are in seconds, arrays are double precision type as before.

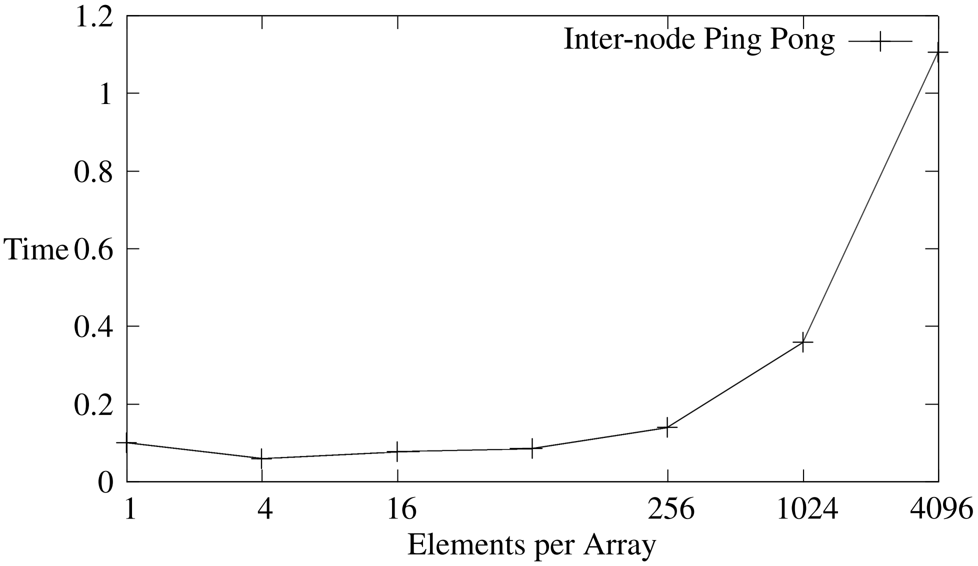

Figure 5.5: Time (seconds) against Array Size (in doubles) for inter-node Ping-Pong

## 5.5 Array Slices

This test shows the performance of transfers of array slices, with times in millionths of a second. The first test tests slicing the first N elements from an array of N x 2 double precision values.

The results show that the slice size remains relatively constant across the size of the blocks, but is higher than the equivalent block transfer times for simple whole-array transfer, this would suggest that the overhead of the slicing operation largely hides the time taken to transfer data.

The second test is for an (N x 2) x (N x 2) array, taking an N x N slice starting from the first element of the array.

Despite a levelling at 32 elements, the lenght of time taken per 2D array slice transfer grows relative to the size of the slice. This is unsurprising given that the recursive method used to transfer multidimensional arrays requires an amount of transfers proprotional to the number of rows transferred. An even greater rate of increase would be seen for arrays with more dimensions

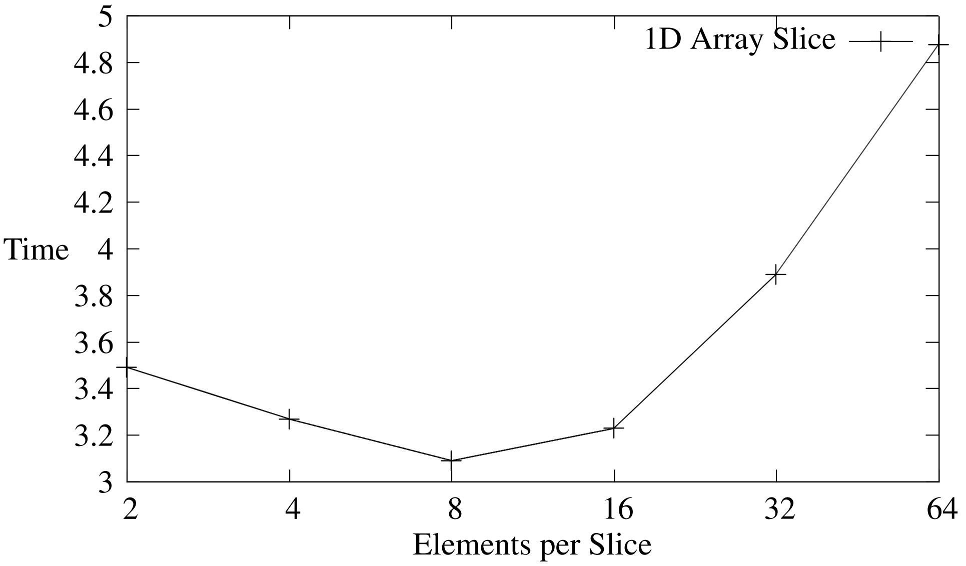

Figure 5.6: Time (microseconds) against Array Size (in doubles) for 1D slice transfer

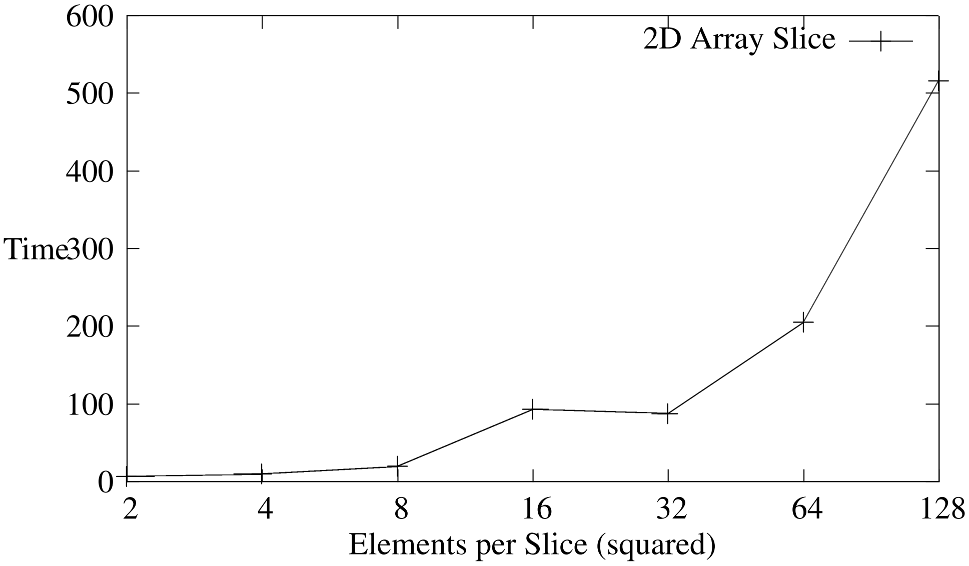

Figure 5.7: Time (microseconds) against Array Size (in doubles) for 2D slice transfer

# Chapter 6

# Conclusions and Future Work

The project has been successful in creating a library written in C++ that provides many of the features of Coarray Fortran without modifying or extending the language, or altering the compiler or any associated tools. In addition, the library provides a straightforward syntax for manipulating data in the Coaray and transferring it between processors without straying far from established conventions for C++ data structures.

Certain conclusions can be drawn from the experience. First is the importance of block transfers of data between processors, instead of sending the elements one by one. Since the purpose and semantics of the library are unknown to the compiler, it is necessary for the library to provide a means of allowing the programmer to express block transfers of data with a convenient syntax. Fortran's array syntax represents an ideal solution, and thus certain aspects of this were emulated in the library - namely whole array assignment, and slices. As can be seen from the benchmark results, use of theses features would be essential from a performance perspective (as well as saving programmer effort). Additional features of Fortran array syntax could be emulated through use of operator overloading. A local non-Coarray version of the library was written for testing purposes, and while not a high priority for this project, would be useful for having local arrays that had the same semantics of the Coarray library, and could even interoperate, for example, allowing data to be moved from a local array to a remote coarray. One issue with the implementation of the library is that the Array type is not compatible with standard C++ arrays due to the way that it is stored and indexed. However, this is not without precedent, there are many other array type classes, for scientific computing, and otherwise, whose implementation and behaviour diverges from the regular C++ array in order to get around limitations of the standard array or to make it more suitable for certain applications.

Another issue is the choice of the OpenSHMEM library as an underlying network transfer protocol. On the surface, it provided many of the basic operations required for the basic functionality of the library, such as collective memory allocation, single-sided transfers of single pieces of data, block transfers and global synchronization. However, for certain more advanced features, the interface and semantics of these operations were quite awkward and others were not pliable for use in a generic programming language context. The non-global barrier had a rather awkward interface for specifying the processors to be blocked, and had requirements for a lock array that would have been hard to hide behind layers of abstraction. The broadcast operations only dealt with 32-bit and 64-bit values. The strided transfers only allowed a single stride length, only allow the programmer to send a single element between strides, and only allows the programmer to send elements of predefined sizes (if this restriction did not exist, it would be possible to get around the second restriction as well. Some of these restrictions, particularly ones regarding fixed element sizes for certain operations, and not for others, seem rather arbitrary, and feel like they are throwbacks to quirks of earlier implementations and the hardware they supported. These issues may be addressed on future implementations of OpenSHMEM. However, perhaps a more fool-proof method of solving these problems would be to use GASNet directly as the network library, and implement the Coarray code in terms of GASNet operations. This would give the implementer greater flexibility with the sort of networking semantics that they wished to use, but would come at the cost of a more complex implementation due to the lower level nature of GASNet calls versus OpenSHMEM ones. Because GASNet is not restricted to the same symmetric heap memory model as OpenSHMEM, it would be possible to implement a more flexible memory model for the Coarray library, allowing Coarrays of pointers to remote data, or allowing Coarrays to have different sizes on different arrays. This would come at the cost of greater setup costs for Coarrays due to the fact that the addresses of the remote data would need to be broadcast across all processors (one desirable property of OpenSHMEM is that the symmetric heap addresses of data items are the same across all processors as long as the code does not engaged in undefined behaviour). Nonetheless, given time constraints, the use of SHMEM allowed a library that contained essential features to be created rapidly and thus was the correct choice in this project.

The flexibility of the C++ language facilitated the creation of a library that behaved similarly to a standard array through operator overloading. The object-orientated features allowed an abstraction over the relatively low level semantics of OpenSHMEM, and data hiding allowed the various parameters needed for the OpenSHMEM operations to be hidden from the user.

Due to time constraints, only relatively simple benchmark code was created. Given more time, benchmarking would test the performance of the library for more realistic code bases. One goal would be to take pre-existing Coarray Fortran benchmarks and translate them to use this library. Other testing would focus on the overheads of the object-orientated operations in realistic codes compared with hand-written OpenSHMEM code, and indeed Coarray Fortran code. It would also be desirable to benchmark its performance with other types of machine running OpenSHMEM, to determine how suitable the library is for other interconnects. An attempt was made to install the OpenSHMEM reference implementation on to a conventional Intel cluster, but there were issues with getting GASNet to run, and thus this was left aside to work on other areas of the project.

Another issue for consideration is how the library would interact with C++ objects. The

Cray PGAS Runtime for C++ library supports accessing object members, and calling remote object methods through the use of a remote procedure call (RPC) system, albeit with certain limitations (for example, the remote procedure calls are limited to two up to function arguments). This was considered beyond the scope of this work, and was not investigated.

While a library like this cannot compete with the potential efficiency and optimization levels offered by a language extension with a dedicated compiler, runtime and toolchain, it could be used to assist the parallelization of legacy code using single-sided transfers, thus lowering the barrier of entry to PGAS programming.

# Bibliography

[1] S. R. Sachs, K. Yelick et al. *Exascale Programming Challenges* 2011, Report of the 2011 Workshop on Exascale Programming Challenges

[2] D. Henty *A Parallel Benchmark Suite for Fortran Coarrays* 2012, IOS Press. In Applications, Tools and Techniques on the Road to Exascale Computing, pp. 281-288.

[3] D. Henty *Performance of Fortran Coarrays on the Cray XE6* 2012, In Proceedings of Cray User Group.

[4] Saraswat et al. *The Asynchronous Partitioned Global Address Space Model* IBM

[5] M. Eleftheriou, S. Chatterjee et al. *A C++ Implementation of the Co-Array Programming Model for Blue Gene/L* 2002. In proceeding of: 16th International Parallel and Distributed Processing Symposium (IPDPS 2002), 15-19 April 2002, Fort Lauderdale, FL, USA, CD-ROM/Abstracts Proceedings

[6] T. Johnson *Coarray C++ ProgrammerâĂŹs Guide Version 1.0 (DRAFT)* 2012. Cray Inc.

[7] Cray Inc. *Using the GNI and DMAPP APIs* 2011 - docs.cray.com/books/S-2446-3103/S-2446-3103.pdf (Accessed August 2013)

[8] W. W. Carlson, J. M. Draper et al. *Introduction to UPC and Language Specification* 1999. CCS-TR-99-157

[9] R. W. Numrich, J. Reid *Co-Array Fortran for parallel programming* 1998. ACM FORTRAN FORUM, Vol. 17, No. 2, Pg. 1 - 31

[10] B. W. Kernighan and D. M. Ritchie *The C Programming Language, Second Edition* 1988. Prentice Hall

[11] International Standards Organization *ISO/IEC 9899, Programming Languages C, Second Edition* Adopted May 2000

[12] W. Gropp *MPI at Exascale: Challenges for Data Structures and Algorithms* http://www.cs.illinois.edu/w̃gropp/bib/talks/tdata/2009/invited-mpi-exascale.pdf (Accessed August 2013)

[13] The OpenSHMEM Project *OpenSHMEM Specification 1.0* 2012

[14] M. Metcalf and J. Reid Fortran 90/95 Explained, Second Edition 1999. Oxford University Press.

[15] S. Meyers *Effective C++, Third Edition* 2005. Pearson Education.

[16] D. Abrahams and A. Gurtovoy *C++ Template Metaprogramming, First Edition* 2005. Pearson Education

[17] Ashby et al. *The Opportunities and Challenges of Exascale Computing* Fall 2010, US Department of Energy.

[18] D. Bonachea *GASNet Specification 1.1* 2002, University of California, Berkeley, Report No. UCB/CSD-02-1207

[19] Kogge et al. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems* September 2010. DARPA IPTO

[20] S. W. Poole et al. *OpenSHMEM: Towards a Unified RMA Model* 2011, Encyclopedia of Parallel Computing - Vol. 4, Pg. 1380

[21] Intel Corporation *Tutorial - Array Notation* http://www.cilkplus.org/tutorial-array-notation (Accessed August 2013)

# Appendix A

# The ISC Student Cluster Challenge

I was one of the members of the four man team that represented EPCC at the Internation Supercomputing Conference Student Cluster Challenge at Leipzig in 2013. We had to run Linpack, four sets of benchmarks, and two ‘secret applications’, which were benchmarks that were announced on the days of the competition where we had to run them. In addition, the cluster hardware could not consume more than three kilowatts of power at any point during the benchmark runs. The four primary benchmarks were -

- The HPC Challenge (HPCC) benchmark suite which features a number of HPC benchmarks, including the STREAM memory benchmark, and Linpack (which had to be run as part of HPCC in addition to the main Linpack run).
- Gromacs (GROningen MAchine for Chemical Simulations), a molecular dynamics package.
- WRF (Weather Research and Forecasting), an meteorological simulation package.
- MILC (MIMD Lattice Computation) A quantum chromodynamics package. This proved to be the most difficult benchmark code to get working due to a seemingly endless list of issues with the code that required us to contact the developers and acquire an alpha of the next version of the code.

The two secret benchmarks were -

- AMG - A benchmark from the US National Energy Research Scientific Computing Center described as “a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids".
- CP2K - Another molecular dynamics package.

Our cluster consisted of a set of four Boston Venom 1U rackmount servers, each featuring two eight-core Xeon E5-2670 CPUs, two Nvidia Tesla K20 GPUs, 64GB of main memory and 120GB of solid-storage. In addition, each node had an FDR Inifiniband interface, and were connected to an 18-port switch[unfortunately the smallest switch the vendor had available]. The infiniband interconnect was used exclusively for MPI traffic. In order to prevent any interference with the performance of MPI, we set up an ethernet network between all four nodes to allow us to ssh into them either from the head node, or our laptops. Since the power consumption of the network switches counted towards the power limit, we used a consumer-grade 4-port 100Mbps Ethernet switch for this network.

We successfully competed in the competition, achieving the second highest Linpack score (with a score of 8.321 Tflops vs the high score of 8.455 Tflops), the best performance for one of the MILC data sets, the second best for one of the others, as well as successfully running the other two main programs and one of the secret applications. (Unfortunately, it took a long time to get CP2K working due to a bug in the latest version, and by the time we got an older version running, there was not enough time to run the whole data set).

My original official role in the team was that of a hardware specialist, in practise, I ended up working with Gromacs and WRF.

My project preparation report, submitted in April, details the work I carried out up to that point, and so I will continue from where that report left off. Not long after that report was submitted, we changed our hardware from ARM-based Boston Viridis servers to the Boston Venom servers described in the introduction. There were two reasons for this - one was that while a single Viridis node consumed about half the power and featured twelve times the number of processing cores as one of the Venom nodes, the Nvidia accelerators in the Venom meant that a single Venom node offered 10 times the performance of a single Viridis node when running Linpack. If the GPUs were removed from the machine and Linpack was run on the CPUs alone, the power consumption of the Venom would be slightly less than that of the Viridis, but the Venom's Linpack score would be twice that of the Viridis. Similar results were found from the other packages.

In the latter CPU-only case, some of the difference could be accounted for by the availability of highly optimized binaries, libraries and compilers for x86 processors, but nonetheless, the fact was that the Venom 'platform' was anywhere between 2-10x more power efficient (in terms of flops per watt) than the ARM system when running Linpack, and similar results were found for other application, inclding at least Gromacs and WRF. The other performance issue with the ARM system was that we never managed to get access to more than one half of a single server unit (and for most of the time we benchmarked it, we only had access to four quad-cores, in other words, one twelfth of a full system). Given that the Viridis used 10Gbps as an interconnect between each quad core processor in the system (of which there were a total of forty-eight) there were concerns as to how well the network would scale if we used several fully populated Viridis servers. Ultimately, these machines were designed for situations where the number of cores available is more important than raw computing performance - such as web serving/hosting and cloud computing applications. They were not intended as HPC systems.

The other issue was slightly more pressing - a fully populated Viridis server had a list price of $50,000, most of which (or so we were told) was accounted for by the price of the so-called Calxeda 'EnergyCards' which were the self contained units from which the Viridis was built from. Each consisted of four quad-core ARM processors each with two integrated 10GBE controllers and a management core, as well as memory slots and storage ports. We would have needed 8-10 fully populated Viridis to fill up our 3kW power budget, and Boston simply did not have this many available for us.

Our original plan when we switched to using Intel systems was to have a mix of four GPU and four CPU-only systems. The plan was that for GPU-dependent benchmarks, we would only use the GPU machines, but for CPU-only ones, we would use the CPUs across all eight nodes. This would mean that we would have an ideal irrespective of whether a given application used GPUs or not. However, this was based on assumptions that the idle power consumption of a CPU only node was much lower than it actually was (we estimated 50W, it was more like 150W). This effectively forced us to choose between aiming to get a high Linpack score (by having a small number of nodes with as many GPUs as we could accommodate) or aiming for overall performance (by using lots of nodes to have as many CPUs as possible) in the power budget.

The timing of the switch over to the new hardware was quite late, and relatively close to the second examination period, which took away 2-3 weeks of time to work on the cluster. The main work around this time involved rebuilding all the applications on the new system, and getting familiar with the compilers and libraries on the Intel system, which were different to those on the ARM system (for example, the Intel system had the Intel Compiler Suite and the Intel optimized maths libraries). Unfortunately, this process had to be repeated as we moved from Boston's CPU only cluster to Boston's GPU cluster, and once again when the GPU cluster was taken aside and given a fresh install in preparation for our use during the competition. I then spent time repeating the process for different compiler flags, and different combinations of libraries, many of which had little to no effect on the overall performance of the benchmarks I worked with.

We travelled to our vendor, Boston, where we were shown the system, how it was wired up, and how to remove one of the nodes, dismantle it, and replace faulty hardware (which came in useful on the night that the hardware arrived on the night before the competition with one of the nodes unable to boot the operating system due to a damage SATA cable) .

The trip unfortunately coincided with one of the GPUs in one of the nodes failing, leading it to give erratic results. Through a stroke of further misfortune, we took that particular node, wiped it clean, and installed a different Linux distribution (Ubuntu server) to compare the performance and background memory and CPU usage compared with the original Red Hat which also had a cluster management tool installed. We managed to get the server running the benchmarks on top of Ubuntu within a few hours starting from scratch, but the faulty GPU gave us the impression that something was wrong with our configuration. Fortunately the guys at Boston used the cluster management software to restore the node to a run fully configured Red Hat installation, however, there was a mismatch between the drivers on the freshly wiped node, and the ones on the other nodes. While these were inconsequential, we concluded that they explained the erratic performance that was actually caused by the faulty GPU. Thus, a lot of time and effort was spent chasing after faulty hardware that failed in ways that suggested a configuration or software failure.

At one point I ran Intel's PowerTop utility, which, amongst other things, suggests certain configuration details that can be used to decrease the power consumption of the machine in Linux. Applying these suggestions given for the server had no meaningful impact on power consumption. Ultimately, these tools are designed for mobile devices where a difference in power consumption of a fraction of a watt can have a noticable impact on battery life. On a machine whose idle power consumption is about the same as ten laptops under load, such differences fade into the realms of statistical insignificance. There's probably scope for an interesting project to design a tool like this for HPC systems, but I somehow doubt it would fit into the time that we had available to us.

The most useful part of the trip (with respect to the competition) was receiving advice from a consultant from Nvidia regarding running Linpack on our cluster. He gave us an optimized Linpack binary, explained the significance of the Linpack input parameters, and how we should tweak them to suit our system. As result of this help, we were able to achieve parallel efficiency of about 98% across the eight GPUs and sixty-four CPU cores in our system, whereas beforehand our scaling was quite poor.

One thing that occured to me around the time of the switch from the Viridis to the Venom was that while the four of us understood the hardware, the operating system, the build process, and the potential tweaks we could try such as the use optimized libraries and custom compiler flags, we didn't really know a lot about the software we were using, what it did, and whether or not we were using it in a sensible manner. It seems to me that the programs we did well on were ones that we spent time learning about, the other ones were sort of like black boxes to us that read in some benchmark data and returned a performance figure.

Officially I was meant to focus on hardware. In practise, this didn't mean a lot since the hardware we used was dictated by what Boston was able to provide us. This is not to say that this was a problem, on the contrary, what we got was probably the configuration what we would have picked if we had experimented with a variety of configurations and found them. However, it narrowed the scope for hardware related work. Setting up the hardware was a straightforward task, and aside from some of the hiccups mentioned above, worked largely as expected. Any hardware decisions (like whether we used a large number of CPU-only nodes or a small number of GPU nodes) were decisions that affected everyone on the team, and thus were taken as a team. In this context, it didn't really make much sense to have a 'hardware guy', and it started to feel like unnecessary compartmentalization.

A similar fate befell the 'OS guy'. The operating system we used was the one that was installed onto the hardware. Aside from our brief experimentation while we had physical access to the hardware, we had a simple realization - the vast majority of HPC hardware runs Linux. The vast majority of HPC software runs on Linux. We were never going to run anything other than Linux. Nor did it matter which distribution of Linux we ran, the odds that switching to a different distribution would offer an appreciable speed difference were vastly outweighed by the odds that there would be no difference or that problems would occur, and thus we would be forced to revert. Similarly, no optimizations were made to the OS, largely because it never seemed like there was anything about it that needed to be touched. The role would have perhaps made more sense in a context where other team members were not familiar with configuring a Linux system and building software for it, but this was not the case for our team.

The application guys fared somewhat better (after all, we all ended up as application guys, irrespective of what we signed up for), but again, the applications roles focused on figuring out how to build the software, and figuring out how to use it and run it across the cluster. As stated before, any significant optimization of performance came from figuring out how to configure the program run for the given data set input. Playing with compiler flags or changing libraries had little to no difference, not least because some of the applications' installation procedures (WRF in particular comes to mind) had a mechanism for determining which compiler was installed on the system, and using the correct flags for all combinations of CPU architectures and compilers it was aware of.

As far as I know, no one went anywhere near the source code of the applications (neither within our team, or in any of the other teams that we talked to), and indeed this never seemed like a good idea. The applications had substantial code bases (Gromacs, for example, consists of 1.67 million lines of C, WRF is about half a million lines of code, primarily Fortran with some C) that have been developed over a long period of time, ported to many varieties of machine. It would seem unlikely in this context that the four of us would find some performance issue with the code on our relatively standard hardware (assuming that there even were any), dig into the code, figure out what it does, profile it, and make changes to the code that would speed up the code to an extent that would justify the time invested (while juggling all the other things that we had to do to prepare for the competition, as well as other requirements of the MSc).

It's hard to imagine how any sort of coding could have been fitted into the procedure without altering the nature of the competition. Personally I would have liked (and I wasn't the only person on the team who was of this opinion) if the organizers of the competition scrapped some (or all) of the pre-existing benchmark codes, and had a cluster programming competition instead, where some problem is presented to which the teams are required to write code to solve the problem in an efficient manner on the cluster. This could encourage designing clusters that are easy to use as opposed to being optimized around achieving a high Linpack score, for example.

While I was in Germany, I mentioned this idea while talking to a coach of one of the other teams who was involved in the organizing the American version of the competition. He said that the idea of having a programming element in the competition had been considered before, but was ruled out as most of the participants in the competition are undergraduates and don't have any experience with HPC programming. Given that the four of us have spent a year in a postgraduate degree to pick up these skills, this seems like a fair point, but nonetheless, it seems like such a competition would have represented a better outlet for the our new-found HPC skills than the 'configure-make-sudo make install' shuffle that occupied a considerable portion of our time.

The other application-related issue was the division of labour. We ended up dividing the application between us so that they could be readied in parallel (I ended up with two of them due to one of the team members being unable to take part for quite some time due to personal problems). In hindsight, we all seem to be of the opinion that we should have worked on each application together. This would have achieved two benefits - one was that in case someone fell ill, we would have all understood the applications. The second was that there was a good chance that the specific problems we ran into with certain applications could have been resolved if we had multiple people looking at them at once. There could be some degree of specialization in preparing for this, for example, one person could figure out how to build the code, one person could figure out how to run it across the cluster and ensure compatibility with accelerators (if relevant). Another could research what the code does, and how to efficiently run different types of input data. It would be important that everyone then shares what they find out so that no tasks can only be carried out by one person.

In conclusion, I guess one would be forgiven for reading the above and concluding that I had no fun and that I regret doing it. This however is not the case. I had a lot of fun was had playing with exotic cluster hardware that I never would have had access to anyway and coaxing some hideous software to well on it. There were definitely some monotonous aspects to what I did, but given that I sometimes do stuff like this my spare time, that wasn't a particular issue. I made great friends with the other guys on the team, and we had a lot of fun together experimenting with things, discussing performance and power figures, strategies and hardware. The guys from Boston and Viglen were great to work with, and we learned from their experience. The trip to Germany was similarly great fun and we met some great people from the other teams and got to talk to vendors and see their new hardware.

Nonetheless, I think the problem for me with the whole experience was that by attempting to make it the basis of our dissertations, it created certain expectations. If it was something that we had did in our spare time, I would focus more on the positives, but given the pressure to turn it into a 'learning experience' it ended up detracting from the enjoyability of the experience, particularly while searching for a dissertation opportunity related to the competition that never arose. This is not to say that it wasn't a learning experience, but it wasn't one that translated into a dissertation given what I was interested in, and given what I signed up for initially. I think that if I was told that I would have to do competition in the spare time while doing a regular dissertation, and that it would be organized as such from the start, I would probably would have still signed up. After all, it's what I ended up doing, and I may have had a lot more fun doing it if I knew what I was getting into from the outset.