Impact of Co-Array Fortran in FLASH for Cosmological Simulations

Salim Chedrawi

August 19, 2011

M.Sc. in High Performance Computing
The University of Edinburgh
Year of Presentation: 2011
Abstract

The introduction of Co-Array Fortran into FLASH, a modular parallel multi-physics application code which is used for cosmological simulations, is investigated in this report. FLASH is an Adaptive Mesh Refinement (AMR) code, which is parallelised using MPI. Communication overhead is a limiting factor in cosmological simulations. The possible substitution of Co-Array Fortran into MPI routines in FLASH to enhance communication overhead is explored. There is interest because the FLASH center has begun to parallelise FLASH using both OpenMP and MPI. Consequently, there is an importance on easy and flexible parallelisation extensions on code units.

Results indicated that MPI_Allreduce dominated MPI communication overhead for the third Unit Test. As a result, a number of implementations of the AllReduce reduction operation were developed for FLASH. It is found that, employing Co-Array Fortran brings FLASH closer towards even faster cosmological simulations. Moreover, results indicate that larger simulations will be able to run using less resources if Co-Array Fortran is employed. If the message length provided is large enough, the implemented Co-Array Fortran version of MPI_Allreduce is faster than both the implemented OpenMP and MPI versions within a node and, for even larger message lengths, can be faster than MPI between nodes. However, in general, for a fixed message length, it is likely that Co-Array Fortran should be used within a node, and MPI between nodes so that we can run larger simulations and be compatible with future machine architectures. Future potential optimisation strategies are also discussed in this report.
# Contents

Abstract ..................................................................................................................................................... ii

Acknowledgements ....................................................................................................................................... viii

Chapter 1 Introduction .......................................................................................................................... 1

1.1 Overview................................................................................................................................................ 1

1.1.1 Extensions to original work plan ........................................................................................................... 2

1.2 Machine Overview ................................................................................................................................... 2

1.2.1 HECToR Cray XE6 ................................................................................................................................ 2

1.2.2 Intrepid IBM BG/P ............................................................................................................................... 3

1.3 FLASH Overview ..................................................................................................................................... 3

1.4 Co-Array Overview ................................................................................................................................ 4

1.4.1 Co-array versus MPI ............................................................................................................................. 5

1.5 Outline of Sections ................................................................................................................................ 6

Chapter 2 Installation .................................................................................................................................. 7

2.1 FLASH on HECToR ............................................................................................................................... 7

2.2 Use Cases ................................................................................................................................................. 8

2.3 Sedov ....................................................................................................................................................... 8

Chapter 3 Unit Tests ................................................................................................................................... 9

3.1 Profiling with CrayPAT .......................................................................................................................... 9

3.2 Unit Tests ............................................................................................................................................... 10

3.3 Gamma EOS Unit Test .......................................................................................................................... 10

3.3.1 Profiling Gamma EOS Unit Test ........................................................................................................... 12

3.4 Helmholtz EOS Unit Test ...................................................................................................................... 13

3.4.1 Profiling Helmholtz EOS Unit Test ....................................................................................................... 14
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.1</td>
<td>Profiling Maclaurin Spheroid Unit Test</td>
<td>17</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>Collective Communication Development</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Design and Implementation of MPI version</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Design and Implementation of three Co-Array versions</td>
<td>20</td>
</tr>
<tr>
<td>4.3</td>
<td>Design and Implementation of OpenMP version</td>
<td>22</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Results and Analysis</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>Runtime Performance</td>
<td>23</td>
</tr>
<tr>
<td>5.2</td>
<td>HECToR Cray XE6 Simulations</td>
<td>24</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Performance Measurement of three Co-Array versions</td>
<td>24</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Performance Analysis of three Co-Array versions</td>
<td>26</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Performance Measurement of Co-Array v2 with MPI version</td>
<td>27</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Performance Analysis of Co-Array v2 with MPI version</td>
<td>30</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Performance Measurement between Co-Array, MPI, and OpenMP versions</td>
<td>32</td>
</tr>
<tr>
<td>5.2.6</td>
<td>Performance Analysis between Co-Array, MPI, and OpenMP versions</td>
<td>35</td>
</tr>
<tr>
<td>5.3</td>
<td>IBM BG/P Simulations</td>
<td>36</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Performance Measurement of three Co-Array versions</td>
<td>37</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Performance Analysis of three Co-Array versions</td>
<td>40</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Performance Measurement of Co-Array v2 with MPI version</td>
<td>40</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Performance Analysis of Co-Array v2 with MPI version</td>
<td>42</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>Future Optimisations</td>
<td>44</td>
</tr>
<tr>
<td>6.1</td>
<td>Communication Optimisation</td>
<td>44</td>
</tr>
<tr>
<td>Chapter 7</td>
<td>Conclusions</td>
<td>46</td>
</tr>
<tr>
<td>7.1</td>
<td>Conclusion</td>
<td>46</td>
</tr>
<tr>
<td>7.2</td>
<td>Summary of Achievements</td>
<td>48</td>
</tr>
<tr>
<td>7.3</td>
<td>Future Work</td>
<td>48</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>56</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Figure 1</td>
<td>Cray Apprentice2 Screenshot showing the time division of the Gamma EOS Unit Test between its subroutines that consume the most time at runtime</td>
<td>12</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Cray Apprentice2 Screenshot showing the time division of the Helmholtz EOS Unit Test between its subroutines that consume the most time at runtime</td>
<td>14</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Time spent in MPI Routines for Maclaurin Spheroid Unit Test on various problems sizes in FLASH 4-Alpha</td>
<td>16</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Cray Apprentice2 Screenshot showing the time division of the Maclaurin Spheroid Unit Test between its subroutines that consume the most time at runtime and the ones that are most called during execution</td>
<td>17</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Cray Apprentice 2 Screenshot of Call graph tree for Maclaurin Spheroid Unit Test showing which subroutines call MPI_Allreduce and the percentage of total time it takes during runtime</td>
<td>18</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Bar graph showing the percentage of total time spent in the four main MPI routines, MPI_BARRIER, MPI_WAITALL, MPI_SSEND, and MPI_ALLREDUCE</td>
<td>18</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Code snippet for MPI &quot;AllReduce&quot; version</td>
<td>20</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Code snippet showing the random number generation and calling of the Co-Array &quot;AllReduce&quot; subroutine for all three Co-Array versions</td>
<td>20</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Code snippet showing the AllReduce operation of Coarray-v1</td>
<td>21</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Code snippet showing the AllReduce operation of Coarray-v2</td>
<td>21</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Code snippet showing the AllReduce operation of Coarray-v3</td>
<td>22</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Code snippet for showing the implementation of the AllReduce operation in OpenMP</td>
<td>22</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Code Snippet showing the instrumentation to perform the timing for the calling procedure of Allreduce</td>
<td>23</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Runtime performance of the three Co-Array versions on HECToR for a fixed array size of 1000 (Graph is log_{10}-log_{10})</td>
<td>24</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Runtime performance of the three Co-Array versions on HECToR with 16 cores (Graph is log-log)</td>
<td>25</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Runtime performance of Coarray-v1 on HECToR (Graph is log_{10}-log_{10})</td>
<td>25</td>
</tr>
</tbody>
</table>
Figure 37 - Code snippet showing a possible Co-Array implementation for a distributed remote gather operation

Figure 38 - The value of the message length where both the MPI/OpenMP and Coarray-v2 codes take the same execution time. (Graph is log_{10}-log_{10})
Acknowledgements

First of all, I would like to thank my supervisor Dr. Gavin Pringle for every bit of his unprecedented help with this dissertation. He was able to organise with the FLASH team a constructive project. He was always available to answer my questions and assist me in unanticipated dilemmas.

I also thank the FLASH center for their help. In particular, my co-supervisor Dr. Anshu Dubey who organised a resilient project on short notice with all the dynamic changes that happened due to preliminary investigations. Thanks also to Chris Daley who provided very useful feedback and help with particular FLASH issues and problems that arose with the various Unit Tests used in this project. He was always available to answer my various FLASH questions, and has helped immensely with my understanding of the FLASH application. I would also like to thank him for running the experiments on the IBM BG/P in Illinois on short notice.

A special thanks to Dr. Alan Simpson who was available to answer my various Co-Array Fortran questions.

The software used in this work was developed by the DOE-NNSA and NSF-supported FLASH Center for Computational Science at the University of Chicago.
Chapter 1

Introduction

1.1 Overview

The following project investigates the introduction of Co-Array Fortran in the FLASH [2] simulation code. FLASH is a modular, parallel multi-physics code that was developed to study thermonuclear flashes and now computes flow problems in astrophysical environments [2]. It is mainly parallelised by the Message Passing Interface (MPI) [13].

Co-array Fortran [10], on the other hand, is an extension of Fortran 95/2003 for parallel processing which interprets a program that is replicated a fixed number of times to run in parallel. These multiple programs are termed images. The images access remote data using co-arrays. This makes remote variables accessible locally.

The FLASH center has started to add hybrid parallelization to FLASH using MPI and OpenMP. They are investigating novel parallel language paradigms to see if FLASH can be parallelised more efficiently. This dissertation involves the investigation of the introduction of Co-Array into FLASH and its performance compared against the MPI and OpenMP versions. As determining the optimal way to parallelise it is crucial in maximising both the performance and the scale of future cosmological simulations.

Most of the work in this project investigated FLASH in order to find key ways to introduce Co-Array and analyzing whether its introduction would improve performance in parallelization compared to other methods. The subroutines created for collective communication are implemented for FLASH 4-Alpha [19].

FLASH4-Alpha is the most recently released version of the code. Since version 3 FLASH made its architecture more flexible to be extended and easily modified by both internal and external developers. In FLASH4-Alpha, every code unit has its own defined interface with its own inheritance and encapsulation rules that make it simple to update, adjust, and add more functionality. As a result, it includes Unit Test that permit thorough testing and isolation of errors for certain physics problems such as the Gravity Unit that contains two Unit Tests such as Poisson3, which is the Maclaurin spheroid problem, and Poisson3_active, that tests the mapping of particles. The components of the Unit Test, once installed, are apparent in two different places in the source tree.
Moreover, FLASH contains parallelised versions of Unit Tests for both adaptive and uniform grids that have been parallelised using OpenMP. Three Unit Tests were considered, the Gamma EOS Unit Test, Helmholtz EOS Unit Test, and the Maclaurin Spheroid Unit Test.

1.1.1 Extensions to original work plan

The original work plan for this project was to only test MPI and Co-Array Fortran on HECToR using three Unit Tests. However, the original plan was found to be based on a common misconception, namely that mixed-mode codes employing MPI and Co-Array Fortran in the same method as mixed-mode codes employing MPI and OpenMP, specifically each MPI task will contain the same number of CAF images. This common assumption was incorrect: there is, in fact, a direct mapping between MPI Tasks and CAF images, i.e. the introduction of CAF into an MPI code, to produce a mixed-mode code, which means that MPI tasks are mirrored by Co-Array Fortran images.

As a result, the first two Unit Tests were dropped. Nevertheless, extensions to the work plan were adopted, namely a comparison of OpenMP was also included. Thus we compared MPI, Co-Array Fortran, and OpenMP. Timing experiments were also done on another machine apart from HECToR, namely the Intrepid IBM BG/P.

1.2 Machine Overview

The supercomputers used to run and analyze performance of Co-Array, MPI, and OpenMP were HECToR [11], a Cray XE6 located at Edinburgh, UK, and Intrepid [23], an IBM BG/P located at Chicago, US, which are now outlined in the following subsections.

1.2.1 HECToR Cray XE6

The UK's national supercomputing service is called HECToR [11]. It is comprised of the Cray XE6 system. HECToR's system is managed by NAG, EPSRC, Cray, UoE HPCx Ltd, along with EPCC, the leading European centre of expertise in advanced research and supercomputing services at The University of Edinburgh.

The HECToR XE6 system is comprised of 3,712 24-core nodes giving a total of 44,544 cores. However each 24-core node is comprised of two 12-core nodes that share memory. The XE6 system gives a memory footprint and a theoretical performance peak to over 59TB and 370TF, respectively. A 5MB L3 cache is shared between six cores.

The XE6 node consists of 24 cores which are named 'Magny-Cours' and are packaged as 2 AMD Opteron 6172 2.1 GHz processors. They share a total of 32 GB of memory accessible through a Non-Uniform Memory Architecture design. A 'Magny-Cours' processor is basically two hex-core dies connected together within the same socket. Each die has direct access to 8GB memory through 2 memory channels. For each die to access memory connected to other dies, a Hyper Transport link for memory transfer must be used. Each die has a four Hyper Transport link, where each is 16 bits in width.
and one link is split into two 8-bit connections. The XE6 system uses the Cray Gemini Interconnect that has a short latency of around 101.5 microseconds. HECToR's nodes use Cray's UNICOS/lc operating system based on the Compute Node Linux kernel.

1.2.2 Intrepid IBM BG/P

The Blue Gene/P is the second generation of the IBM Blue Gene supercomputer. It was designed through collaboration between IBM, Argonne National Laboratory, and LLNL.

The Intrepid BG/P is found in Argonne National Laboratory southwest of Chicago, Illinois [18]. The laboratory is managed for the United States Department of Energy by UChicago Argonne, LLC, which is composed of Jacobs Engineering Group and the University of Chicago.

Each BG/P rack contains 1024 compute nodes. Each compute node has 4 cores. Each core is a 850MHz PowerPC 450 processor with a dual floating-point unit called the 'double hummer'. Each node has a 2GB memory. Moreover, each I/O node has 64 compute nodes. Intrepid has 40 racks of 163840 cores.

The BG/P architecture allocates the nodes in blocks or partitions. Each partition must have at least one I/O node and are restricted in size to powers of two. On the Intrepid system, the smallest partition allowed is 512 which leads to available partitions 512, 1024, 2048, 4096, 8192, 16384, 24576, 32768, and 40960. The development queue only supports small partition sizes of 64, 128, 256, and 512. Moreover, it is not possible to include a job span over multiple partitions. In addition, the CPU hours charge is four times the partition size since the BG/P node has four cores and it is charged based on partition size and not job size. Additionally, the Blue Gene system uses a torus network which is only fully available at the mid-plane level.

1.3 FLASH Overview

FLASH is a modular, parallel multi-physics Fortran90 simulation code that was developed to study thermonuclear flashes and reactive flow problems in astrophysical environments [2]. The Message Passing Interface (MPI) for inter-process communication is used in order to achieve portability and scalability through parallelization on parallel supercomputers. FLASH uses the HDF5 [4] or PnetCDF library [16] for parallel I/O. FLASH is an Adaptive Mesh Refinement (AMR) code that is based on two interchangeable discretization grids, specifically a uniform grid and a block-structured adaptive grid that uses the PARAMESH library [15]. The latter forms the computational domain that puts the needed resolution elements for computation in regions of high density. "As such, the use of an adaptive grid is important and desirable since a high resolution grid is applied to the regions of the computational domain rather than the need for the extra computational time and storage space of a uniform grid" [2]. Furthermore, the grid is dynamic in which, as the simulation evolves, the domain changes.
The FLASH code is developed by the FLASH Center for Computational Science at the University of Chicago. As described in Section 1.1, since version 3, the FLASH4 code has made its architecture more flexible to be extended and easily modified by internal and external developers. Every code unit has its own defined interface with its own inheritance and encapsulation rules that makes it simple to add more functionality, update, and adjust. The new update that involved the separation of the grid architecture from the actual physics has added flexibility which enables users to switch between an adaptive and a uniform grid. Furthermore, one of the important features found in FLASH4-Alpha is the new software verification and validation test-suite application called FlashTest. It is used to setup, compile, execute, and test a series of FLASH code simulations frequently. The simulations can also be visualized in a browser using FlashTestView.

1.4 Co-Array Overview

A brief introduction is provided to have an idea about the composition and utilization of the Co-Array Fortran programming model.

Co-Array Fortran [10] is an extension of Fortran 95/2003 for parallel processing. It interprets a program that is replicated a fixed number of times. Each replication is termed an image with its own set of data objects. The images are executed asynchronously where the execution path differs between images. Some of the conditions permit images to synchronize. The images access remote data, that is data which reside on other images, using co-arrays. Co-arrays are arrays or scalars that are accessed remotely where every image can access data objects on any other image. The Fortran syntax for declaring a co-array is by specifying a co-dimension for the array or scalar:

```
real, dimension(10), co-dimension[*] :: x.
```

This makes the variable \( x \) remotely accessible. Every co-array has the same size on every image. Both a remote memory and local memory is used. Synchronization is very important in Co-array Fortran, especially since remote data is sent and received. At the program start, there is implicit synchronization. However a barrier for synchronization can be used such as `sync all`, whenever necessary. Usually synchronization is used explicitly before and after accessing co-arrays.

Co-array Fortran has a bottom-up approach to global data where it assembles rather than distributes data. For instance, the programmer can assemble a 2D data structure from a 1D array. He can also arrange images with various grids by using co-arrays with co-dimensions. As such, images are organized into logical grids for the co-array. Therefore co-dimensions create a map so that images can locate the co-arrays of other images by accessing the remote co-array using the grid coordinates. This enables various ways to assemble a multi-dimensional data set.

Fortran arrays are defined by rank, shape, and bounds. In contrast, Co-array Fortran arrays are defined by co-rank, co-shape, and co-bounds. Moreover, co-arrays declared
in procedures must have the save attribute which retains value between procedure calls. Co-array structures give flexibility for non-Co-Array, allocatable and pointer components. Co-arrays can be passed as arguments, can be allocatable, and can be dummy arguments.

In an image's I/O, each image has its own set of input/output units. Units are independent on each image. The default input unit is pre-connected on image 1 only. However the default output unit is available on all images.

Co-arrays are used in various contexts. It can be used as a complete application that uses co-arrays, or mixed with MPI. It can also be used as an incremental addition to serial or to an MPI code. Co-arrays are used for some of the communication since it is simpler and it can overlap communication with computation, which contributes to giving a high parallel efficiency. Moreover, they are used for work-sharing schemes. In collective communication operations, if the implementation does not have MPI collectives and scalability is required to a large number of images, then the use of MPI for the collectives, rather than Co-Array Fortran, is recommended if both MPI and Co-arrays are supported [20]. However, in this report, we present certain cases where Co-Array Fortran performs faster than a particular MPI collective routine.

1.4.1 Co-array versus MPI

Co-array implementations are simpler than MPI implementations. Co-array syntax allows the expression of remote data in a natural way where there is no need for complex explicit communications. Co-array implementation is considered to be orders of magnitude faster than MPI with operations such are remote gather with small number of indices [20]. However this assumption will be contradicative based on our experiments. Co-array structures can be inserted into existing MPI code to improve and simplify communication. The calling tree can be left intact if Fortran pointers are used which simplifies the introduction of co-array variables. Co-arrays can be introduced in specific parts of the code. Moreover, they cannot be pointers, but are allowed to be a derived type with allocatable or pointer components. Nevertheless, Co-Array Fortran is not portable if a compiler is not present, unlike MPI, where users can download a free library. Furthermore, Co-Array Fortran needs users to re-write their code. However, it can be used between nodes, unlike other PGAS languages, such as OpenMP.
1.5 Outline of Sections

Chapter 2 describes the installation, testing, and verification of the FLASH code in the supercomputer.

Chapter 3 describes the analyzed Unit Tests. It includes an explanation of the Unit Tests studied, their installation, profiling, and analysis of their simulation.

Chapter 4 explains and describes the implementation of the various cases created to study the performance of the various parallel programming models in collective communication.

Chapter 5 shows the simulation results of the various cases implemented, namely the original MPI, three alternative yet equivalent CAF codes and an equivalent OpenMP version, to assess their relative performance for a range of processors counts and various message lengths.

Chapter 6 discusses the future optimisations that could be undergone based on the results found in Chapter 5.

Chapter 7 concludes the report with a summary of achievements and evaluation of the project.
Chapter 2

Installation

In order to verify that all the FLASH applications run correctly, some problems with known solutions are distributed with the FLASH code. These were installed and executed. The Sedov sample problem, for example, was installed for FLASH 4-Alpha, and is a simple 2D hydrodynamics simulation with an analytical solution which permits direct tests for correctness.

Section 2.1 describes the first initial steps in unpacking and configuring FLASH 4-Alpha.

Section 2.2 describes the test problem installed for FLASH 4-Alpha. The Sedov problem is used to verify the installation of FLASH 4-Alpha.

In order to install and configure different FLASH problems, a Python script is provided. It is executed as such:

```
./setup problem [-arg1 -arg2 -arg3]
```

where `problem` is the name of the simulation to be run, and `-arg1`, `-arg2`, and `-arg3` are optional arguments to add to the configuration of the simulation problem. Once the script is run, an object directory is created and all the necessary files are copied and linked into it with the appropriate `Makefile.h` that sets the paths for the problem-specific compilers and libraries.

2.1 FLASH on HECToR

The first step in installing FLASH is unpacking the FLASH source file using the command:

```
tar -xvf FLASH.tar
```

Once we enter the created FLASH directory, we need to configure the FLASH source tree for the Sedov explosion problem using the setup script.
2.2 Use Cases

Different Use Cases are used to ensure that the additional progress in development do not establish new errors.

Therefore, in FLASH 4-Alfa, the test cases with problem-specific source files are copied to new specific directories.

Serial and Parallel runs were then executed. In the parallel test cases, the application was run on 1, 2, 4, 8, 16, 32, 64, and 128 processors.

On every run, the application prints out an output file. The output files are then compared against the file from the serial run.

2.3 Sedov

In order to verify the proper installation of FLASH 4-Alfa, a test problem called Sedov is utilized. It engages tracking spherical blast waves in a non-planar symmetrical homogeneous medium from an explosion originating from a point.

The Sedov installation includes a machine specific Makefile for the Cray XE6, that contains the paths to the Fortran90, C, and HDF5 libraries. In order to create the object directory and copy the suitable source files into it, the following setup script is used.

```
./setup Sedov -auto
```

The `flash.par` input files configures the simulation run. HDF5 output files are created every \( t = 0.01 \) seconds until 1000 time steps or \( t = 0.05 \) seconds have been taken, whichever comes first. Moreover, IDL [5] routines that are found within FLASH 4-Alpha are used to analyse the HDF5 files. Its use is simplified using a GUI called `xflash3`. Once the image is generated, it is compared to the one found in the FLASH User Guide [17].

For more information on the detail steps for installation, please see FLASH User Guide on the FLASH website [17].
Chapter 3

Unit Tests

3.1 Profiling with CrayPAT

CrayPAT or Cray Performance Analysis Tool is found on Cray systems, such as HECToR, and used to analyse the performance of programs.

In order for CrayPAT to generate tracing experiments, a tool called `pat_build` logs every specified event. Moreover, it generates asynchronous sampling experiments where the program is polled periodically for general statistics. In tracing experiments, the requested events are constantly monitored giving more detailed information however acquiring more overhead in contrast to sampling experiments, which makes the code slow down, consequently making profilers sometimes report false times.

For better profiling, a sampling experiment should be initially run and then the generated statistics are then employed to know which events to follow through a follow on tracing experiment. The Automatic Profiling Analysis (APA) option is used in this project which automates this process.

In order to visualise the generated data by CrayPAT, Cray Apprentice2 is used which is a performance data visualisation tool.

First, the program is instrumented using `pat_build` to be used for a performance analysis experiment. Then the instrumented program is executed. The resulting data file is then converted to a Cray Apprentice2 data format using `pat_report`. Finally, Cray Apprentice2 will create a variety of interactive graphical reports using the experiment data file thus creating a graphical representation of the CrayPAT report.

In order to do tracing for MPI communication, the flag `-g mpi` is added to `pat_build`. All detail of the program's MPI communication and messages will be logged.

Profiling was also investigated using `gprof`, however it was not able to generate an output log file for the FLASH simulations. It seemed that compiling and linking FLASH with profiling enabled worked correctly, yet the profile data file was not generated once the program was executed on HECToR. Actually, it would run slower
because it would take more time collecting and writing profile data. Moreover, the input and argument that are given when the program is run has a dramatic effect on what the profile information will show. As a result, since FLASH involves a lot of complex subroutines with different inputs and arguments, it could have affected the outputted profile information, therefore, profiling was only performed using CrayPAT.

3.2 Unit Tests

At the start of the project, three Unit Tests were considered: these are the Gamma Equation of State Unit Test, the Helmholtz EOS Unit Test, and the Maclaurin Spheroid Unit Test.

Initial investigations were done for the Gamma and Helmholtz EOS Unit Tests in which their execution and performance were studied using CrayPAT. It was found that the Gamma and Helmholtz EOS Unit Tests were similar in that they ran in serial and did not use MPI communication, unlike the Maclaurin Spheroid Unit Test.

The Equation of State, or EOS, unit executes the equation of state for the nuclear burning and hydrodynamics solvers. The EOS function's interface that operates on a one-dimensional vector is used to find thermodynamic quantities using density, temperate or internal energy, and composition. The Eos_wrapped function found in Eos_wrapped.F90 translates a segment of a block into the data format necessary by the EOS function which it later calls. After the return from the EOS function, the returned data is translated back to the same segment of the block.

In FLASH 4-Alpha, there are four implementations of the EOS unit which are Gamma, Gamma/RHD, Multigamma, and Helmholtz.

The Gamma EOS Unit Test implements a 'perfect-gas equation taking relativistic effects into account.' [17]

The Helmholtz EOS Unit Test implements a 'fast Helmholtz free-energy table interpolation to handle degenerate/relativistic electrons/positrons and includes radiation pressure and ions (via the perfect gas approximation).’ [17]

An equation of state routine is called more than $10^9$ times for a two-dimensional simulation. As for a three-dimensional simulation of stellar phenomena, it is called more than $10^{11}$ times.

The Helmholtz EOS utilizes a table look-up scheme for performance and uses more physics.

3.3 Gamma EOS Unit Test

The Gamma EOS Unit Test is setup with a threaded implementation.
The Unit Test files are put in a new object directory called `eos_gamma_time`.

Once in the directory, the Unit Test is compiled using make by using the PGI compiler. In order to compile using the Cray Compiler, the real variables should be changed to double precision. Therefore in the Makefile.h file, the flags `-r8` and `-i4` were changed to `-s real64` and `-s integer32` respectively. Then the job is submitted to HECToR and ran using:

```bash
./flash4 2>&1 | tee gamma.out
```

By enabling the `timeEosUnitTest` variable, the setup scripts includes the `Eos_unitTest.F90` EOS source file in the object directory which increases the loop count for the EOS calculation to 30,000 times.

For better performance studies, the loop count in `Eos_unitTest.F90` is increased to more than 30,000.

In order to include a different parallel implementation, the `threadWithinBlock` is enabled, which is processed in the `Config` files `source/physics/Eos/EosMain/Config` to conditionally include a different implementation of `Eos_wrapped.F90`.

Most of the OpenMP threading is done in the `Eos_wrapped.F90`. Both Gamma and Helmholtz EOS implementations use the same version of `Eos_wrapped.F90`. Extra `threadprivate` directives to ensure thread safety have been included, which can be viewed by running a search:

```bash
find source -name *.F90 | xargs grep $omp
```

In order to profile the serial version of Gamma and Helmholtz for performance studies, `threadWithinBlock` and `timeEosUnitTest` are set to False.
3.3.1 Profiling Gamma EOS Unit Test

Half of Gamma EOS Unit Test's runtime is found in `amr_initialize_` in `amr_initialize.F90` which constitutes 57% of total runtime. This subroutine initializes the AMR package. It uses PARAMESH, an adaptive mesh library. Since this will not be threaded, then other subroutines should be taken into consideration for analysis and threading.

The `amr_lblk_cc_cp_remote.F90` also uses the PARAMESH adaptive mesh library. The `amr_lblk_cc_cp_remote` routine constitutes 14% of total runtime. This was also not threaded and should not be considered for threading since it uses the PARAMESH library.

`Eos_wrapped.F90` which constitutes 14% of total runtime is the routine that is mostly threaded using OpenMP. Since the EOS Unit Test does not use explicit MPI communication, then this routine is best for threading. The `Eos_wrapped` function acts as a simple wrapper to the EOS interface. The interfaces use a flexible data structure `eosData` to pass thermodynamic quantities in and out of the function.
Simulation_initBlock.F90, which constitutes 14% of total runtime, initializes a 3D domain and uses an EOS call where the resultant pressure and energy are saved. This function could also be threaded.

This routine was not considered for Co-Array Fortran testing since it was found that it did not use MPI communication and would not give significant scaling in threading them. However even though it can be parallelised using OpenMP, Co-Array Fortran introduction would require to dramatically change the entire FLASH code which would not be practical in the short time span of this project.

### 3.4 Helmholtz EOS Unit Test

The Helmholtz EOS Unit Test is setup with a threaded implementation.

```
./setup unitTest/Eos/Helmholtz -auto -3d +ug
threadWithinBlock=True timeEosUnitTest=True +noio
-objdir=eos_helmholtz_time
```

The Unit Test files are put in a new object directory called eos_helmholtz_time. The +ug directive specifies that memory should be allocated statically.

Once in the directory, the Unit Test is compiled using make by using the PGI compiler. As previously done in section 3.3 with the Gamma Unit Test to compile on the Cray Compiler, the real variables should be changed to double precision. Therefore in the Makefile.h file, the flags -r8 and -i4 were changed to -s real64 and -s integer32 respectively. Then the job is submitted to HECToR and ran using:

```
./flash4 2>&1 | tee helmholtz.out
```

The OpenMP threading of the Helmholtz EOS Unit Test is also done the same way in the same files as the Gamma EOS Unit Test.

In order to exercise much more of the Unit Test, the parameter lrefine_max in flash.par can be increased to make a bigger problem.
3.4.1 Profiling Helmholtz EOS Unit Test

![Cray Apprentice2 Screenshot showing the time division of the Helmholtz EOS Unit Test between its subroutines that consume the most time at runtime](image)

Figure 2 - Cray Apprentice2 Screenshot showing the time division of the Helmholtz EOS Unit Test between its subroutines that consume the most time at runtime

Based on Figure 2, 40% of runtime is getc which involves getting cell coordinates. Moreover, it appears that there are no inclusive hits found in subroutine calls within other subroutines.

The Helmholtz EOS Unit Test already has OpenMP directives in Driver_initParallel.F90, eos_vecData.F90, Grid_data.F90, Timers_getSummary.F90, Timers_init.F90, Timers_reset.F90, and Timers_stop.F90

Most of the threading for the threaded version is done in Eos_wrapped.F90.

Helmhotz EOS Unit Test is threaded the same as the Gamma EOS Unit Test except that it uses eos_vecData.F90 which includes some OpenMP directives.

This routine was also not considered for Co-Array Fortran testing since it was also found that it did not use MPI communication like the Gamma EOS Unit Test, and would also not give significant scaling in threading them. However even though it can
also be parallelised using OpenMP. Co-Array Fortran introduction would require to dramatically change the entire FLASH code which would not be practical in the short time span of this project.

3.5 Maclaurin Spheroid Unit Test

The Maclaurin Spheroid Unit Test is a Gravity test problem. It is essentially the Poisson3 Unit Test [17]. The accuracy of gravitational solver is quantifiable since there is an analytical solution for it. The analytical functions express 'the gravitational potential at the surface of, and inside a homogeneous spheroid called a "Maclaurin spheroid"'. Since the spheroid has spherical symmetry then reproducing a solution using multipole expansion is simple [17]. Since an infinite number of multipole moments is required for a non-symmetric solution, the test calculates a solution up to a certain $l_{\text{max}}$(specified in runtime parameter mpole_lmax). As a result an error is generated that is subjugated by the first non-zero term in the expansion's remainder [17]. The solution for points inside the spheroid is the sum of the dipole and monopole moments.

The setup line used to install the test for weak scaling is:

```
./setup  unitTest/Gravity/Poisson3 -auto -3d -maxblocks=60 -opt +noio +pm4dev +newMpole -noclobber -objdir=poisson3_weak +cube16
```

As previously done in sections 3.3 and 3.4, in order to compile using the Cray Compiler, the real variables were changed to double precision. Therefore in the Makefile.h file, the flags -r8 and -i4 were changed to -s real64 and -s integer32 respectively.

Since the Cray Compiler sets its OpenMP on by default, the -h noomp flag was used in the compiler to turn off OpenMP.

There are two possible types of domain decomposition: Uniform Grid and AMR. Uniform Grid employs using a specific grid of blocks which replaces 1 block per processor, while AMR (Adaptive Mesh Refinement) employs using a dynamic grid and which in this case employs managing it using lrefine_max and lrefine_min in the flash.par file.

A weak scaling experiment is run to know an optimal problem size to carry out the essential profiling, where a weak scaling experiment means that the problem size increases with the number of MPI processes and the execution time remains constant, then the code is said to exhibit weak scaling. In contrast, strong scaling is where the problem size if fixed. The weak scaling experiment is achieved by fixing $l_{\text{refine\_min}}=l_{\text{refine\_max}}$ in flash.par in order to have the same average number of leaf blocks per MPI process. They are set to the values 6, 7, 8, 9, 10 for experiments run on 16 cores and profiled using CrayPAT. The simulation is 3-
dimensional. By setting lrefine_min=lrefine_max=6, the refinement will be fixed during the adaptive grid simulation so that the entire grid is uniform and contains no regions of adaptability. The computational domain is then divided into blocks which are distributed between the processors. As such, the scaling properties of the Maclaurin Spheroid Unit Test were investigated. It should be noted that the largest problem that could be investigated was for lrefine_max=lrefine_min=9.

Results for the FLASH application are shown in Figure 3:

![Figure 3 - Time spent in MPI Routines for Maclaurin Spheroid Unit Test on various problem sizes in FLASH 4-Alpha](image)

As a result, the optimum lrefine_max value used was 9 since it gives the longest execution time and the longest time spent in MPI routines. This is desirable as this will give us a bigger problem size in order to be able to profile the Unit Test more efficiently and know which are the most dominant MPI communication bottlenecks. Since timings that result in less than a second would not be robust and accurate as they could be affected by OS flutter and noise. Problem sizes greater than that result in smaller execution times is probably due to memory and cache allocation which resulted in less MPI overhead since less data is communicated between the processes. The percentage spent in MPI routines from total execution time was 18.3% from total execution time. The four main MPI routines used were MPI_BARRIER with 7.90%, MPI_WAITALL with 1.6%, MPI_SSEND with 3.2%, and MPI_ALLREDUCE with 5.6%.
3.5.1 Profiling Maclaurin Spheroid Unit Test

The Maclaurin Spheroid Unit Test, which uses MPI communication was investigated and analyzed for the introduction of Co-Array with a view to employing both MPI and Co-Array Fortran within the same code.

Profiling was undergone using CrayPAT. In Figure 4, generated using Cray Apprentice2, it can be seen the time division of the Unit Test between its subroutines that consume the most time at runtime:

![Figure 4 - Cray Apprentice2 Screenshot showing the time division of the Maclaurin Spheroid Unit Test between its subroutines that consume the most time at runtime and the ones that are most called during execution.](image)

Figure 5 shows two small parts of the entire generated Call Graph Tree found in A.7, which shows the called subroutines and the percentage of total time taken by each. Here, it is shown which subroutines call MPI_Allreduce and the percentage of total time it takes during runtime.
In Figure 6, using the profiling reports generated using CrayPAT, the percentage of total time spent in the four main MPI routines, namely MPI_BARRIER, MPI_WAITALL, MPI_SSEND, and MPI_ALLREDUCE are shown. The timings were the accumulation of time from all the subroutines, and not just the ones mentioned previously.

Based on Figures 4, 5, and 6, the measurements clearly show that MPI_Allreduce which is used over the multipole moment arrays and found in both the mpi_amr_comm_setup and amr_refine_derefine_routines, is the dominant operation.
Chapter 4

Collective Communication Development

Rather than introducing Co-Array Fortran into FLASH, it was decided that, as MPI_Allreduce is the most expensive MPI communication routine in the Maclaurin Spheroid Unit Test, a small simple MPI code test using MPI_Allreduce was to be written which mimics the use of MPI_Allreduce within FLASH. Then, various Co-Array versions, and a simple OpenMP version would be written to perform exactly the same operation as carried out by the MPI_Allreduce routine.

The reduction operation MPI_Allreduce combines values from all processes and stores the result back to all of them. The implementation of Allreduce to find the maximum value, is achieved in three different parallel programming languages, namely Co-Array Fortran, MPI, and OpenMP. In the first stage, a MPI version was implemented that uses the MPI_Allreduce function found in the mpif.h library. In the second stage, three distinct versions of the "Allreduce" reduction operation were designed and implemented in Co-Array Fortran. Performance communication patterns are taken into account in order to optimise and balance the communication between images. The third stage involved producing an OpenMP version that performs an Allreduce reduction operation between threads on an array.

4.1 Design and Implementation of MPI version

In the main file of the MPI code, found in Section A.1, the integer array, max_local, which is the send buffer and the maximum value array, max_global, which is the receive buffer are declared as variable allocatable arrays. The size of the array, array_size, is introduced by setting it as a command line argument at runtime. In order to initialize the integer array, random integers are generated using the random_number() subroutine. Then MPI_Allreduce is called several times in a loop of 100,000 iterations. The extent of this loop was chosen to ensure that the execution time is large enough in order for the experiments not be affected by OS flutter and yet small enough for them to not waste resources. MPI_MAX is set in the MPI_Allreduce parameter in order to find the maximum value in the array between all MPI tasks. Only the 100,000 iterations are only timed, thus any allocation of memory and/or initialisation of arrays are not included in the execution times.
Then the first image is accessed, which performs two loops. The first loop iterates content is not updated from remote images before it is used, local to that image function accessible. In order to find the maximum value of an array, the intrinsic function maxval() is used which is then assigned to the maximum value parameter.

```fortran
start = MPI_WTIME()
  do i = 1, max_iter
    call MPI_Allreduce(max_local, max_global, array_size,
                      MPI_INT, MPI_MAX, MPI_COMM_WORLD, ierror)
  end do

end = MPI_WTIME()
time = end - start
```

**Figure 7 - Code snippet for MPI "AllReduce" version**

### 4.2 Design and Implementation of three Co-Array versions

All three Co-Array versions have a main file, presented in Section A.2 that calls the three alternative Allreduce subroutines. In the main file, the Co-Array integer array and the maximum value Co-Array are both declared using co-dimensions. The size of the array is introduced by setting it as a command line argument at runtime, as per the MPI case above. In order to initialize the integer array, random integers are generated using the intrinsic function random_number(). Then the Allreduce subroutine is implemented several times in a loop of 100,000 iterations to match the MPI implementation above. As with the MPI case, only the 100,000 iterations are only timed, thus any allocation of memory and/or initialisation of arrays are not included in the execution times.

```fortran
... integer, allocatable, dimension(:) :: v
integer :: vmax[*]
integer :: array_size
...
call random_seed
call random_number(x)
v = INT(100*x)
start = MPI_WTIME()
  do i = 1, max_iterations
    call allreduce_max_allget(v, vmax, array_size)
  end do
end = MPI_WTIME()
time = end - start
```

**Figure 8 - Code snippet showing the random number generation and calling of the Co-Array "AllReduce" subroutine for all three Co-Array versions**

In the first Co-Array version, namely Co-Array-v1 found in Section A.3, the allreduce_max_allget subroutine is implemented requiring three parameters, the Co-Array integer array, the maximum value Co-Array, and the array size. Specifying a co-dimension for the array declares a Co-Array which makes it remotely accessible. In order to find the maximum value of an array, the intrinsic Fortran function maxval() is used which is then assigned to the maximum value parameter local to that image. Then to ensure that all images set the local maximum and that all content is not updated from remote images before it is used, sync all is applied. Then the first image is accessed, which performs two loops. The first loop iterates...
through the number of images to find the maximum value across the images by using the Fortran intrinsic function max(). The local maximum of other images are accessed through the co-dimension of the array. Then once the loop is done and the maximum value is found, the first image enters a second loop that also iterates through the number of images, wherein the result, namely the maximum value is sent to all other images. This is implemented by accessing the different local maximum values using its Co-dimension. Finally sync all is used to synchronise after accessing Co-Arrays which ensures all new content is available to all the images.

```fortran
vmax = maxval(v)
sync all
if (this_image()==1) then
  do i = 2,num_images()
    vmax=max(vmax,vmax[i])
  end do
  do i = 2, num_images()
    vmax[i]=vmax
  end do
end if
sync all
```

**Figure 9 - Code snippet showing the AllReduce operation of Coarray-v1**

The second Co-Array version, namely Co-Array-v2 found in Section A.4, is simplified using better communication patterns in order to avoid congestion on the network such as the serial bottlenecks of v1. In this optimized version, all images get data from others simultaneously but this is optimized so communication is more balanced as will be further explained in the loop divisions. As with v1, the maxval() subroutine is used to find the maximum value of the array. Then the result is set to the local maximum value. Sync all is applied to ensure all images set local maximum. Then two loops are then executed. The first loop iterates through the images that are numbered larger than the current image. The second loop iterates through the number of images that are numbered less than the current image. Both loops find the maximum value between the local image's maximum value and each of the remote image's maximum values, and the local maximum value to the end result. This form of iteration creates a more balanced communication pattern and makes the Allreduce operation more optimized as will be shown in the results and analysis in Chapter 5. Finally, a sync all is then used to synchronise after accessing Co-Arrays which ensures the new content is available to all images.

```fortran
vmax = maxval(v)
sync all
  do i = this_image()+1,num_images()
    vmax=max(vmax,vmax[i])
  end do
  do i = 1,this_image()-1
    vmax=max(vmax,vmax[i])
  end do
sync all
```

**Figure 10 - Code snippet showing the AllReduce operation of Coarray-v2**
The third Co-Array version Co-Array-v3 found in Section A.5, is an optimised version of Coarray-v2 above, however with added communication imbalance. In this implementation, all images also get data from others simultaneously by properly synchronizing and assigning values. Once again, the maxval() subroutine is used to find the maximum value of the array, the result is set to the local maximum value, and finally Sync all is applied to ensure all images set local maximum. Version 3 then enters a loop which iterates through the number of images to find the maximum value between all the local image's maximum and all the remote images in turn. A sync all is then used to synchronise after accessing Co-Arrays which ensures the maximum value is available to all images.

```
vmax = maxval(v)
sync all
  do i = 1,num_images()
    vmax=max(vmax,vmax[i])
  end do
sync all
```

Figure 11 - Code snippet showing the AllReduce operation of Coarray-v3

4.3 Design and Implementation of OpenMP version

In the main file, found in Section A.6, the integer array is declared as a variable allocatable array. As with the previous codes, the size of the array is introduced by setting it as a command line argument at runtime. In order to initialize the integer array, random integers are generated using the random_number() subroutine. Once again, in order to find the maximum value of an array, the subroutines max() is used to find the maximum value between the local maximum value and the local value in the array, it is enclosed in a loop that iterates through all the array values in order to enclose the loop with the omp parallel directive to distribute the tasks to different threads in order to parallelise it, which is executed several times in a loop of 100,000. The 100,000 iterations are only timed, thus any allocation of memory and/or initialisation of arrays are not included in the execution times.

```
start = OMP_GET_WTIME()
do j = 1, max_iterations
  !$omp parallel do default(none) shared(max_local,
    max_global,n) private(i)
do i = 2, n
    max_global = max(max_global, max_local(i))
end do
  !$omp end parallel do
end do
end = OMP_GET_WTIME()
time = end - start
```

Figure 12 - Code snippet for showing the implementation of the AllReduce operation in OpenMP
Chapter 5

Results and Analysis

5.1 Runtime Performance

The runtime of the five Allreduce implementations are measured using the standard MPI timer MPI_Wtime. This is done by inserting a call to MPI_Wtime before and after the call to the Allreduce subroutine. The 100,000 iterations are only timed, thus any allocation of memory, and/or initialisation of arrays, and I/O are not included in the execution times calculated. The MPI_Wtime returns a double precision value which contains the elapsed time in seconds. The following table shows the instrumentation to perform the timing for the calling procedure of Allreduce.

```
integer :: i
DOUBLE PRECISION :: start, end, time
integer, parameter :: max_iterations = 100000

start = MPI_WTIME()
do i = 1, max_iterations
   call allreduce_max_allget(...)
end do
end = MPI_WTIME()
time = end - start
```

Figure 13 - Code Snippet showing the instrumentation to perform the timing for the calling procedure of Allreduce

The files are compiled to -O3 optimisation.

First, the three Co-Array Fortran versions of the Allreduce operation were analyzed and their performance compared to each other in order to determine the optimal implementation with potentially less communication and synchronization bottlenecks. Then the performance was measured against the MPI and OpenMP versions of the Allreduce operation. All the tests were performed three times each, and the minimum execution time recorded, to ensure that the results were not too degraded due to the nature of sharing the interconnect in HECToR.
5.2 HECToR Cray XE6 Simulations

5.2.1 Performance Measurement of three Co-Array versions

The three Co-Array versions v1, v2, and v3 are simulated. The experiments are run using 1, 2, 4, 8, 16, 32, 64, and 128 images and a fixed array size of 1000. The results are shown in Figure 14:

![Graph](attachment:image.png)

**Figure 14 - Runtime performance of the three Co-Array versions on HECToR for a fixed array size of 1000 (Graph is log₁₀-log₁₀)**

From these results, we can see that all three versions perform almost the same when employing a single HECToR node. However, Co-Array-v2 performs faster than the other versions as the number of cores increase to 32 cores and beyond, where the interconnect comes into play.

Another experiment was undergone to study the effect of the message length against the execution time. The array size of the send buffer is varied from 10 to 100, 1000, 5000, 8000, 10000, 20000, and 100000 with 16 cores. It was run at 16 cores in order to have communication within a node and not have interconnect bottlenecks between nodes. The results are shown in Figure 15:
From these results we can see that the array size did not have a significant effect between the three version's execution times. Even though Co-Array-v2 was able to execute faster the Co-Array-v3 with the scaling of a larger number of cores since it was optimized for more balanced communication between the images, all three versions are affected the same by the message length.

A third experiment was performed to study the effect of the message length against the scaling of the number of cores for the three Co-Array versions. The array size is varied from 1000 to 2000, 5000, 7000, and 10000. The results are shown in Figure 16, 17, and 18 for the respective implementation versions:
From these results we can see that while the execution time remains fairly constant within a node for the three implementations, it increases after more than 24-cores, since any communication is no longer performed using shared-memory preferences, but is performed using the slower communication over the interconnect.

5.2.2 Performance Analysis of three Co-Array versions

Based on the results, all three versions perform comparatively the same when employing a single HECToR node; however, Co-Array-v2 performs faster than the
other versions as the number of cores increase to 32 cores and beyond, where the interconnect comes into play.

The array size did not have a significant effect on the three version's execution times. Both the versions v2 and v3 were able to outperform v1 and handle the scaling of a larger number of cores. This is due to the optimization where all images get data from others simultaneously by properly synchronizing and assigning values. However Co-Array-v2 was able to execute faster the Co-Array-v3 since it was optimized for more balanced communication between the images.

Moreover, based on the scaling of the number of cores along with the array size, the results indicate that while the execution time remains fairly constant within a node for the three implementations, it increases after more than 24-cores, as any communication is no longer performed using shared-memory preferences, but is performed using the slower communication over the interconnect.

Nevertheless, the execution times when using various array sizes indicate that the performance of the three Co-Array versions are not widely affected by the message length since the run time is fairly the same with array size increase which shows that Co-Array can handle large simulations. As a result, all three versions all have the same characteristic when it comes to scaling the array size. However, v2 performs faster against v3 when it comes to scaling against number of cores, and especially v1 that does not scale well with large number of nodes.

Now that the optimal Co-Array implementation has been established, namely v2, its performance is now compared to the MPI and OpenMP versions of the Allreduce reduction operation.

5.2.3 Performance Measurement of Co-Array v2 with MPI version

The MPI version for the MPI_Allreduce simulation is now compared against the Co-Array-v2 version. The results were repeated three times and the minimum execution time recorded. The experiment was performed to study the effect of the message length against the scaling of the number of cores for the MPI version. The array size was varied against 1000 to 2000, 5000, 7000, 10000. The number of cores was varied between 1, 2, 4, 8, 16, 32, 64, and 128 MPI Tasks. The results are shown in Figure 19:
From these results we can see that that while varying the array size in the Co-Array versions, they all converge towards the same execution time with the same number of cores respectively. However, in the MPI version, the execution times for each array size, while varying the number of cores, are distinctly increasing values that are separate from each other. So it can be inferred that with higher array sizes, execution time in the MPI version is widely affected unlike the Co-Array versions.

A second experiment was performed to study the effect of the scaling of the number of cores between the MPI and Co-Array versions. The experiment was run against 1, 2, 4, 8, 16, 32, 64, and 128 MPI Tasks using an array of size 1000. The results are shown in Figure 20:
From these results we can see that, within nodes, where HECToR is between 8 and 16 cores per node, the Co-Array version performs faster than the MPI version. However, as the number of cores increases past 24 (the number of cores per node) and intra-node communication starts to dominate, the MPI version performs much faster than the Co-Array implementation.

Another experiment was undergone to study the effect of the message length against the execution time. The array size of the send buffer was varied from 10 to 100, 1000, 5000, 8000, 10000, 20000, and 100000 with a fixed number of 16 cores. The results are shown in Figure 21:
From these results we can see that for smaller array sizes, the performance of Co-Array while constant, is much slower than the MPI version. However, as the array size increases dramatically, the performance of the Co-Array version outperforms the MPI implementation.

5.2.4 Performance Analysis of Co-Array v2 with MPI version

A final experiment was undergone to determine if MPI or Coarray-v2 should be employed for a given number of cores and message size. This can be found by locating the point of intersection of the message length against the execution time plots between the MPI and Coarray-v2 versions for various core counts. The array size of the send buffer was varied from 1000 to 2000, 5000, 7000, and 10000. The number of cores was varied between 1, 2, 4, 8, 16, 32, 64, and 128 MPI Tasks. The results are shown in Figure 22.

![Figure 22 - Runtime performance between the MPI and Coarray-v2 versions on HECToR (Graph is log-log10)](image)

The various points of intersection, where both the MPI and Coarray-v2 versions take the same time to run, were taken and Figure 23 was produced:
Based on Figure 22, it can be inferred that before the intersection point, the MPI version performs much faster than Coarray-v2. However, after the intersection point, the MPI version becomes much slower dramatically, while Coarray-v2's execution time fairly increases, maintaining a constant robust performance.

Figure 23 contains the size of the message where the time for the MPI code and the time for the Coarray-v2 code is the same, for a given number of cores. Thus, if a simulation requires a specific number of cores and employs a specific message size, and these correspond to a point below the curve in Figure 23, then the programmer should employ MPI, otherwise the programmer should employ Coarray-v2.

It is important to note that the MPI code is unable to run for very high array sizes due to the MPI library's own internal memory requirements. Increasing the buffer, by using MPI_Buffer_attach(), would solve this issue; however, this would mean the user would have less memory for arrays which would, in turn, force a smaller simulations. Furthermore, based on scaling the number of cores with the array size, the results indicate that while varying the array size in the Co-Array versions, they all converge towards the same execution time with the same number of cores respectively. However, in the MPI version, the execution times for each array size, while varying the number of cores, are distinctly increasing values that are separate from each other. As a result, it can be inferred that with higher array sizes, execution time in the MPI version is widely affected unlike the Co-Array versions.

As a result, it can be inferred, that for larger array sizes, Co-Array Fortran will run while MPI fails. Moreover, as MPI uses much more memory than Co-Array Fortran, then scaling to large number of nodes will tend to lower the performance, especially since, in the future, we expect that nodes will contain 100s or even 1000s of cores. In particular, MPI performs slower as the message length increases while Co-Array is
influenced less. As such, larger simulations will be able to run using fewer resources if Co-Array Fortran is employed.

5.2.5 Performance Measurement between Co-Array, MPI, and OpenMP versions

The OpenMP version for the Allreduce simulation is simulated against the Co-Array-v2 and MPI version. This experiment was performed to study the effect of the message length against the scaling of the number of cores for the OpenMP version. The array size was varied from 1000 to 2000, 5000, 7000, and 10000. The number of threads were varied between 1, 2, 4, 8, and 16 cores. The results are shown in Figure 24:

**Figure 24 - Runtime performance of OpenMP version on HECToR (Graph is log_{10}-log_{10})**

From these results we can see that with higher array sizes, execution time increases in the OpenMP version which shows that it is widely affected by the message size, unlike the Co-Array version which shows that Co-Array Fortran is more suited to larger array sizes.

The second experiment was run against 1, 2, 4, 8, and 16 OpenMP threads with an array size of 1000. The results are shown in Figure 25:
Figure 25 - Runtime performance between the OpenMP, MPI, and Coarray-v2 versions on HECToR (Graph is log₁₀-log₁₀)

From this experiment, we can see that the OpenMP version performs much faster than both the Co-Array and MPI versions; however, it is limited to the number of cores within each node so it cannot scale as far as Co-Array.

Another experiment was undergone to study the effect of the message length against the execution time. The array size is varied from 10 to 100, 1000, 5000, 8000, 10000, 20000, and 100000 with a fixed number of 16 cores. The results are shown in Figure 26:

Figure 26 - Runtime performance between the OpenMP, MPI, and Coarray-v2 versions on HECToR with 16 cores (Graph is log₁₀-log₁₀)

From this experiment, we can see for small array sizes, OpenMP performs much faster than both the Co-Array and MPI implementations. However, as the array size increases
dramatically, the performance of the Co-Array version outperforms the OpenMP and MPI implementations. Since as previously stated, MPI cannot run for very high array sizes.

A final experiment was undergone to determine if OpenMP or Coarray-v2 should be employed for a given number of cores and message size. This can be found by locating the point of intersection of the message length against the execution time plots between the OpenMP and Coarray-v2 versions for various core counts. The array size of the send buffer was varied from 1000 to 2000, 5000, 7000, and 10000. The number of cores was varied between 1, 2, 4, 8, and 16 threads. The results are shown in Figure 27.

![Figure 27 - Runtime performance between the OpenMP and Coarray-v2 versions on HECToR (Graph is log-log)](image)

The various points of intersection, where both the MPI and Coarray-v2 versions take the same time to run, were taken and Figure 28 was produced:
Based on Figure 27, it can be inferred that before the intersection point, the OpenMP version performs much faster than Coarray-v2. However, after the intersection point, the OpenMP version becomes much slower dramatically, while Coarray-v2's execution time fairly increases, maintaining a constant robust performance.

Figure 28 contains the size of the message where the time for the OpenMP code and the time for the Coarray-v2 code is the same, for a given number of cores. As a result, if a simulation requires a specific number of cores and employs a specific message size, and these correspond to a point below the curve in Figure 28, then the programmer should employ OpenMP, otherwise the programmer should employ Coarray-v2.

5.2.6 Performance Analysis between Co-Array, MPI, and OpenMP versions

The scaling over the message length indicates that for small array sizes, OpenMP performs much faster than both the Co-Array and MPI implementations. However, as the array size increases, the performance of the Co-Array version outperforms both the OpenMP and MPI implementations. (Indeed, MPI cannot run for very high array sizes, as previously stated). Furthermore, based on scaling the number of cores with the array size, the results indicate that with higher array sizes, execution time in the OpenMP version is widely affected by the message size, unlike the Co-Array version which shows that Co-Array Fortran is more suited to larger array sizes.

Furthermore, we can deduce, for certain intersection points where message length versus execution time meet for various core counts for both the OpenMP and Coarray-v2 plots, that before the intersection point the OpenMP version performs much faster than Coarray-v2. However, after the intersection point, the OpenMP version becomes much slower dramatically, while Coarray-v2's execution time fairly increases,
maintaining a constant robust performance. Thus providing points of reference to know when to use OpenMP, MPI, or Co-Array.

As a result, based on the experiments, users of HECToR can employ Co-Array Fortran within a node and MPI between nodes, for large message sizes, so that they can run larger simulations and be compatible with future machine architectures that will handle more cores per node.

5.3 IBM BG/P Simulations

Chris Daley of the FLASH Center kindly ran the codes on their IBM BG/P, namely Intrepid. Daley found that some predicaments had to be confronted in order to solve and run the simulations. One of the issues was that, whilst the GNU f95 supports Co-Array Fortran on some platforms, Co-Arrays are not supported on PowerPC (PPC) architectures [21]. An alternative solution was to use the Rice CAF [22] compiler which has pre-requisite packages such as Boost, Rose, GASNet, and Java which had to be installed on the Intrepid BG/P. Daley kindly compiled Boost and Rose for the login nodes and GASNet, and Co-Array Fortran were cross-compiled for compute nodes, where he found that GASNet required gcc higher or equal than v4.3 while Co-Array Fortran required gfortran higher or equal than v4.4. The gfortran requirement could not be satisfied since Intrepid has gcc/gfortran-4.1.2 and an experimental gcc/gfortran-4.3.2. As a result xlc/xlf were used for C and Fortran compilers by Daley. Compilation problems arose since the Rose source-source translation phase produced invalid code for the xlf compiler since the Rice Co-array Fortran compiler has only been tested with gfortran and pgf90 and not xlf. As a result, Daley employed a workaround; a compiler wrapper script was produced where the Co-Array Fortran script thinks it is the Fortran compiler. This fixes the invalid source code in the intermediate Fortran files during the build process. The wrapper script then invoked the actual Fortran compiler with the arguments which were passed by the Co-Array Fortran script. The .f90 extensions were changed to .caf in order to achieve source-source translation. Moreover, since the Co-Array image array was not permitted as a stack variable then a module-scoped Co-Array image array was used in the allreduce subroutine. sync all has not been implemented, therefore it was replaced with a team_barrier(). Further, num_images() and this_image() were also not implemented and were replaced with team_size() and team_rank().

Job runs were submitted by Daley in SMP mode with one application process per node (1 active core per node) since GASNet and DCMF do not support other modes. Therefore communications will not take advantage of any shared memory optimizations. All of the experiments used binaries compiled and linked with -03 -qarch=450. An aggressively compiled binary using -04 -qarch=450 failed at runtime.
5.3.1 Performance Measurement of three Co-Array versions

The three Co-Array versions v1, v2, and v3 were simulated. The experiments are run against 1, 2, 4, 8, 16, 32, 64, and 128 images with an array size of 1000. The results are shown in Figure 29:

![Number of Cores vs. Execution Time](image)

**Figure 29 - Runtime performance of the three Co-Array versions on Intrepid with an array size of 1000 (Graph is log_{10}-log_{10})**

This experiment shows us that v2 and v3 were able to outperform v1 while scaling to a larger number of cores. However unlike HECToR, v2 and v3 almost have the same performance.

Another experiment was undergone to study the effect of the message length against the execution time. The array size of the send buffer is varied from 1000 to 2000, 5000, 7000, and 10000 with a fixed number of 4 cores. The results are shown in Figure 30:
From this experiment we can see that all three versions' execution time increase slightly with the message length. The array sizes did have an effect on the three version's execution times, unlike HECToR.

A third experiment was performed to study the effect of the message length against the scaling of the number of cores for the three Co-Array versions. The array size is varied from 1000 to 2000, 5000, 7000, and 10000. The results are shown in Figures 31, 32, and 33 for the respective implementation versions:
From these tests, we can see that the execution time is unequal at the beginning especially within a node (up to 4 cores) for the three implementations. It increases but the plots start converging after increasing the number of nodes thus increasing the communication overhead. The change in array sizes show that the three Co-Array versions eventually converge in execution time by the number of cores with any message length which shows that the array sizes do not have a significant effect with scaling.

**Figure 32** - Runtime performance of Coarray-v2 on Intrepid (Graph is log10-log10)

**Figure 33** - Runtime performance of Coarray-v3 on Intrepid (Graph is log10-log10)
5.3.2 Performance Analysis of three Co-Array versions

Based on the BG/P results, Co-Array-v2 and Co-Array-v3 have comparable performance. The array sizes did have a significant effect on the three version's execution times, unlike HECToR, however v2 and v3 were able to outperform v1 and scale to a larger number of cores. Moreover, based on the scaling of the number of cores along with the array size, the results indicate that the execution time is unequal at the beginning especially within a node (up to 4 cores) for the three implementations. It increases but the plots start converging after increasing the number of nodes thus increasing the communication overhead. Nevertheless, the change in array sizes indicate that the three Co-Array versions eventually converge in execution time by the number of cores with any message length which shows that the array sizes do not have a significant effect with scaling.

Now we compare the performance of Coarray-v2 with the MPI version of the Allreduce reduction operation.

5.3.3 Performance Measurement of Co-Array v2 with MPI version

The MPI version for the MPI_Allreduce simulation was executed by Daley on the BG/P. The experiment was performed to study the effect of the message length against the scaling of the number of cores for the MPI version. The array size was varied against 1000 to 2000, 5000, 7000, and 10000. The number of cores was varied against 1, 2, 4, 8, 16, 32, 64, and 128 MPI Tasks. The results are shown in Figure 34:

![Figure 34 - Runtime performance of MPI version on Intrepid (Graph is log10-log10)](image_url)

From this experiment, we can see that with higher array sizes, execution time in the MPI version increases unlike the Co-Array Fortran versions, where the execution times converge towards a constant regardless of message length.
A second experiment was run against 1, 2, 4, 8, 16, 32, 64, and 128 MPI Tasks using an array size of 1000. The results are shown in Figure 35:

![Number of Cores vs. Execution Time](image)

Figure 35 - Runtime performance between the MPI and Coarray-v2 versions on Intrepid (Graph is \(\log_{10}\log_{10}\))

From the results of this experiment, we can see that Co-Array Fortran performed faster than MPI for 2 and 4 nodes, i.e. within a node. However, its performance significantly degraded and MPI became much faster when communication between nodes initiated. Therefore within inter-node communication, which for this BG/P is 4 cores per node, the Co-Array version performs faster than the MPI version. However, as the number of cores increases and intra-node communication initiates, the MPI version performs much faster than the Co-Array implementation.

Another experiment was undergone to study the effect of the message length against the execution time. The array size of the send buffer is varied from 1000 to 2000, 5000, 7000, and 10000 with a fixed number of 4 cores. The results are shown in Figure 36:
From this experiment, we can see that Co-Array Fortran performs faster than MPI even though the array size increases dramatically, the performance of the Co-Array version outperforms the MPI implementation.

5.3.4 Performance Analysis of Co-Array v2 with MPI version

Based on the BG/P results, the scaling in the number of cores show that within internode communication, which for this BG/P is 4 cores per node, the Co-Array version performs faster than the MPI version for large message sizes. However, as the number of cores increases and intra-node communication initiates, the MPI version performs much faster than the Co-Array implementation. Furthermore, the scaling in the message length indicates that Co-Array Fortran performs faster than MPI even though the array size increases dramatically, the performance of the Co-Array version outperforms the MPI implementation.

Moreover, based on scaling the number of cores with the array size, the results indicate that with higher array sizes, execution time in the MPI version increases unlike the Co-Array Fortran versions, where the execution times converge towards a constant regardless of message length.
As a result, Co-Array Fortran will perform slower than MPI when scaling to a large number of nodes, though we showed that both Co-Array and MPI were influenced by the message lengths. However, MPI performed slowly with large message lengths when compared to Co-Array Fortran, while Co-Array Fortran performed faster and especially while communicating within a node. Co-Array Fortran codes, all running between and within nodes, eventually attained the same performance, regardless of message length; however, the execution time is constant within nodes while it increases dramatically between nodes. In particular, since nodes will be able to hold more cores, thus increasing inter-node communication, then Co-Array Fortran will have a better performance than MPI when performing an AllReduce collective communication. As such, larger simulations will be able to run using less resource if Co-Array Fortran is employed.
Chapter 6

Future Optimisations

The profiles for the Maclaurin Spheroid Unit Test indicate that MPI_Allreduce dominates the MPI communication. To further improve the runtime performance, potential communication optimisations are discussed in section 6.1.

6.1 Communication Optimisation

The results found in section 3.4 show that MPI_Allreduce dominates the MPI communication time. However, the subsequent MPI calls that create a significant overhead are the MPI_BARRIER and MPI_SSEND calls, in order of severity.

Since one of the optimisations involves replacing the MPI_Allreduce call with a Co-Array "AllReduce" call, then the MPI communication might be additionally improved by replacing the point-to-point communications such as MPI_SSEND by Co-Array Fortran, which represented around 3.2% of total runtime with a simulation of lrefine_max=9. Most of the MPI_SSEND overhead is found in mpi_xchange_blocks which is part of the mpi_amr_comm_setup routine. It involves around 1.68% of total runtime. Replacing this code with Co-Array Fortran would create an opportunity to express the communication much more simply and give the compiler the possibility to overlap communication with calculation. Nevertheless, replacing the MPI operations with Co-Array Fortran involves changing the original code and will require vast effort in large industry codes. Even so, one of the techniques in introducing Co-Array into FLASH is placing relevant arrays in global scope (modules) in order to avoid multiple declarations [20].

Further optimisation can be improved by optimising other collective operations. Co-Array Fortran can be good for collective operations when there are irregular communication patterns that do not have a counterpart within the set of available MPI collectives. Additionally, Co-Array Fortran should be used when there is an opportunity to overlap communication with computation [20].
For instance, a distributed remote gather can be used in poor load balanced scenarios with irregular communication patterns that could be found in FLASH. As such, the following Co-Array Fortran implementation for GET can be appropriate.

```fortran
INTEGER :: index(num_elts)
REAL :: table_caf(n), buffer_caf(num_elts) !num_elts << n

IF (this_image() .eq. 1) THEN
!Image 1 gets values from all the other images
  DO i=1,num_elts
    pe = (index(i)-1) / nloc + 1
    offset_var = MOD(index(i)-1, nloc) + 1
    buffer_caf (i) = table_caf(offset_var)[pe]
  ENDDO
ENDIF
```

Figure 37 - Code snippet showing a possible Co-Array implementation for a distributed remote gather operation

The array table is distributed across the images, while the index and buffer are replicated.

Therefore, future analysis could be done on other Unit Tests in FLASH to detect irregular communication patterns with MPI and then study the effect of the distributed remote gather on poorly load balanced scenarios which could replace relevant MPI routines.
Chapter 7

Conclusions

7.1 Conclusion

In this dissertation, we present the investigation of the possible introduction of Co-Array Fortran into FLASH. Three Unit Tests were considered. The initial investigations led to the conclusion that threading the Gamma and Helmholtz EOS Unit Tests using Co-Array Fortran was not suitable since they did not include MPI communication and would not give significant scaling by threading them. However, the Maclaurin Spheroid Unit Test was the one used to determine the performance impact due to the insertion of Co-Array Fortran.

It was found through thorough profiling using CrayPAT of the Maclaurin Spheroid Unit Test that the MPI_Allreduce operation dominates the MPI communication. The implementation of various versions of the operation in Co-Array performed in the desired way. Several test cases were undergone, and the results were compared against MPI and OpenMP versions of the AllReduce operation.

The Co-Array implementations v2 and v3 outperformed v1 and were able to handle a larger number of cores. This was due to the communication optimization which enabled all images to get data from others simultaneously. Moreover, Co-Array-v2 was able to execute faster the Co-Array-v3 as it was optimized for more balanced communication between the images.

It can also be concluded that the three Co-Array Fortran versions were not extensively affected by the variation of the message length since the run time was comparatively the same with array size increase for all three cases. The execution time converged for all the array size variations with respect to the number of cores. This demonstrated that Co-Array Fortran can handle large simulations.

In addition, it can be concluded, that for certain high array sizes, Co-Array Fortran will run while MPI fails due to lack of memory. Furthermore, MPI performs slower as the message length increases while Co-Array Fortran is influenced less. Moreover, as MPI uses much more memory than Co-Array Fortran, then scaling to large number of nodes will tend to lower the performance of MPI. This will become key in the future since

46
nodes will be able to hold more cores. As such, larger simulations will be able to run if Co-Array Fortran is employed within nodes rather than MPI.

Figure 38 - The value of the message length where both the MPI/OpenMP and Coarray-v2 codes take the same execution time. (Graph is \(\log_{10}\log_{10}\))

Figure 38 contains the size of the message where the time for the MPI/OpenMP code and the time for the Coarray-v2 code is the same, for a given number of cores. Thus, it can be concluded that if a simulation requires a specific number of cores and employs a specific message size, and these correspond to a point below the curve in Figure 38, then the programmer should employ MPI or OpenMP depending on the plot, otherwise the programmer should employ Coarray-v2.

Moreover, it was found that, provided the message length is large enough, the Coarray-v2 version of MPI_Allreduce was faster than both the OpenMP and MPI versions within a node and, for even larger message lengths, can be faster than MPI between nodes. However, in general, for a fixed message length, it is likely that Co-Array Fortran should be used inside a node, and MPI between nodes.

Therefore, in conclusion, users should employ Co-Array Fortran within a node and MPI between nodes so that they can run larger simulations and be compatible with future machine architectures that will handle more cores per node. Furthermore, users should employ OpenMP within a node for small message sizes, while employ Co-Array Fortran within a node for large message sizes.

We have found that MPI_Allreduce with an equivalent Co-Array Fortran routine is beneficial when running within nodes. It should be noted that MPI_Allreduce is a commonly used routine. As such, this result will not only be of interest to the FLASH Center, but will be of significant interest to the HPC community in general.
7.2 Summary of Achievements

Initial investigations for the introduction of Co-Array Fortran into FLASH were presented to the FLASH center. Using the CrayPAT profiler on HECToR, the Maclaurin Spheroid test is analysed. The analysis reveals the MPI communication bottlenecks. In addition, a tested Co-Array implementation of the AllReduce reduction operation was delivered for FLASH4-Alpha. The delivered method called Co-Array-v2 brings FLASH4-Alpha closer towards code parallelism for performing faster cosmological simulations. Prior to this project, no knowledge of Co-Array Fortran and its introduction into FLASH existed.

The several versions of the Co-Array implementations are compared against each other and the optimal one is compared against the MPI_Allreduce and OpenMP methods. Moreover, their performances are analysed and evaluated. Several potential optimisations are proposed that could help introduce Co-Array Fortran into the FLASH code. For instance, the communication optimisations can be useful for other collective operations and the replacement of MPI operations for simpler expressions.

This project gives constructive feedback to the FLASH team about the introduction of Co-Array Fortran into the FLASH code for parallelism. The Co-Array collective operations presented in this report can contribute to the parallelism of FLASH.

The Co-Array Fortran implementations were tested on both the HECToR and IBM BG/P supercomputers. Many problems with the Cray and IBM compilers were discovered and repaired.

7.3 Future Work

Further work could be done in incorporating Co-Array Fortran wrappers to replace MPI wrappers for some collective communication routines within nodes. Their runtime performance should be investigated depending on the type of problem and resources needed to simulate in FLASH. This is because although some Co-Array collective communication could improve for inter-node communication and can handle larger simulations, MPI collective communication performs better for intra-node communication and therefore handle much larger scaling. It could be beneficial to analyse and interpret the effects of the proposed optimisations found in the previous sections of this report.

The introduction of Co-Array Fortran to replace point-to-point MPI communication can not only create simpler code, but also reduce synchronisation since Remote Memory Access operations are used where there is participation of only one processor in the communication. The reduction of synchronisation could highly optimise the FLASH code since a lot of the communication bottlenecks in the Maclaurin Unit Test were due do MPI_BARRIER and MPI_WAITALL.
Since there is no touchstone Co-Array Fortran compiler, more investigation should be undergone in code compilation and optimisation. Since Co-Array operations and wrapper scripts had to be adjusted and created to be run between HECToR and the IBM BG/P, the codes should be incorporated inside a portable benchmarking suite for future testing to be able to run on several machines and achieve good performance without the need to alter the essential code.
Appendix A

A.1 MPI

program mpiallreduce
 implicit none
 include 'mpif.h'
 integer :: rank, size, ierror, tag, status(MPI_STATUS_SIZE)
 integer :: i, j
 double precision :: start, end, time
 integer, parameter :: max_iter = 100000
 integer :: array_size
 integer, allocatable, dimension(:) :: max_local
 integer, allocatable, dimension(:) :: max_global
 real, allocatable, dimension(:) :: x
 integer :: i_extent
 integer :: N
 CHARACTER(LEN=100) :: arg
 N = COMMAND_ARGUMENT_COUNT()
 IF (N < 1) THEN
  write(*,'(a)') "usage: ./mpiallreduce option1"
  write(*,'(a)') "option1=integer"
 ELSE
  CALL GETARG(1,arg)
  read(arg,*) array_size !Now convert string to integer
 ENDIF

 ALLOCATE(max_local(array_size),max_global(array_size),x(array_size))
 call MPI_INIT(ierr)
 call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
 call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
 call random_seed
 call random_number(x)
 max_local = INT(100*x)
 start = MPI_WTIME()
 do i =1, max_iter
  call MPI_Allreduce(max_local, max_global, array_size,
   MPI_INT, MPI_MAX, MPI_COMM_WORLD, ierror)
 end do
 end program mpiallreduce= MPI_WTIME()
 time = end - start
 write(*,*) 'Time = ', time, 'for array size = ', array_size,
 'from MPI task ', rank
 do i =1,3
  i_extent = 10**i
"
start = MPI_WTIME()
do j = 1, max_iter
    call MPI_Allreduce(max_local, max_global, i_extent, 
                      MPI_INT, MPI_MAX, MPI_COMM_WORLD, ierror)
end do
end do= MPI_WTIME()
time = end - start
write(*,*), 'Time = ', time, 'for extent', i_extent, 'from MPI task', rank
end do
call MPI_FINALIZE(ierr)
end program
A.2 Main Coarray file

program main
  implicit none
  include 'mpif.h'
  integer :: myimage, numimage, i
  integer :: array_size
  integer, allocatable, dimension(:) :: v
  integer :: vmax[*]
  double precision :: start, end, time
  real, allocatable, dimension(:) :: x
  integer, parameter :: max_iterations = 100000
  integer :: temp
  integer :: N
  CHARACTER(LEN=100) :: arg
  N = COMMAND_ARGUMENT_COUNT()
  IF (N < 1) THEN
    write(*,'(a)') "usage: ./main option1"
    write(*,'(a)') "option1=integer"
  ELSE
    CALL GETARG(1,arg)
    read(arg,*) array_size
  ENDIF
  ALLOCATE(v(array_size),x(array_size))
  numimage = num_images()
  myimage = this_image()
  call random_seed
  call random_number(x)
  v = INT(100*x)
  start = MPI_WTIME()
  do i = 1, max_iterations
    call allreduce_max_allget(v,vmax,array_size)
  end do
  end = MPI_WTIME()
  time = end - start
  write(*,*) 'vmax = ', vmax, 'by image = ', this_image(), 'during time = ', time
end program main
A.3 Coarray-v1

SUBROUTINE allreduce_max_allget(v,vmax,array_size)
  integer, intent(in) :: array_size
  integer, intent(in) :: v(array_size)
  integer, intent(out) :: vmax[*]
  integer i
  vmax = maxval(v)
  sync all
  if (this_image()==1) then
    do i = 2,num_images()
      vmax=max(vmax,vmax[i])
    end do
    do i = 2, num_images()
      vmax[i]=vmax
    end do
  end if
  sync all
END SUBROUTINE allreduce_max_allget

A.4 Coarray-v2

SUBROUTINE allreduce_max_allget(v,vmax,array_size)
  integer, intent(in) :: array_size
  integer, intent(in) :: v(array_size)
  integer, intent(out) :: vmax[*]
  integer i
  vmax = maxval(v)
  sync all
  do i = this_image()+1,num_images()
    vmax=max(vmax,vmax[i])
  end do
  do i = 1,this_image()-1
    vmax=max(vmax,vmax[i])
  end do
  sync all
END SUBROUTINE allreduce_max_allget

A.5 Coarray-v3

SUBROUTINE allreduce_max_allget(v,vmax,array_size)
  integer, intent(in) :: array_size
  integer, intent(in) :: v(array_size)
  integer, intent(out) :: vmax[*]
  integer i
  vmax = maxval(v)
  sync all
  do i = 1,num_images()
    vmax=max(vmax,vmax[i])
  end do
  sync all
END SUBROUTINE allreduce_max_allget
A.6 OpenMP

program ompallreduce
use omp_lib
  implicit none
  integer :: max_global
  integer :: n  ! Size of integer array
  integer, parameter :: max_iterations = 100000
  integer, allocatable, dimension(:) :: max_local
  integer :: i, id, nthreads, j
  double precision :: start, end, time
  real, allocatable, dimension(:) :: x
  integer :: Z
  CHARACTER(LEN=100) :: arg

  Z = COMMAND_ARGUMENT_COUNT()
  IF (Z < 1) THEN
    write(*,'(a)') 'usage: ./ompallreduce option1'
    write(*,'(a)') 'option1=integer'
  ELSE
    CALL GETARG(1, arg)
    read(arg,*) n
  ENDIF
  ALLOCATE(max_local(n), x(n))
  id = OMP_GET_THREAD_NUM()
  nthreads = OMP_GET_NUM_THREADS()
  call random_seed
  call random_number(x)
  max_local = INT(100*x)
  max_global = max_local(1)
  start = OMP_GET_WTIME()
  do j = 1, max_iterations
    !$omp parallel do default(none) shared(max_local, max_global,n) private(i)
    do i = 2, n
      max_global = max(max_global, max_local(i))
    end do
    !$omp end parallel do
  end do
  end = OMP_GET_WTIME()
  time = end - start
  write(*,*) 'Global sum = ', max_global, 'by processor', id, 'in time = ', time, 'for array size = ', n
end program

A.7 Maclaurin Spheroid Call Graph tree
References


[22] Coarray Fortran 2.0 Documentation [Online], http://caf.rice.edu/documentation/index.html