Evaluating Fortran Coarrays and MPI on a Modern HPC architecture

Daniel Robinson

August 19, 2011
Abstract

The increasing numbers of cores sharing memory within a node, on MPP systems, has raised the question as to whether pure MPI applications are extracting the best performance from the architecture and whether message passing is the most appropriate model for these machines. Among several emerging programming models in HPC, the PGAS (Partitioned Global Address Space) and hybrid models are potential successors to pure MPI programming in an attempt to gain better performance out of many-core and multi-tier hierarchical architectures. These models aim to decrease the cost of message passing between cores sharing memory and in the case of PGAS languages and MPI-2, provide direct remote memory access functionality to aid in overlapping communication and computation. This project aims to compare and evaluate Fortran coarrays and MPI as a means of developing a distributed parallel sorting algorithm for a Cray XE6 system.

The test application was chosen to be sorting because it is a common computational task used in many other applications and there is no limit to problem size other than physical constraints like time and memory availability. The sorting application will be a parallelised variant of the quicksort, a popular serial sorting algorithm which due to its divide and conquer technique lends itself reasonably well to parallelisation. Fortran Coarrays were chosen as the PGAS language because they are now officially part of the Fortran 2008 standard and so support for developing with coarrays will increase therefore usage of coarrays is also expected to increase. The massively parallel XE6 architecture on the UK national supercomputing service, HECToR, will be used to test the code which provides a good representation of a modern MPP computer, making it an ideal test machine for comparing message passing and PGAS models.
# Contents

<table>
<thead>
<tr>
<th>Chapter 1 Introduction</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 2 Background</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Fortran Coarrays</td>
<td>3</td>
</tr>
<tr>
<td>2.1.1 Declaring Coarrays and Intrinsic Functions</td>
<td>3</td>
</tr>
<tr>
<td>2.1.2 Communications</td>
<td>4</td>
</tr>
<tr>
<td>2.1.3 Synchronization</td>
<td>5</td>
</tr>
<tr>
<td>2.2 HECToR</td>
<td>5</td>
</tr>
<tr>
<td>2.2.1 Compiler</td>
<td>5</td>
</tr>
<tr>
<td>2.2.2 Timer</td>
<td>6</td>
</tr>
<tr>
<td>2.2.3 Compute node architecture</td>
<td>6</td>
</tr>
<tr>
<td>2.2.4 Network Software Interface</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Quicksort</td>
<td>7</td>
</tr>
<tr>
<td>Chapter 3 Distributed Parallel Quicksort</td>
<td>9</td>
</tr>
<tr>
<td>3.1 Design</td>
<td>9</td>
</tr>
<tr>
<td>3.1.1 Select Pivot</td>
<td>9</td>
</tr>
<tr>
<td>3.1.2 Redistribute Keys</td>
<td>9</td>
</tr>
<tr>
<td>3.2 Implementation</td>
<td>11</td>
</tr>
<tr>
<td>3.2.1 Overview</td>
<td>11</td>
</tr>
<tr>
<td>3.2.2 Select Pivot</td>
<td>12</td>
</tr>
<tr>
<td>3.2.3 Counter</td>
<td>12</td>
</tr>
<tr>
<td>3.2.4 Scan</td>
<td>13</td>
</tr>
<tr>
<td>3.2.5 Global swap</td>
<td>13</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table 1: $10^6$ kpc scan time</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2: High core count sorting efficiency</td>
<td>25</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Coarrays</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>XE6 node</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>GNI and DMAPP</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Key redistribution</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>Pivot tweaking</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>Local sort performance</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>Initial state</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>Second state</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>Third state</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>Final state</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>$10^5$ kpc pivot select</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>$10^5$ kpc counter</td>
<td>18</td>
</tr>
<tr>
<td>13</td>
<td>$10^6$ kpc scan and scan.v2</td>
<td>19</td>
</tr>
<tr>
<td>14</td>
<td>$10^6$ kpc scan.v2</td>
<td>20</td>
</tr>
<tr>
<td>15</td>
<td>$10^3$ kpc scan.v2</td>
<td>20</td>
</tr>
<tr>
<td>16</td>
<td>$10^3$ kpc global swap</td>
<td>21</td>
</tr>
<tr>
<td>17</td>
<td>$10^6$ random kpc sort efficiency</td>
<td>22</td>
</tr>
<tr>
<td>18</td>
<td>$10^5$ random kpc sort efficiency</td>
<td>23</td>
</tr>
<tr>
<td>19</td>
<td>$10^4$ random kpc sort efficiency</td>
<td>23</td>
</tr>
<tr>
<td>20</td>
<td>$10^3$ random kpc sort efficiency</td>
<td>23</td>
</tr>
<tr>
<td>21</td>
<td>Work plan</td>
<td>31</td>
</tr>
<tr>
<td>22</td>
<td>$10^6$ kpc pivot select</td>
<td>33</td>
</tr>
</tbody>
</table>
Acknowledgements

Thanks to Dr. Lorna Smith at EPCC and Harvey Richard at Cray for their advice during the project. Thanks to EPSRC for enabling me to undertake the MSc through one of their studentships.
Chapter 1

Introduction

Modern HPC architectures commonly consist of a hierarchy of processing cores, with several multi-core processors within a single node. These types of architectures are challenging current programming models to get the best possible performance out of the hardware [1]. The majority of large scale distributed memory applications are parallelised with the MPI library [1]. MPI is an implementation of the message passing model, by where each process holds its own private data and communicates with other processes by explicitly sending and receiving messages [1]. Sending and receiving messages is a robust though costly communications method as both participants must stop working to actively communicate with one another. This type of communication may not be optimal for cores which are located close to one another and/or sharing an address space. Communications could potentially be much faster and efficient through the use of shared variables and allow for overlapping communications with computation. MPI-2 has attempted to address this with remote memory access functions which could behave like shared memory access on multi/many-core processors and/or multi-processor nodes, though the majority of MPI programs still mainly use MPI-1 functions for communications [1].

The PGAS model (Partitioned Global Address Space) is an alternative to the message passing model which may be appropriate for modern HPC systems. Each process reserves some memory to be part of a single global address space. The global address space is logically partitioned such that the programmer can specify how data is to be distributed across processes and take advantage of locality of reference [1]. PGAS languages communicate using direct remote access to variables in the global address space and require the use of synchronisation to ensure program correctness, similar to shared memory programming.

This project aims to compare the PGAS and message passing models in terms of their usability and performance on a modern HPC architecture. A parallel quicksort will be developed using Fortran coarrays and Fortran with MPI-1 and observed for how well they perform on the UK national supercomputing service HECToR. HECToR is currently a Cray XE6 system [2] and so should provide an excellent test system for comparison of the two models on a modern HPC architecture.
The test application is suitable because it addresses a real computational problem, one which occurs in many other applications and has no limit for potential problem sizes. The sorting algorithm makes use of a combination of point-to-point and collective communications.

The dissertation structure follows a similar structure to that which the work was undertaken. Chapter 2 briefly covers background material on the PGAS model and Fortran Coarrays, the software and hardware of the test machine and the serial quicksort algorithm.

Chapter 3 covers the design and implementation of the parallel sorting algorithm, the reasons behind certain decisions that were made and any notable differences between the MPI and coarray versions.

Chapter 4 looks at the profile data from the main components of the parallel sort in order to provide a better understanding and analysis of the final performance results.

Chapter 5 contains the overall performance results and analysis of the sort. The experiences of developing with coarrays and MPI are also discussed here.

Chapter 6 finishes with the conclusions from the project and suggests any related future work.
Chapter 2

Background

2.1 Fortran Coarrays

Coarrays are a small and simple syntax extension to standard Fortran which allow programmers to write parallel applications for both shared and distributed memory architectures. Fortran coarrays are an implementation of the PGAS model for parallel programming, the PGAS model is similar to shared memory programming with the additional benefit of being able to express the layout of data which is necessary to create scalable high performance programs for distributed systems [3] A Fortran coarray program runs as a fixed number of copies executing asynchronously, each copy of the program is referred to as an image [4]

2.1.1 Declaring Coarrays and Intrinsic Functions

To declare a variable as a coarray simply use an additional trailing subscript in square brackets similar to declaring a normal array, for example:

real :: coScalar[*]

integer :: coArray(2,2)[*]

These indicate that every image should have a real scalar and an integer 2x2 array which can be accessed by other images via the codimension index. The codimension is given a wildcard value to ensure it is initialised with the number of images executing the program as any coarray must exist on all images. This is a bottom up construction as the programmer must declare how the data structure will appear on each image instead of declaring the global object and specifying how it will be distributed like in the UPC language for example. Figure 1 illustrates the previous example when run on 4 images.

It is possible to declare variables with multiple codimensions, in this instance it makes sense to provide the upper bounds to all of the codimensions except the last one, for example:

integer :: myCoarray[3,3,*]
The codimensions cycle in the same order as normal Fortran array dimensions, so [1,1,1], [2,1,1] refer to images 1 and 2 respectively.

The most frequently used coarray functions are those which return the unique image index of the calling image this_image() and the total number of images num_images(), also image_index(z, cosubs) returns the image index of the coarray z with cosubscripts cosubs.

![Coarrays on 4 images, each coarray must exist on all images.](image)

**2.1.2 Communications**

Communicating with coarrays is a very simple matter, the programmer only needs to write standard assignment statements indicating the cosubscript in order to put or get data to or from a remote image. For example the following code is a coarray all-reduce-sum:

```fortran
numTotal = 0
do i = 1, num_images()
    numTotal = numTotal + num[i]
end do
```

Communicating in this manner for a collective operation may take some experimenting to find a good implementation. The above example tells each image to access all other images in the same sequence, 1 to num_images(). This technique may cause the network to become flooded at certain points as all images try to access the same image at once. A better implementation for such an operation may be to stagger the access to other images in a sequence starting from a neighbouring image, for example:

```fortran
numTotal = num
do i = this_image() + 1, num_images()
    numTotal = numTotal + num[i]
end do
do i = 1, this_image() - 1
    numTotal = numTotal + num[i]
end do
```
2.1.3 Synchronization

The communications in PGAS languages are single-sided meaning only one image needs to be concerned with transferring data whilst the other image remains, effectively, unaware of the data being copied to or from its local variables. This type of communication means that images must explicitly synchronise with each other to ensure program correctness, compared with MPI-1 communications where message passing itself acts as a form of synchronization, because all participating processes are actively involved in the data transfer process. The programmer is responsible for using synchronization statements to avoid race conditions. Blocks of coarray Fortran code are split into execution segments bounded by synchronization points to ensure that remote values are not prematurely overwritten or read. Synchronization is done using the statements:

```
sync all
sync images(imageList)
```

The `sync all` statement causes the calling image to halt until it has synchronized with every other image executing the program. The `sync images` statement causes the calling image to synchronise with every image in the argument `imageList`. It is important to note that images can synchronise using different `sync all` statements, such that if images follow different paths, through a control of flow statement, each path must have matching synchronisation statements or the program will almost certainly deadlock or produce incorrect results.

2.2 HECToR

The application was developed and tested on the UK national supercomputer service HECToR which is currently a Cray XE6 system[2]. This section will briefly discuss the compiler, timer, compute node architecture and network of the HECToR service.

2.2.1 Compiler

The project made exclusive use of the Cray programming environment to use the Cray Fortran compiler for compiling both coarray and MPI versions of the application. The module `PrgEnv-cr` is loaded using the `module swap` command (assuming the current programming environment is the Portland group):

```
module swap PrgEnv-pgi PrgEnv-cr
```

MPI code is compiled using the `ftn` wrapper which automatically links the MPI library. Coarray code is compiled by adding the flag `−h caf` to the `ftn` wrapper.

The Cray Fortran compiler has several optimisation flags which can be used to attempt to obtain an automatic performance increase by compiling under different constraints. The compiler flags below were used but found to have no effect compared to using the default values:
-O 3 General level of optimization either 0, 1, 2 or 3 (default 2)

-O aggress Treat program units as individual optimization regions, can improve the optimization of large program units but may increases compile time and executable size (default noaggress)

2.2.2 Timer

To measure the performance of the codes the MPI function MPI_Wtime was used in both MPI and coarray versions. The times were measured either side of specific sections of code for profiling and across the length of the global sort for the total performance. The MPI function MPI_Wtick is used to obtain the resolution of the MPI_Wtime readings. On HECToR MPI_Wtime has a resolution of $1 \times 10^{-6}$ seconds.

2.2.3 Compute node architecture

An individual node on HECToR has two AMD 2.1GHz 12-core processors each of which is comprised of two hex-core dies connected via the same socket[2]. Although a node has 32GB of memory which is logically shared between all the cores, each hex-core die is a NUMA region where all cores have equal access to 8GB of DDR3 memory, access to the other three memory blocks within the node is more costly via the hyper-transport links where latency is increased and bandwidth is reduced[5]. Figure 2 shows a simple diagram of the node architecture.

Figure 2: XE6 node consisting of 4 NUMA regions connected via hyper-transport links, image referenced from[6]
Two nodes share a Gemini interconnect router chip containing 10 network links which is used to construct the processors into a 3D torus configuration [5]. The Gemini interconnect has 5GB/s bandwidth and 1 - 1.5 µs latency and supports single sided communications which can bypass the operating system, this hardware support will be highly beneficial for PGAS languages [7].

2.2.4 Network Software Interface

The Gemini network uses the user Generic Network Interface (uGNI) to implement MPI and the Distributed Shared Memory Application (DMAPP) interface to implement SHMEM and PGAS communications [8]. Both APIs allow direct access to the Gemini Hardware Abstraction Layer (GHAL), by-passing the operating system [8].

![Figure 3: GNI and DMAPP software layers, image referenced from [8].](image)

2.3 Quicksort

Quicksort was originally devised by C.A.R. Hoare [9] in 1961. It is a very popular sorting algorithm as it is simple to implement, often through a short recursive function call and is capable of good performance. The algorithm essentially has 3 steps:

1) Select a pivot value

2) Partition the list into two with keys less than the pivot in one list and keys greater than the pivot in the other list

3) Apply steps 1 and 2 to each sub-list.

If there is any knowledge about the distribution of keys before the sort, the pivot selection criteria can be altered so the sort will operate closer to its optimal case. The optimal case for a quicksort of n keys is \( O(n \log(n)) \) when each section is split exactly in half, and worse case \( O(n^2) \) when each section only has one element cut off of the end.
Due to its divide and conquer technique, it is almost trivial to implement parallelisation in a shared memory architecture.

On a shared memory system, each time a thread divides a section of keys both sections can be placed on a stack/queue data structure and idle threads can pick up these sections and work on them in isolation as they are completely independent. One downside to this technique is that the first pass through the keys is completely serial and this is the longest individual section in the sort so for p threads p-1 are sitting idle, similarly the second step involving two sections will be using two threads whilst p-2 sit idle. For very large thread counts and large key sets, this will lead to many cores sitting idle for long periods of time before finally receiving a section to work on.

Additionally the situation is made far worse when using this type of parallelisation technique on a distributed architecture as the time for 1 process/thread to scan across the entire key set, over multiple nodes will be much higher due to increased latency and reduced bandwidth over the network. The next chapter will discuss other parallelisation techniques for the quicksort to overcome these problems.
Chapter 3
Distributed Parallel Quicksort

3.1 Design
The initial base for the distributed parallel quicksort will start from an algorithm almost identical to the serial version only with the keys spread across multiple images. The global sort will start from an initial state where each process has the same number of keys and perform an initial local sort so the keys on each process are in ascending order. The sort will finish when the keys are in ascending order across all processes in rank order. The design for the distributed parallel quicksort must overcome a particular issue with the quicksort algorithm in order to be efficient at large core counts on a distributed architecture which will quickly become apparent.

3.1.1 Select Pivot
The first step is to pick a pivot value from the global set. This can be done in any way and should be modified by the user to most efficiently bisect the sets of keys. The pivot selection criteria are very important in achieving an efficient quicksort but as the criteria will need to vary depending on the distribution of key values this project will use collective reduce operations to find the global maximum and minimum values then simply use the mid-point as the pivot. This is because the keys being sorted in the development of the parallel quicksort will be randomly generated with an approximately even distribution and so this method should cut each section close to the halfway point.

3.1.2 Redistribute Keys
The next step of the algorithm is to move the keys, which are less than or equal to the pivot, to the images with lower indexes of the current set and move the keys greater than the pivot to the higher indexed images of the current set. Figure 4 shows an example of this step, on 4 images, and realises the main difficulty with a parallel distributed quicksort, set overlapping.
For even a small number of random keys the choice of pivot selection is almost certainly never going to split the keys at a point which coincides with an image boundary. The example in figure 4 shows that the split occurs in the middle of image 3. In such a case the following recursive calls would be to sort the white and red sections independently. If the first statement within the recursive quicksort was to determine whether any of the keys in the set are owned by the calling image and to exit the call if there are no local keys involved, then images 1, 2 and 3 would continue sorting the white set whilst image 4 would return and then call the quicksort call for the red set. However this raises the issue that image 4 would be waiting on image 3 to finish sorting the white section until it was completely sorted which could potentially involve many more recursive calls.

One method to avoid this difficulty may be for image 4 to take ownership of the 2 red elements from process 3 so it can continue working on them independently. This is a fine solution for a problem of the size in figure 4, however when each image contains millions of keys the memory management of such a method would be difficult and may result in a very slow algorithm. There is the potential for many millions of keys to require redistribution to many images which then has further knock-on impacts on load balancing as some images now have more keys than others. Additionally, using coarrays to store the keys does not lend itself easily to this type of method as each image must contain the exact same coarrays of the same size. Dynamically allocating and deallocating coarrays implicitly causes a synchronization across all images so using dynamically allocated coarrays in this way would not be advisable either. Coarrays currently don’t support subsets of images so the coarrays are still required to be the same on all images even when dynamically allocated.

A second method may be to try and interrupt the recursive calls such that after image 3 has completed the first recursive call, to sort the white section in figure 4, it then passes back out and processes the first call to the red section. This method would require that each image keeps a record of every set of keys it is responsible for and ultimately spreads the idle time from one group of images to all of them, as each process which contains section boundaries continually changes which section it is sorting causing other images to sit and wait for it to come back to their section. The method would be difficult to implement because it breaks up the smooth recursive or iterative calls to the main sort function. It also requires a complex way of keeping a record of all the sections each image is currently involved in sorting.

A third method could be to simply say all images are responsible for all sections of the key array until the section is below some critical size. This would mean that all images take a piece of the current section and follow the same recursive function calls. Whilst this is not a completely unreasonable suggestion for a shared memory architecture.
where each image could work on the array without having to read or write the values from remote memory locations, on a distributed system it would be necessary to move large amounts of data around throughout the entire sort which would severely limit scaling to small core counts.

A fourth technique is to use the standard quicksort algorithm but to enforce a tweaking of the pivot value such that the set splits directly on an image boundary such that the two new sets are each completely owned by a unique set of contiguous images and each image maintains the same total number of local keys throughout the entire sort. This avoids load imbalance, memory management issues and potentially large amounts of idle time. There is a cost incurred in tweaking the pivot value but there are numerous ways to do this which may be able to make this cost small and perhaps even insignificant when compared to the savings. Figure 5 shows this technique using the same example as figure 4. This method may provide a simple implementation for a robust, flexible and scalable distributed parallel quicksort algorithm.

Figure 5: Pivot tweaking to achieve an even split of keys to avoid overlapping sets and load imbalance. The white and red sets are now completely independent.

3.2 Implementation

This section will run through the main components of the implemented quicksort to describe how the coarray and MPI version were implemented and how they differ in communication patterns.

3.2.1 Overview

Each image enters the parallel quicksort with the same number of randomly generated keys stored in a 1D coarray. For the purpose of testing, each image uses a random number generator with a unique seed to create the keys. The sort first calls a serial quicksort to sort the keys into ascending order on each image, the serial sort need not specifically be a quicksort, any will do, provided upon return the keys are in ascending order as the global sort phase requires this to be the initial state.

In the coarray version each image maintains a record of the images with largest and smallest image indexes in the current team of images working on the same set of keys, these two values are used for synchronizing teams of images independently and for exiting the global sort loop.
Each image also keeps a lookup table to store the number of keys which are less than or equal to the pivot value on each image within the current set. This table acts as a communications map during the swapping of keys between different images, the details of which will be described later.

The global sort phase is called in a do-while loop, the condition for stopping is when the team of images or MPI communicator size is one, meaning that the set of keys which the image is responsible for, covers only its local keys and so has finished sorting. The majority of the distributed parallel quicksort is spent within the global sort phase. This is where all of the communications of the quicksort are called and so each function will be described in detail for both coarray and MPI versions to allow for a more complete analysis of the results.

### 3.2.2 Select Pivot

The first call within the global sort is to find the value between the global maximum and minimum values to use as the pivot. Because the local keys are already in ascending order each time the global sort is called, the local minimum and maximum values can be directly picked out from the ends of the local arrays and then collective communications can be used to find the global values.

The MPI version uses two all reduce calls, one for the minimum and one for the maximum value. These functions are optimised to make use of tree algorithms which should be more efficient at higher core counts, whereas the coarray communications will use a flat communication pattern. The coarray version uses the lowest index image in the current team to scan through the current team for the global maximum and minimum values, it then calculates the pivot value locally and synchronises with the rest of the team so they may remotely read the pivot value back.

### 3.2.3 Counter

The second function is used to count how many local keys an image has which are less than or equal to the pivot value. The local count is stored in a 1D array in the element with the same index as the image index. The count values are then broadcast among the rest of the current team of images so all images know how many keys in the current global set are less than or equal to the pivot value and how many are local to each image within their team. This will act as the communication map during the global swap function.

The MPI communications are simply a gather to the root process, followed by a broadcast of the entire array. The coarray version initially attempted to use the message round a ring method to send data in one direction, always taking data from the same neighbouring image. This technique was chosen as it avoids using flat collective communications for every image which may flood the network at large core counts. This does however require a team synchronisation every iteration which ultimately proved to be very costly. Instead, a second version of the coarray counter function was implemented where each image uses a flat broadcast to place its count value into every
other images counter lookup table which performed far better, as it only required one synchronisation after the broadcast.

3.2.4 Scan
The Scan functions are used to move the point where the set will split. After the counter function, every image knows how many keys are less than or equal to the pivot and so can calculate where the section split will occur. If the default section split does not coincide with an image boundary then the program calculates which image boundary is closest and scans keys up or down accordingly.

In the case where the default split point is closest to the upper boundary of an image, the scan down function is called to find the lowest valued keys in the upper set and reassign them to the lower set to move the split point towards the upper image boundary. The root process performs the scan by point-to-point communications with each process in the communicator, evaluating each key in turn as it is received. After one pass the deficit of keys in the lower set has been reduced by one via updating the relevant elements in the counter lookup table. This is then repeated until the split point is on the process boundary.

In the MPI version, at the end of each individual scan, the root process must broadcast the rank of the process with the smallest key so that process can then send its next lowest value to be part of the following scan. In the coarray version, the root image does not need to broadcast after each individual scan as the point to point communications are one sided. This allows the root image to get the next key value without the other image needing knowledge of this. The image team synchronises when the root image has finished the complete scan so the other images may get the fully updated counter lookup table from the root image.

3.2.5 Global swap
After all the images have an updated counter lookup table the keys can be swapped between them. Each image can use the lookup table to independently calculate which keys it should take from other images to split the set correctly. This process is a swap which implies that keys being exchanged share a one to one relationship. If image ‘x’ must take 13 keys from image ‘y’ then so too, image ‘y’ must need to take 13 keys from image ‘x’. In terms of communications this is not terribly important for coarrays as each image simply takes what it needs, synchronises and then overwrites its old key values. But for MPI this is beneficial as the point-to-point communications can match up exactly and using the MPI_Sendrecv function provides a simple way of implementing the MPI version of the global swap communications.

There is one issue with the MPI_Sendrecv, which is that processes will frequently need to send and receive keys from more than one process. A process may therefore be required to wait until the complimentary process has finished handling any earlier MPI_Sendrecv calls it had to make before reaching this particular one. This is not an issue with coarrays as each communication is single sided and so each image never has a need to wait until the final synchronisation.
3.2.6 Section Merge

During the development of the parallel sort the requirement to keep each local section sorted, before the next call to the global sort, was satisfied by simply recalling the serial quicksort. This is an incredibly inefficient method as the sections of keys transferred between images during the global swap are already in ascending order. This means that each image typically has two to four long sections of sorted keys after each round of global swapping. A much quicker method of sorting is to merge the already sorted lists rather than re-perform a full local quicksort every iteration of the global phase.

This section of the sort does not directly contribute to the discussion of MPI and PGAS languages, however it was a crucial step, not only in getting the sort to run faster for pure performance sake but also to reduce the proportion of run time that was spent in the local sort phase in order to make the performance results from the different communications more visible. Figure 6 shows the amount of time spent using the generic quicksort and the developed custom merge sort.

![Figure 6: Local sort performance](image)

3.2.7 Correctness Test

To ensure that the sort behaved as intended figures 7 to 9 detail the movement of keys throughout the first global phase of a small scale run. Figure 10 shows the final key distribution at the end of the sort. Every time a sort completes the program compares the elements either side of image boundaries and checks they are in ascending order. If the keys are not in order an error message is generated for the output file containing details of where the keys are out of order to aid in debugging later on.
Each core contains multiple ascending lists.

Figure 7: Initial state, after the random keys have been locally sorted.

Figure 8: Second state, immediately after the global swap function. Each core contains multiple ascending lists.

Figure 9: Third state, immediately after the local section merge. This is identical to the initial state only duplicated over multiple independent sets.
Figure 10: Final state, at the end of the distributed quicksort the keys are in ascending order across all images in ascending index order.
Chapter 4

Profiling & Improvements

4.1 Profiling results

To understand the sort results fully it will be necessary to first present profile data for the main components of the global phase of the sort, to observe how they scale with core count and problem size (keys per core - kpc).

The pivot selection and counter subroutines are very simple in both their function and implementation when compared with the scan and global swap subroutines, which have more complex communication patterns. The pivot selection and counter subroutines use a simple combination of collective operations. The MPI pivot select uses two all-reduce calls whilst the coarray version uses a simple reduce to the root image followed by a broadcast. The MPI counter uses a gather to root followed by a broadcast from root, in the coarray version each image performs its own broadcast, placing its own count value into every other images counter table. The use of purely collective operations within these two subroutines automatically generates the expectation that the optimised MPI variants will perform better than the equivalent coarray versions. Figures 11 and 12 confirm this expectation, the total amount of time spent within the select pivot and counter subroutines respectively show that the MPI collectives are a better solution, out of the box, than the implemented coarray equivalents. The MPI versions scale far better to higher core counts than the coarray versions.

![Figure 11: 10^8 kpc pivot select](image)

Figure 11: 10^8 kpc pivot select, each plot shows the total amount of time spent within the select pivot subroutine using both parallel APIs.
Figures 11 and 12 only show the scaling for the problem size of $10^5$ keys per core. Where the scaling profiles differ greatly between problem sizes they shall be included within the main text of the dissertation, however in the instance of the pivot select and counter subroutines, the MPI version scales better at all problem sizes and so the plots for other problem sizes can be found in appendix A.

4.1.1 Scan

The scan subroutines lie at the heart of this particular parallel distributed quicksort algorithm. They are responsible for ensuring the splitting of keys coincides with an image boundary to allow the resulting sets to be processed in parallel and without altering the number of keys local to each core. The initial version of the scan function, described in section 3.2.4, used the root image to receive the relevant key value from the other team members one at a time and evaluate whether this was the largest or smallest value it had so far scanned. Once a value had been scanned it was discarded if it was determined that it was not a maximum or minimum value depending on the scan direction. This means that after each individual scan, the following scan would involve the root image reevaluating the exact same keys except for one, via more point to point communications. This was simple to implement however it was a very inefficient method.

A second version of the scan was implemented as the initial version was found to be a bottleneck. The main improvement was for the root process to gather the set of keys to be scanned in the first instance and stored on the root process rather than discarded after each evaluation. After each individual scan, only one new key needs sending to the root process which reduces the required number of point to point communications by almost 100%. As the root process is performing the scan, the requirement to send the next key from the process which provided the previously selected key is done via a single sided get, in the coarray version. However for the MPI root process to receive the new key it must first let the complimentary process know it must send the key. This requires that the root process performs a broadcast after each individual scan to let the current communicator group know who is responsible for sending the next key.
Figure 13 shows how the different implementations compare when sorting 1 million keys per core. Of the initial implementations the MPI version comes off best, whilst both versions use the same point to point communications each message consists of two integers so the MPI version will be using the eager message protocol such that the messages are, for the most part, sat waiting at the root process in the message queue before the root process posted the relevant receive calls which essentially hides much of the latency between the point to point communications. In contrast, the coarray version requires the root image to take charge of all the communications and so the time to fetch the remote keys greatly reduces the performance.

The second implementation is a massive improvement for both parallel APIs but the improvement favours coarrays as it can avoid the need to broadcast after every individual scan, as described earlier.

The scan subroutines provide an additional opportunity to mix coarrays and MPI within an individual function. Figure 13 shows that the coarray implementation is the better version of the scan subroutine, but it still requires an initial gather of the keys and a final broadcast of the lookup table at the very start and end. The profile data from the select pivot and counter subroutines shows that the MPI collectives perform and scale much better than the simple coarray flat collectives. It is possible to mix MPI and coarray code on HECToR without any difficulties other than those arising within the specific application code. A mixed version of the scan function was created using the coarray point to point middle section and an MPI gather and broadcast at the start and end respectively. An additional benefit to the MPI collectives is that it allows the subroutine to avoid calling any coarray synchronisation statements which can also be a cause of reduced performance (see next section: global swap). Figures 14 and 15 show the performance of the improved scan function, using pure coarray and MPI implementations as well as a mixed mode version, with problem sizes $10^6$ and $10^5$ random keys per core respectively.
The performance gains from using the MPI collectives are imperceptible when run with problem size $10^6$ kpc on up to 192 cores (fig. 14). The reason for this is because the serial scanning section in the middle takes up almost all of the scan time. Having more keys per core increases the maximum possible deficit size and so the deficit will on average be substantially larger for $10^6$ kpc than it will be for $10^3$ kpc. For example, should the worst case scenario happen where the default pivot value results in a split point which is directly in the middle of an images set of keys, when using $10^6$ kpc the deficit is half a million keys, and when using $10^3$ kpc the deficit is only five hundred keys. Due to the reduced average linear scan time when using $10^3$ keys per core, the performance gains from MPI collectives can be seen in figure 15. Additionally, the hybrid performance difference is more apparent with increasing core count due to the optimised MPI collectives improved scalability.

The MPI collectives, whilst insignificant up to 192 cores using $10^6$ kpc, do still have an effect, it just requires a larger core count to make the impact significant. Table 1 shows data from some additional sorts, which were run to illustrate this point, on a larger number of cores.
Table 1: $10^6$ kpc scan time, on larger core counts.

<table>
<thead>
<tr>
<th># Cores</th>
<th>CAF</th>
<th>MPI</th>
<th>MIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>384</td>
<td>40</td>
<td>408</td>
<td>31</td>
</tr>
<tr>
<td>768</td>
<td>166</td>
<td>1320</td>
<td>124</td>
</tr>
<tr>
<td>1536</td>
<td>436</td>
<td>3030</td>
<td>308</td>
</tr>
</tbody>
</table>

4.1.2 Global swap

The global swap subroutine only uses point to point communications to swap keys between images (see section 3.2.5). Figure 16 shows the amount of time spent within the global swap subroutine when sorting $10^3$ kpc. There is also a mixed version of the global swap which is simply the coarray version but with MPI barriers instead of coarray synchronizations. Figure 16 shows that the mixed mode version scales better because of this.

![Figure 16: $10^3$ kpc global swap](image)

The coarray version scales better than the MPI version as each image may take the keys it requires at its own convenience. The MPI version requires matching MPI_Sendrecv calls which causes some processes to wait as the complimentary process may be handling earlier calls to the send/receive function, this behaviour was predicted beforehand (section 3.2.5).

Now the main components of the quicksort have been discussed the total sort performance can be analysed in terms of these components.
Chapter 5

Results and Analysis

5.1 Sort Performance

The wall time was measured for sorting random keys using four problem sizes defined by the number of keys per core (kpc) rather than total keys per run. This is largely due to the fact that the key array is declared on each individual core and so it is simpler to run the code on multiple core counts using the same number of keys per core (kpc). Each set of data points for a given number of cores therefore still provides a direct comparison between the different parallel APIs, but the difference in time to sort between different core counts does not provide a direct comparison of how the program scales with a constant number of total keys. For this reason the wall time results are presented as a plot of sorting efficiency so the plots of individual APIs show how efficiency changes with core count for a constant number of keys per core (kpc). Figures 17 to 20 show the performance results for the total sort time. The hybrid version uses the pure MPI select pivot and counter subroutines in addition to the hybrid scan and global swap subroutines described in sections 4.1.1 and 4.1.2 respectively.

Figure 17: $10^6$ random kpc sort efficiency, blue and red plots are for pure coarray and MPI implementations respectively, the green plot is a hybrid implementation using MPI and coarrays.
Figure 18: $10^5$ random kpc sort efficiency, blue and red plots are for pure coarray and MPI implementations respectively, the green plot is a hybrid implementation using MPI and coarrays.

Figure 19: $10^4$ random kpc sort efficiency, blue and red plots are for pure coarray and MPI implementations respectively, the green plot is a hybrid implementation using MPI and coarrays.

Figure 20: $10^3$ random kpc sort efficiency, blue and red plots are for pure coarray and MPI implementations respectively, the green plot is a hybrid implementation using MPI and coarrays.
5.1.1 API comparison

The most obvious result from overall performance plots is that the pure MPI implementation is always the poorest performer. The reason for this is almost entirely down to the necessity for the MPI version to broadcast after each individual scan to enable the correct process to send its next relevant key to the root process, whereas the root image in the coarray version can simply fetch the relevant key for itself (fig. 14, 15 and table 1). The coarray global swap also improves the overall performance over the MPI version but only by a small fraction compared to the scan performance differences.

The hybrid code uses the best components from both implementations as well as hybrid scan and global swap subroutines which ensures its spot as the most efficient of the three versions. Because the main performance limiter is the scan function the coarray code performs almost equally as well as the hybrid version when using $10^6$ and $10^5$ keys per core. The hybrid version only begins to perform notably better at the maximum core count of 192 (fig. 17, 18). But at $10^4$ and $10^3$ keys per core the proportion of time spent in the serial scan section of the sort is reduced and so the benefits of the efficient MPI collectives become more apparent (fig. 19, 20).

5.1.2 Sort behaviour

A consequence of the sort bisecting the keys during each iteration is that it performs far more efficiently when run on a square number of nodes as this allows the keys to be split in half continuously allowing for optimal performance until the final two section splits occur. When the set of cores splits from six to three, the following split must simply remove one core from the set which then sits idle as it has finished and the remaining two cores then make the final section split. The last two splits are an unfortunate consequence of the XE6 nodes being comprised of twenty four cores and so cannot be avoided on HECToR.

When the number of nodes the sort is executed on is not a square number the sort will reach the situation where an odd number of cores are responsible for a set much earlier on in the sort, and so the following splits will always be forced to produce uneven set sizes. This also has a knock on impact in the instance of the test cases, used in the project, as the pivot selection criteria is increasingly likely to choose a pivot value which would result in the default split point being very close to the centre of an images local set. This ultimately results in the scan function, which is a serial section, having to repay close to the maximum possible key deficit.

The result of this is the shape of the plots in figures 17 to 20, most noticeably in the hybrid plot as it performs the best on all problem sizes. The efficiency can be seen to spike at two, four and eight nodes and dip on non-square node counts.

As an additional point of interest the sort was also executed using one million keys per core on 16, 32 and 64 nodes to observe whether the efficiency would drop or continue to grow. Table 2 shows the results of the additional tests.
The sorting efficiency continues to increase for all version of the quicksort up to 64 nodes (1536 cores) which was the highest core count attempted. The most efficient sort completed during the tests was using the hybrid code sorting 1,536,000,000 random keys in 1.22 seconds on 1536 cores, achieving over 1.26 billion keys on average sorted per second (table 2).

### Table 2: High core count sorting efficiency, using $10^9$ kpc

<table>
<thead>
<tr>
<th># Cores</th>
<th>CAF</th>
<th>MPI</th>
<th>MIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>384</td>
<td>4.65E+08</td>
<td>1.63E+08</td>
<td>4.87E+08</td>
</tr>
<tr>
<td>768</td>
<td>6.68E+08</td>
<td>2.00E+08</td>
<td>7.60E+08</td>
</tr>
<tr>
<td>1536</td>
<td>1.07E+09</td>
<td>2.69E+08</td>
<td>1.26E+09</td>
</tr>
</tbody>
</table>

5.2 Development Experience

For the most part the coarray syntax provided a clear and simple interface for writing the parallel quicksort and no major problems were encountered with the coarray syntax or semantics. The main issue found when developing the coarray version was that the lack of support of subsets of images and built in collective communications made the task more difficult and tedious than the MPI version.

#### 5.2.1 Communicators

The luxury of MPI communicators has no true coarray equivalent. It may be arguable that communicators are often used for sending communications in specific directions within 3D structured grid simulations, and coarrays cater for this instance by allowing programmers to declare coarrays with multiple codimensions. But communicators are far more flexible and so can be used in a much wider array of situations.

The MPI implementation of the quicksort simply uses the communicator split function to divide each set of process after each global phase. The communicator keeps each independent set of processes completely separate and the MPI communication calls can be written much more easily because of this. This greatly reduces the amount of low level code required to keep a track of which image belongs to which set and what section of the lookup table should be used for a given set and where the point to point communications should start and stop indexing in the codimension.

#### 5.2.2 Collectives

The MPI collective functions provide a feature rich, easy to use and generally highly optimised implementation for collective communications. Collectives can be implemented with basic coarray syntax but this is generally not advisable unless they are for a very specific and odd communication pattern where they may perform better than the MPI equivalent. For the case of supported HPC facilities like HECToR, the MPI library is very likely to be finely tuned by the hardware vendor to the specific machine architecture and so implementing custom collectives to perform better would
be very difficult, time consuming and not generally what application developers want to spend time on.

5.2.3 Hybrid

The final phase of development within the project was to combine the best parts of the two pure implementations in the hope that the hybrid code would maintain the best performance features from both pure versions. Whilst this was not a completely trivial task, it only took one week to integrate the coarray scan and global swap subroutines as well as the necessary low level infrastructure code into the MPI version. The final result was a better implementation which maintained the best characteristics of the parents. The single sided communications which proved so beneficial in the scan and global swap subroutines were used in conjunction with the MPI collective operations, including the MPI barrier for synchronisation, without any real difficulties.

The two programming models are not so different as to necessarily mean that writing hybrid code will be an arduous task. The default implementation on HECToR, sensibly assigns the images to physical cores in the same order as the rank in the MPI_COMM_WORLD communicator. For a linear application like the sorting algorithm developed here, this made the hybrid model quite straight forward and painless to implement.
Chapter 6

Conclusions

6.1 Parallel APIs

Within the scope of the project the final pure MPI version of the code was observed to perform the weakest. However this was down to a single key function which favoured the use of coarray single sided communications. Given more time, there are plenty of opportunities to improve the distributed parallel quicksort algorithm. It is easily conceivable that further improvements could allow for much of the MPI performance bottlenecks be hidden or removed and may ultimately tip the balance back in favour of the MPI version, depending on the algorithm that is used (fig. 13).

The development experience with coarrays is largely a positive one. The syntax is very simple and easy to learn and read. Coarrays are useful when single sided communications are necessary but when MPI communications will do there is little need or purpose in replacing them with coarray syntax. The additional loops and variables for codimension indexing makes the code look more cluttered than a single MPI function call. The prominent feature missing from coarrays as far as this project is concerned, is support for subsets of images. The MPI version was easier to write because of the ability to manage subsets of processes with a single communicator variable and just splitting the communicator at the end of each iteration.

The main conclusion from the project is that the MPI library is a feature rich and generally highly optimised interface for parallel programming on distributed HPC architectures, mainly due to its age and popularity in HPC. Coarray Fortran, by comparison, is a relatively new arrival to HPC and so hasn’t had the same amount of time or resources devoted to its implementation. It lacks much of the higher level functionality such as communicators for subsets of images and well defined collective implementations, but it still provides an easy to use interface for writing one sided communications which can prove very beneficial in certain situations arising in some HPC applications. What’s more, the ability to combine coarrays with MPI, which in this instance was relatively straight forward and painless, is a very attractive option as the cost of doing so may not be very high as the two programming models overlap considerably, certainly more so than MPI and openMP. Though the choice to use a hybrid code must be evaluated for each project individually and will be implementation
dependent, coarray Fortran certainly has some good aspects which may make it a preferable choice to openMP when developing a hybrid code.

6.1.1 Cray Coarray Collectives

The Cray compiler recently include some early implementations of new Cray Fortran coarray collective operations, namely co_max, co_min, co_sum and co_bcast. The project did not make use of these collectives as they are not part of the standard.

6.2 Suggestions for Additional Work

6.2.1 Test Application

The distributed parallel quicksort, developed over the course of the project, is a firm base from which many iterative improvements could be made to create a very scalable and efficient sorting algorithm. Some key areas of improvement could be:

1. The early implementations of the new Cray Fortran coarray collectives should be compared with the standard MPI collectives.

2. Using multiple clustered pivot values each iteration. For example, 10 pivot values per iteration clustered around the default pivot value would provide an immediate improvement for little additional performance costs, as having more pivot values with a sensible distribution would allow the sort to choose the pivot which is closest to an image boundary, reducing the average key deficit the scan function has to repay during the splitting of each set. This would greatly improve the overall sort performance for both coarray and MPI implementations, though the MPI version more so as its scan implementation is more costly than the coarray version.

3. The scan function could be improved by using a deeper scan buffer. The initial scan had no buffer to store scanned keys and so required a very large number of point to point communications. The improved version used a buffer to store the received keys meaning following iterations only required a single point to point communication. If the buffer was made several elements deep such that the root process could receive multiple keys from each process during the initial gather, this would once again greatly reduce the number of communications needed for each complete scan and could be used to hide the latency of any further communications within the scan, though the scan would still benefit from single sided communications.

4. Parallelising the scan function. The current implementation is a serial version where the root process performs the scan on its own. It is possible to parallelise the work of the scan function which would speed up the scan time, especially when used in conjunction with the third suggested improvement above. This would probably work best with coarrays as the single sided communications between neighbouring cores sharing memory would be quick and efficient, probably more so than using MPI.

The list of improvements, suggested above, was to be implemented during the project if there had been sufficient time. The three improvements are not trivial changes to the
code and there was insufficient time to implement them here. The suggested improvements would provide some of the more impressive performance enhancements. There are many other potential improvements but they concern less time consuming components of the algorithm and so would not give very significant improvements to the overall sort performance.

6.2.2 MPI & PGAS Comparison

The specific implementation of the application developed within the project favoured the use of single sided communications in a key area of the algorithm. The comparison of MPI and PGAS languages based purely on overall application performance is therefore not a valid one. It is necessary to look into the individual components to understand how they contribute to the overall performance which can then be taken in the correct context. It would be interesting to perform similar studies using very different types of HPC applications like 3D structured grids, sparse/dense linear algebra, N-body simulations etc. Additional interest may be generated by performing the comparison tests with other PGAS languages like UPC, Chapel and X10. Using a range of HPC architectures would also be worthwhile as not all machines will have hardware support for single sided communications. It is quite possible that the codes used in this project run on a different architecture, like a Cray XT4, would see the coarray performance take a substantial drop where as the MPI version may not be as affected.
References


[6] [http://www.hector.ac.uk/cse/documentation/Phase2b/](http://www.hector.ac.uk/cse/documentation/Phase2b/) (referenced 05/08/2011)

[7] [http://www.hector.ac.uk/service/hardware/](http://www.hector.ac.uk/service/hardware/) (referenced 05/08/2011)


Appendix A

Work Plan

Figure 21: Work plan, from the project preparation. The original plan was to implement three sorting algorithms, each one taking four weeks to complete.

The original work plan was to write three sorting algorithms, with four weeks assigned to each one. Each sort had one week assigned to each of the four main development sections, design, coarray implementation, MPI implementation and obtaining results. After the first week implementing the coarray version it became apparent that the implementation of the quicksort was more complex than anticipated and so ran over to two weeks. The MPI version followed quickly though, as the coarray version provided a useful template. During the testing of the coarray and MPI codes the decision was made to take both implementations further as the initial performance was poor, but also the quicksort contained several types of communication patterns which would provide plenty of discussion on its own. The quicksort had plenty of room for improvement and further work was done on the counter, scan, and global swap functions, each of which took several weeks due to the difficulty of implementing more complex versions and the additional testing that was required. A hybrid sort was also developed which took a week to put together and an additional week for testing and results to be completed on it.

The original plan was ultimately flawed for several reasons. Developing reasonably good implementations of distributed parallel sorting algorithms is not a straightforward task. Parallel sorting algorithms are complex and distributed variations even more so. The time allocated to each sort was far too short.
The idea to develop a range of sorts rather than focusing on one was a mistake. A range of sorts was chosen as each would use a different range of communication patterns and so would provide many opportunities for different comparisons of the two parallel APIs. But the quicksort which was implemented used a sufficient number of different communication patterns anyway and in developing it further from the initial working version enabled much greater depth of the results analysis and discussion surrounding the main topic.

The errors in the work plan originate from the inexperience of the author. Before the dissertation no software projects lasting over one or two weeks had ever been undertaken and so any estimate of work load and work time were almost certain to be wildly under or over estimated.

The choices that were made during the project to continue with the quicksort were hopefully justified as the finished code has some good characteristics and performs, in some instances, hundreds of times faster than the initially developed code. Working on one application allowed for closer examination of the code and performance than would have been possible if the time had been spread over three applications. The additional sorts would not have necessarily contributed anymore additional material to the PGAS/MPI discussion than the quicksort already had. They would have mainly contributed to a comparison of the different sorting algorithms which is not the main focus of the project and so little is lost having not done them.

Another benefit from focusing on one sort algorithm was that it was possible to do a hybrid version of the code which would certainly not have happened if all three had been attempted.

Risk Analysis

The risk analysis in the project preparation assumed that the main risk to occur would be an inability to meet deadlines due to difficulties in implementing the sorting algorithms. This turned out to be true but mainly because the work plan was grossly misjudged, as described above.

The original plan had left two weeks free at the end of the project to absorb any risks which occurred, however due to the vast difference in the work that was carried out and that which was planned, when it became apparent that the project would focus on a single application the work plan essentially became a rolling week-by-week assessment of what could be improved upon in the code and running further tests and performance results until time had run out. This work schedule allowed for flexibility of the project to make the most relevant changes each week which resulted in the greatly improved quicksort algorithm. There was no risk of running out of ideas for improvements as the algorithm still contains many good opportunities to improve both MPI and coarray versions, some of which are given in the suggestions for future work section (6.2.1).
A serious risk but with very low chance of occurring was the risk of HECToR going offline for long periods of time. The risk was acknowledged but not prepared for because HECToR is a national facility and is well maintained. The only periods of down time were for regular scheduled maintenance, which were used as opportunities for writing the dissertation report.

Additional Graphs

The following figures are from the final version of the quicksort. They were omitted from the main body of text simply because they show repeated results.

Figure 22: $10^6$ kpc pivot select

Figure 23: $10^4$ kpc pivot select
Figure 24: $10^3$ kpc pivot select

Figure 25: $10^6$ kpc counter

Figure 26: $10^4$ kpc counter
Figure 27: $10^3$ kpc counter

Figure 28: $10^5$ kpc scan

Figure 29: $10^4$ kpc scan
Figure 30: $10^6$ kpc global swap

Figure 31: $10^5$ kpc global swap

Figure 32: $10^4$ kpc global swap
Figure 33: $10^3$ kpc speedup over MPI. The plot shows the relative speedup of CAF and MIX versions over the pure MPI version, executed on 1 to 8 nodes using $10^3$ keys per core. The MIX version is continually better but particularly so on 4 and 8 nodes. This shows nothing in addition to figure 20, and so was not included in the main text.

**Early Results**

The following graphs are from tests taken at intermediate points throughout the development of the quicksort. All significant improvements that were made are noted within the main body of the dissertation.

Figure 34: $10^6$ random kpc sort efficiency. The plot shows the sorting efficiency with size $10^6$ kpc before the addition of the local merge algorithm resulting in a very low overall performance by two orders of magnitude in sorting efficiency. There are also two plots for the MPI and CAF versions, blue for the original scan implementation and red for the improved scan. Here it is clear that the scan is still the key to the performance of the code as the CAF version goes from being the worst on all core counts to being the best. The MIX version at this stage was the best pure MPI and CAF
subroutines integrated together, the mixed subroutines had not been implemented at this stage.

Figure 35: $10^6$ reverse sorted kpc. The plot shows the sorting efficiency when using problem size $10^6$ kpc but importantly the keys are in reverse order. This results in each default pivot splitting the sets directly in half with no deficit until an odd number of cores is reached. This means that the square node numbers have far greater performance because they avoid any scan calls until the penultimate split, as described in section 5.1.2. The reverse sorted plot was useful for checking the correctness of the program during the early stages of development as the observed behaviour was as expected and it gave some preliminary ideas as to which components of the sort were responsible for the main changes in performance. The plots shown here are for the initial version of the quicksort before any improvements had been made. It is worth noting that the $10^6$ kpc random sort for the final version of the quicksort performs better than this special case where hardly any scans or global swaps take place, this is due to the improved local merge algorithm in the final version.