Optimising GS2 through Hybrid Parallelisation

David J. Hardman

August 23, 2013

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2013

Abstract

Clustered architectures are now ubiquitous in high performance computing; nearly all supercomputers today are built using a collection of nodes, each having multiple cores with shared memory. The hybrid programming model has been born out of the need for a programming model that fits this hybrid architecture; suited to both the shared and distributed memory aspects of such a system. It is debated, however, whether or not hybrid implementations lead to better performance than simply having a message-passing implementation.

GS2 is a gyrokinetic simulation code parallelised using the MPI message-passing library. In this project we investigated the addition of OpenMP directives to GS2, in order to create a hybrid version of the code. We then set about testing the performance of our hybrid version compared to that of the original MPI-only version.

It was found that with the correct ratio of MPI processes to OpenMP threads, a performance increase is possible. We found this performance increase to be cumulative and proportional to the number of time steps simulated, i.e. the more time steps simulated the larger the increase in performance when compared to the MPI-Only version of GS2. Furthermore, the addition of OpenMP threads allowed GS2 to scale to much larger core counts, as well as outperforming the like for like MPI process counts.

Contents

Chapter 1 Introduction 1

Chapter 2 Architectures 2

2.1 Shared Memory 2

2.2 Distributed Memory 4

2.3 Shared Memory Cluster 5

Chapter 3 Programming Models 6

3.1 Shared Memory 6

3.1.1 Communication 8

3.1.2 Synchronisation 8

3.1.3 Work distribution 10

3.2 Message Passing 11

3.2.1 Communication 12

3.2.2 Synchronisation 15

3.3 Hybrid 15

3.3.1 Hybrid Programming Model 16

3.3.2 Performance Considerations 17

Chapter 4 GS2: The Gyrokinetic code investigated 19

4.1 GS2 19

4.2 The ‘timeadv’ routine 20

4.3 Initial Benchmarking and Profiling 21

4.3.1 Benchmarking 22

4.3.2 Scalasca Profiling 24

4.3.3 Communications and Data Distribution 26
Chapter 5 Implementation 28
5.1 System Used 28
5.2 Introducing OpenMP 29
5.2.1 Thread Ratio 31
Chapter 6 Results and Analysis 33
6.1 Scaling Performance 33
6.2 Workload Performance 38
Chapter 7 Conclusions 41
7.1 Future Work 42
Appendix A Input Datafile 43
Appendix B Example Submission Scripts 48
References 50

List of Figures

Figure 2.1: Symmetric Multiprocessor Shared Memory Architecture 3

Figure 2.2: PE Focus in Shared Memory Architecture 3

Figure 2.3: NUMA Shared Memory Architecture.....4

Figure 2.4: Distributed Memory Architecture 4

Figure 2.5: Shared Memory Cluster Architecture 5

Figure 3.1: OpenMP executing code with both Serial and Parallel Regions.....7

Figure 3.2: Graphical representation of the Broadcast routine.....13

Figure 3.3: Graphical representations of the Scatter and Gather routines 14

Figure 3.4: Graphical representation of the all reduce function.....14

Figure 3.5: Threads running within a NUMA region, the master thread (orange PE, labelled 0) acting as the MPI process for the NUMA region.....16

Figure 4.1: Graph comparing the achieved parallel performance of GS2 with the ideal parallel performance. Source for graph [9].....20

Figure 4.2: Example of a call tree in the timeadv routine 21

Figure 4.3: Graph showing the execution time of GS2 for 1000 time steps using varying numbers of PEs, initial data point is 64 PEs 22

Figure 4.4: Graph showing the speed up of GS2 for 1000 time steps using varying numbers of PEs, initial data point is 64 PEs 23

Figure 4.5: Graph showing the parallel efficiency of GS2 for 1000 time steps using varying numbers of PEs, data has been re-biased to begin at 1 for 64 PEs 23

Figure 4.6: Top level Scalasca report, showing hotspots for execution time. 24

Figure 4.7: Extended call trees in Scalasca output, showing the ‘timeadv’ function responsible in both cases for the large execution time. 25

Figure 4.8: Scalasca screen showing the proportional time consumed by MPI communications, and routines in which communications are most commonly called. .26

Figure 4.9: Scalasca screen showing the amount of data transferred throughout the run time of GS2, along with the routines involved in transferring the higher proportions of this data. 27

Figure 5.1: Chart showing time spent in advance steps, initialisation and total execution time against the number of threads per MPI process for the hybrid implementation of GS2 running 1000 time steps 32

Figure 6.1: Graph showing the Total Execution time for Hybrid GS2 across varying numbers of PEs for 1000 steps 34

Figure 6.2: Graph showing the Total Execution time for both Hybrid and MPI-Only GS2 against the number of PEs used. Initial data point is at 128 PEs. 34

Figure 6.3: Graph showing the speed up against number of PEs used for both Hybrid and MPI-Only GS2. Initial data point is at 64 PEs. 35

Figure 6.4: Graph showing Parallel Efficient against number of PEs used for both Hybrid and MPI-Only GS2. Initial data point is at 64 PEs, data has been re-biased to begin at 1. 35

Figure 6.5 : Graph showing the Total execution time against the number of MPI processes used for both Hybrid and MPI-Only GS2. Initial data point is at 64 MPI processes. 36

Figure 6.6: Graph Showing the Initialisation time against Number of MPI processes for both MPI-Only and Hybrid GS2. 37

Figure 6.7: Graph Showing the Advance Step time against Number of MPI processes for both MPI-Only and Hybrid GS2. 37

Figure 6.8: Graph showing Total Execution time against Number of time steps for both Hybrid and MPI-Only GS2. 38

Figure 6.9: Graph showing the Initialisation time against the number of time steps for both Hybrid and MPI-Only GS2. 39

Figure 6.10: Graph showing the Advance Step time against the number of time steps for both Hybrid and MPI-Only GS2. 40

List of Code Supplements

Code Supplement 3.1: Threads setting an individual element in a shared array 8

Code Supplement 3.2: Effective use of the ‘NOWAIT’ clause for consecutive parallelised do loops 9

Code Supplement 3.3: Use of the ‘barrier’ directive following a ‘master’ region 9

Code Supplement 3.4: Example of named critical sections in synchronisation 10

Code Supplement 3.5: Master Only and Funnelled hybrid schemes implemented in code. 17

Code Supplement 5.1: The main computational loop within the transfrom2_5d_accel routine (not including comments) 30

Acknowledgements

I would like to thank my supervisor, Mr Adrian Jackson, for his guidance and support throughout this project.

I would also like to thank EPCC, specifically the teaching staff for the MSc in High Performance Computing, for providing me with the knowledge and opportunity to undertake this project.

Finally, I would like to thank my friends and family, for their support and encouragement throughout the year.

Chapter 1

Introduction

In the past architectures used in high performance computing generally fell into one of two categories, shared memory or distributed memory. Shared memory architectures consist of two or more processors connected to a single address space, via an interconnect. Whereas distributed memory is multiple nodes, each having a processor with its own memory, connected via an interconnect. There is also the third option of creating a clustered architecture were each node in a distributed memory machine is a shared memory multiprocessor (SMP). The advent of multi-core processors has made this hybrid architecture ubiquitous throughout modern high performance computing.

One such system is HECToR, the system used whilst undertaking this project and the UK national supercomputing service at the time of writing. With the rise of this type of architecture opportunities for improved programming models becomes available for suitable programs. Currently the standard message passing approach is commonly used; however this does not take advantage of the shared memory capabilities within a node. It stands to reason that a shared memory programming model could be used for intra-node computation, whilst maintaining message passing for the inter-node communication. Taking this approach also allows us to take an existing code parallelised using the message passing approach, and add shared memory threading to computationally intensive sections of code.

For the purposes of this study we will be using the ‘GS2’ code, which is used to model particles in nuclear fusion plasma. Currently GS2 is parallelised using the MPI message passing library, and we will be adding OpenMP directives to provide the thread control. In doing this we hope to see a decrease in execution time, when using similar computational resources. Chapters two and three will provide background to system architectures and associated programming models respectively, while chapter four looks more closely at GS2 and the specific performance hotspots associated with it. Moving on to chapters five and six we outline the implementation of OpenMP directives and discuss the performance results.

Chapter 2

Architectures

This chapter will cover the architectures used throughout high performance computing. For the purposes of this project we will classify these parallel architectures in terms of their memory, specifically: shared memory, distributed memory and clusters. The figures given to exemplify these architectures throughout this chapter where created by the author, based on the information presented and referenced.

For continuity throughout report we will refer to an independent computational unit as a processing element (PE), i.e. a core on a multi-core chip would be a single processing element, or a single core CPU would be considered a PE.

The information presented in this chapter is based on that found in [1].

2.1 Shared Memory

A shared memory machine involves several PEs connected via a bus/interconnect to a single global space memory. Shared memory systems are run using a single operating system (OS) across the entire system; this creates the appearance of a single machine from the users' perspective. This OS is also responsible for automatically moving jobs around the PEs.

One form of shared memory machine is the symmetric multiprocessor (SMP) shown in figure 2.1 below. In an SMP system each PE has equal access to all parts of memory. For shared memory architectures communication is entirely implicit. Any change in shared memory is accessible to other PEs, using these changes in memory, not explicit messages; data is 'exchanged' between PEs.

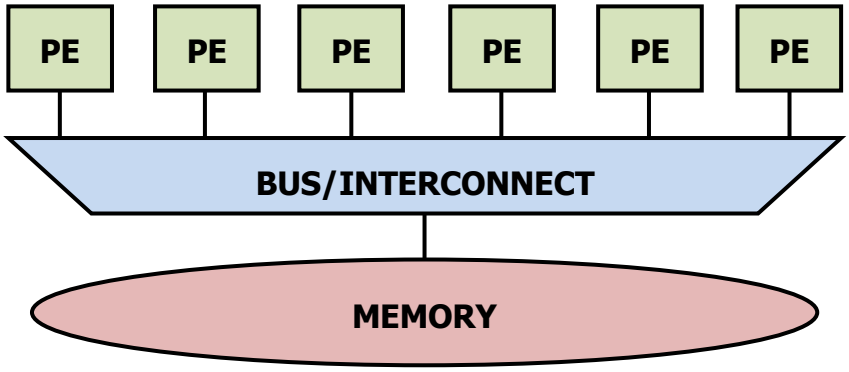

Figure 2.1: Symmetric Multiprocessor Shared Memory Architecture

Having multiple independent PEs using the same global address space also complicates the use of caching, specifically maintaining cache coherence between each PE. A more detailed look at the architecture outlined in figure 2.1, shows that each PE would actually have a cache structure as represented in figure 2.2. Cache coherency can be implemented using various mechanisms, such as 'directory-based coherence' or 'snooping', and is not usually something a programmer is concerned with.

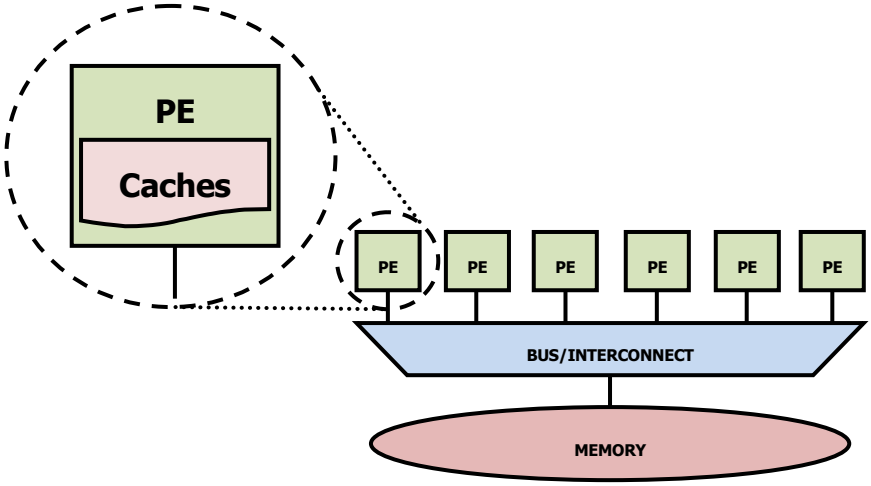

Figure 2.2: PE Focus in Shared Memory Architecture

Another implementation of the shared memory architecture is 'Non-Uniform Memory Access' (NUMA), as shown in figure 2.3. In this case although physically separate, memory is still addressed as a single global space. However, PEs take less time to access data on physically close memory, specifically on what is referred to as "local" memory. In figure 2.3, the memory labelled X would be considered 'local' to PEs 1, 2, 3. A cache coherent NUMA architecture is referred to simply as CC-NUMA.

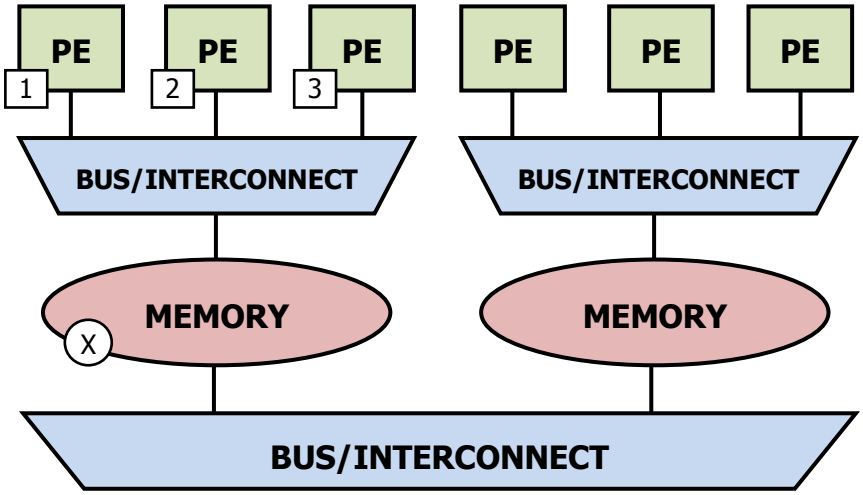

Figure 2.3: NUMA Shared Memory Architecture

Using a NUMA shared memory architecture can also help to abate the occurrence of a bottleneck on the bus/interconnect. In SMP systems this can cause problems in scaling, as adding more PEs may overwhelm the bus/interconnect. Through the use of CC-NUMA architecture the scaling of a shared memory system can be vastly improved, however maintaining a coherent cache between all of the PEs remains a limiting factor.

2.2 Distributed Memory

Distributed memory systems consist of numerous independent PEs connected via an interconnect mechanism, with each PE having a local private memory space; as shown in figure 2.4. For distributed memory architectures each PE runs its own OS, this allows each PE to operate completely independently. It also means that each process run for a particular program is fixed on a set PE. Data is passed between PEs via explicit messages, these messages are direct in that they are sent specifically between PEs; rather than altering a shared memory space.

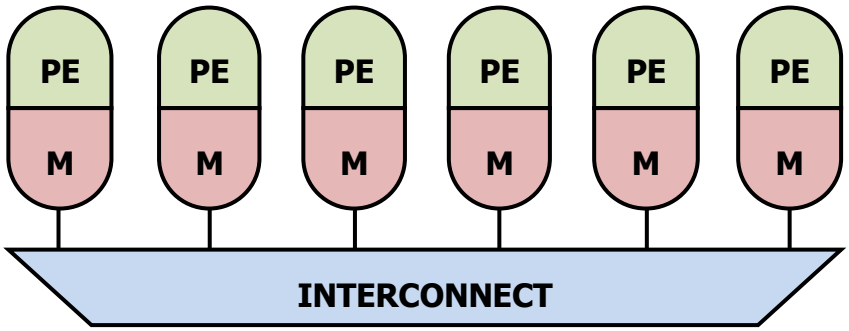

Figure 2.4: Distributed Memory Architecture

In a distributed system adding PEs increases the memory bandwidth as such a distributed system could grow to almost any size, assuming power consumption and other physical factors such as cooling are not limiting. However, for a distributed system to scale well the interconnect used becomes a key factor. The interconnect 'shape' is often an important feature of a distributed memory machine, as an effort is made to reduce aspects of the network such as the bisection width or diameter.

Unlike shared memory architectures a distributed system can continue to utilise cache memory effectively as each PE operates solely on a local private memory space. However, to fully exploit the benefits of a distributed memory system there is often a larger overhead for the programmer. For example the explicit nature of communications means careful attention must be paid to ensure messages are sent and received by the correct PE, at the correct time.

2.3 Shared Memory Cluster

A shared memory cluster brings together the features of shared and distributed memory architectures, as a collection of shared memory nodes connected via an interconnect, as shown in figure 2.5. This should not be confused with distributed shared memory architectures, which are physically similar to distributed memory systems, but each PEs local memory is utilised in a single shared global address space.

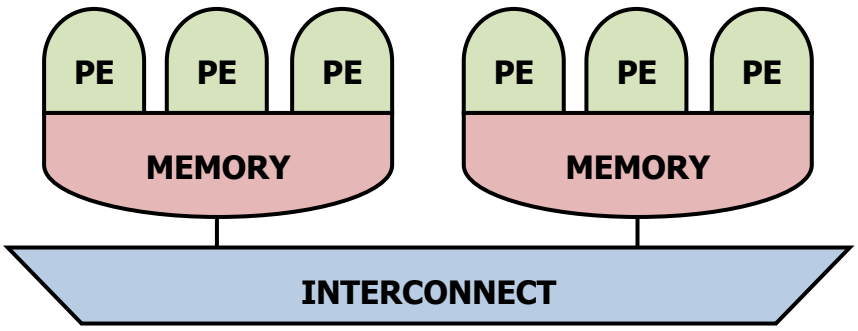

Figure 2.5: Shared Memory Cluster Architecture

Each node of a cluster has a local memory space shared between PEs within the node, however it is private from other nodes in the system. This allows intra-node communication to be implicit via shared memory, whereas inter-node communication is explicit via message passing. It is common to find a multi-core processor used as a node in a shared memory cluster, due to their cost effective nature. However, it can be difficult to take advantage of the heterogynous nature of the architecture and it is often more difficult to decipher its performance.

Chapter 3

Programming Models

In this chapter we will discuss the programming models applied throughout this project. We will be using the 'process interaction' classification, i.e. the classification is based on the mechanisms used for process communication in a parallel system. We begin by covering the 2 programming models that are brought together for hybrid computing; the shared memory and distributed memory models. It should be noted that throughout this chapter details of programming APIs will be used that may be available across various languages, as this project was carried out using Fortran, this is the language we will be using as context for discussing any APIs or code examples.

3.1 Shared Memory

If following section is based on information found in [2] and [3]. Although some of the concepts and principles covered in this section will be universally true for all shared memory Application Program Interfaces (APIs), will be focusing on the OpenMP API as it was used during the implementation of the project. The OpenMP API is a joint project owned by a set of hardware and software vendors, this group is called the OpenMP Architecture Review Board (ARB). The ARB is responsible for overseeing and approving new versions of the OpenMP specification.

As with all shared memory APIs, OpenMP is based on the concept of threads. Generally speaking in high performance computing a single thread will execute on a single PE, however if a chip has simultaneous multi-threading, or similar technology, multiple threads could execute on the same PE. A program parallelised using OpenMP will begin in a serial manner by running a single thread. This initial thread is known as the master thread and will continue to execute for the entire program run. When the master thread reaches a section of the code to be run in parallel (or parallel region), the predefined number of threads is spawned using the fork/join model. As you can see from figure 3.1 once in a parallel region all threads (including the master thread) execute the same code, and at the conclusion of the parallel region the all the child threads terminate and the master thread continues to execute through the code.

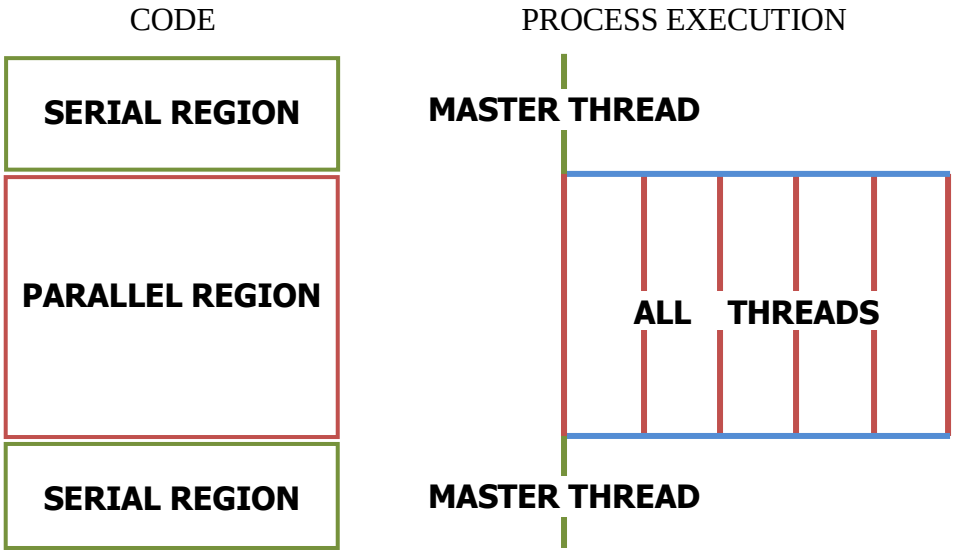

Figure 3.1: OpenMP executing code with both Serial and Parallel Regions

OpenMP uses a combination of runtime library routines, compiler directives and environment variables to implement parallelism in a serial code. As stated above, all threads will execute the same code within a parallel region, however, through the use of conditional statements using the threads ID, different threads to execute different paths within the region. When entering a parallel region with a compiler directive often a number of clauses can be employed allowing the programmer to attach additional attributes to a parallel region. These attributes include the scope of variables used within a region, work distribution and collective functions.

The two main categories for variables within a parallel region are ‘private’ and ‘shared’. If a variable is set to ‘shared’ for the parallel region, a single copy of the variable will be held and accessible to all threads executing that region. If a shared variable has been instantiated and set before the parallel region, it will retain its value when entering the region. When a variable is flagged as private for a parallel region each thread will instantiate its own copy, which can be only be accessed by that thread. A private variable will always be empty upon entering the parallel region, unless the ‘firstprivate’ clause is used, in which case each threads copy of the variable will be instantiated with the value of the variable prior to the parallel region.

3.1.1 Communication

Communication between threads in OpenMP is implicit, using shared variables to make data publicly available to all threads. Using shared variables in this way places the onus on the programmer to ensure that individual threads do not overlap in their access to a shared variable. This form of communication is particularly useful when implementing it with a vector variable such as a shared array. A simple example can be seen in code supplement 3.1 below.

```
1   ! OpenMP directive
2   !$OMP PARALLEL SHARED(id_array), PRIVATE(id)
3     ! Retrieve the threads id
4     id = OMP_GET_THREAD_NUM()
5     ! As Fortran array bounds begin at 1 by default and
6     ! OpenMP thread IDs begin at 0 we add 1 to the ID so
7     ! it may be used as a valid array index, each thread then
8     ! assigns its ID to a different element in id_array
9     id_array(id+1) = id
10  !$OMP END PARALLEL
```

Code Supplement 3.1: Threads setting an individual element in a shared array

This code allows all threads to access the data in the entire array, including the master thread even after the parallel region has finished as the array is declared shared. It is often good practice to add the 'default' clause, set to 'none', to parallel regions. Adding this clause ensures that each variable must be specifically set to either private or shared. In this case, if a variable is used within a parallel region and has not been explicitly declared shared or private, an error will be thrown at compile time. This can be a very useful tool for programmers, ensuring every variable in a parallel region has been accounted for.

3.1.2 Synchronisation

Many OpenMP directives employ implicit synchronisation before exiting the parallel region they denote; the standard 'parallel' directive is an example of this. In these cases each thread must reach the end of the parallel region before the master thread can continue executing the serial code. For some directives, this implicit barrier can be removed using the 'no wait' clause. This can be of particular use in cases where 2 consecutive, independent do loops are parallelised, as the 'do' directive has the aforementioned implicit barrier. See code supplement 3.2 for an example were a 'no wait' clause may be implemented effectively.

```
1   !$OMP PARALLEL SHARED(a, b), PRIVATE(i, j)
2   ! First do loop does not need an implicit barrier
3   !$OMP DO NOWAIT
4     DO i = 1, 20
5       a(i) = i
6     END DO
7   !$OMP END DO
8   ! Second do loop is not influenced by the first
9   !$OMP DO
10    DO j = 1, 10
11      b(j) = j
12    END DO
13  !$OMP END DO
14  !$OMP END PARALLEL
```

Code Supplement 3.2: Effective use of the 'NOWAIT' clause for consecutive parallelised do loops

OpenMP also has a mechanism for explicit synchronisation, using the 'barrier' directive. This directive should be employed carefully by the programmer as it can add considerable overheads. It may also lead to a program to deadlock if used in regions of code only executed by a subset of the total threads, as it requires all threads to enter the barrier before they may continue executing. See code supplement 3.3 for an example of the barrier directive being used alongside a directive without an implicit barrier.

```
1   !$OMP PARALLEL SHARED(a), PRIVATE(i)
2   ! The master directive does not have an implicit barrier
3   !$OMP MASTER
4     a = 0
5   !$OMP END MASTER
6   ! 'a' must be set before all threads execute the loop
7   !$OMP BARRIER
8   !$OMP DO
9     DO i = 1, 10
10      a(i) = a(i) + 2
11    END DO
12  !$OMP END DO
13  !$OMP END PARALLEL
```

Code Supplement 3.3: Use of the 'barrier' directive following a 'master' region

Another form of explicit synchronisation in OpenMP is a critical section. Critical code, indicated with the use of the 'critical' directive, can only be executed by a single thread

at any given time. Critical section can also be ‘named’, this allows critical sections to be grouped, no thread can enter a critical section if a thread is currently executing code in a critical section with the same name, see code supplement 3.4.

```
1   !$OMP PARALLEL SHARED(a), PRIVATE(i)
2   ! The critical sections are named 'ALPHA'
3   !$OMP CRITICAL (ALPHA)
4     stack = getNext(list)
5   !$OMP END CRITICAL
6     call orderList(stack)
7   ! Once all the stacks have been retrieved then return them
8   !$OMP CRITICAL (ALPHA)
9     call addItem(list, stack)
10  !$OMP END CRITICAL
11  !$OMP END PARALLEL
```

Code Supplement 3.4: Example of named critical sections in synchronisation

The critical directive is blocking, in this context a blocking directive is a directive at which all threads must wait before entering, while a non-blocking routine allows threads to continue executing other code until the directive is available for execution. For a non-blocking synchronisation routine, a lock is ideal as it can be either blocking or non-blocking. In this case a lock must be held before executing a section of code, once completed the lock is then released, allowing another thread to enter the lock routine. For single statements, rather than employing a critical region, an ‘atomic’ directive may be used. An atomic directive may incur less overhead than a critical region but it may only be used for statements of the follow forms: *x = x op expr*, *x = expr op x*, *x = intr* **(***x, expr***)** or *x = intr***(***expr, x***)**; where *op* is one of +, *, -, /, .and., .or., .eqv., or .neqv; and *intr* is one of MAX, MIN, IAND, IOR or IEOR.

3.1.3 Work distribution

The OpenMP directives used to indicate work distribution amongst threads are the ‘do’, ‘single’ and ‘master’ directives. The single and master directives have similar functions but with some key differences. They both indicate that a section of code should be executed once, by a single thread only. However, the master directive has the additional caveat that the thread to execute this code must be the master thread, whereas the code in a single directive is executed by the first thread to reach the block. The single directive is one of the directives which have an implicit barrier at the end of the block, which is not the case for the master directive. Therefore during the execution of a single region all other threads are idle until the thread has completed the block, while during the execution of a master region, all other threads continue on through the code after the master region.

The do directive is used to divide the iterations of a loop between threads. This is another example of a directive having an implicit barrier in OpenMP, upon completion

of a threads allotted loop iterations it then waits until all threads have completed before moving on. The use of loops to exploit parallelism in a code is very common, as such there is a shorthand directive combining the 'parallel' and 'do' directives, the 'parallel do' directive. If a do directive has no additional clauses the iterations of the loop will be divided amongst the threads as equally as possible, however this can be an ambiguous process. For example 7 iterations may be divided amongst 3 threads as 3, 3, 1 iterations or 3, 2, 2 iterations. There is however the 'schedule' clause which provides some control to the programmer as to how the iterations of a loop are distributed across the threads. The schedule directive takes the form SCHEDULE (*kind[, chunksize]*), where *kind* takes one of the following values:

- ***STATIC:*** the 'chunks' of iterations are assigned in a block cyclic schedule.
- ***DYNAMIC:*** the chunks are executed in a first come first serve basis by the next available thread until all the chunks have been executed.
- ***GUIDED:*** chunks are assigned to threads in a similar way to dynamic, however size of the chunks varies. Chunks begin large and reduce in size exponentially, each chunk being the size of the remaining iterations divided by the number of threads.
- ***AUTO:*** allows the runtime to decide on the assignment of iterations to threads.
- ***RUNTIME:*** the schedule is defined at runtime by the user with the environment variable OMP_SCHEDULE.

chunksize is a positive integer indicating the number of iterations in a chunk to be executed contiguously by a thread. It is optional to add *chunksize* to the clause in which case static scheduling behaves the same as not stating any type of schedule, while the dynamic and guided schedules default to a *chunksize* of 1 if not explicitly stated. A *chunksize* cannot be given for the auto or runtime schedule types.

Each schedule has characteristics that may be useful depending on the situation. For well-balanced loops, i.e. those where iterations all require similar time to execute a static schedule works well and has the least overhead associated with it. Dynamic scheduling is good for loop in which iterations have a large variance in execution time, but does not take advantage of data locality. Guided usually incurs less overheads than dynamic, but it can produce disastrous performance if the first set of iterations of a loop are the most computationally expensive. An automated schedule choice can be beneficial in situations where a loop is executed repeatedly.

3.2 Message Passing

The information presented in this section is based on that found in [4] and [5]. The code used for this project was previously parallelised with the Message Passing Interface (MPI) standard, as such the following chapter will focus on the aspects of MPI and not any other message passing libraries. Message passing programming is used almost ubiquitously in high performance computing. This is due to its ability to run on both shared and distributed memory architectures. Although it is more suited to distributed memory architectures as each PE has a private memory space and therefore must communicate with other PEs via explicit messages.

In message passing programming each PE runs a copy of the same code, each referred to in MPI as a process. Each process will execute the same code, but as with OpenMP the path of execution can be altered using the process ID, in MPI this ID is referred to as the process rank. This rank is unique within a set collection of processes, called a communicator. The initial global communicator containing all processes is referred to as MPI_COMM_WORLD.

3.2.1 Communication

Communications in MPI take place within a communicator, identifying a receiving process by its rank within this communicator. The sender and receiver may be referred to as the source and destination respectively. There are 2 types of communication in MPI, point-to-point and collective.

In point-to-point communication there is a single source and destination. The source calls a 'send' routine while the destination must call a corresponding 'receive' routine. The send routine states the data that is to be sent in the message, the send buffer. While the receive routine specifies the receive buffer, where the data should be stored. A message will also contain metadata describing the message, this is called the status and is stored separately from the receive buffer. Along with the rank of the sender/receiver and a communicator, a point-to-point message may also be 'tagged'. A tag must be a non-negative integer value and in many cases programs simply set all tags to 0. However, tags can be useful if specific message want to be chosen from all of the received messages.

A receiver may also employ 'wildcarding'. Using the MPI_ANY_SOURCE and/or MPI_ANY_TAG environment variables, a message can be received from any sender and/or with any tag respectively. The programmer can retrieve the actual source and value of a tag by looking at the 'status' of a message. If a receive matches multiple messages from the same sender in the "inbox", MPI guarantees message order preservation, i.e. messages will be processed in the order they were sent.

There are several communication 'modes' that can be used when sending a message. These modes determine when a call to the send routine completes for the process calling it. These modes are as follows:

- ***Synchronous:*** Only completes when the receive has completed.
- ***Buffered:*** Always completes (unless an error occurs), irrespective of receiver.
- ***Standard:*** May be ether synchronous or buffered, depending on availability of buffer space.
- ***Ready:*** Completes when a message has arrived at a process (that process may not have posted the necessary receive).

Unlike the send routine, the receive routine will simply complete when the message has arrived.

For a point-to-point message to succeed the following requirements must be met:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.

- The call to the send and receive routines must specify the same communicator.
- The call to the send and receive routines must specify the same tag.
- The call to the send and receive routines must have matching message types.
- Receiver's buffer must be large enough.

Collective communications are called by all processes in a given communicator, for example a collective communication called using the MPI_COMM_WORLD communicator must be called by all processes. A collective communication may have multiple sources and multiple destinations and unlike point-to-point messages a tag is not included. Collective communications can be used to distribute or collect data across many processes, globally reduce a value after a distributed work effort or synchronise the processes within a communicator. To 'copy' data from a process to all processes within a communicator the broadcast routine is used, see figure 3.2.

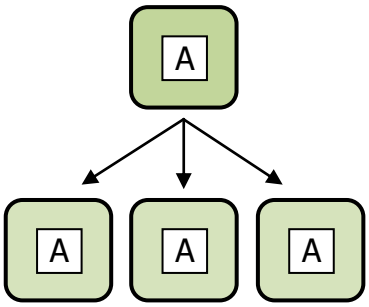

Figure 3.2: Graphical representation of the Broadcast routine

For data distribution and collection 2 important routines are the scatter and gather routines respectively. These routines have a root process, this is the process from which the data is being distributed and to which the data is collected for the scatter and gather routines respectively, see figure 3.3 below.

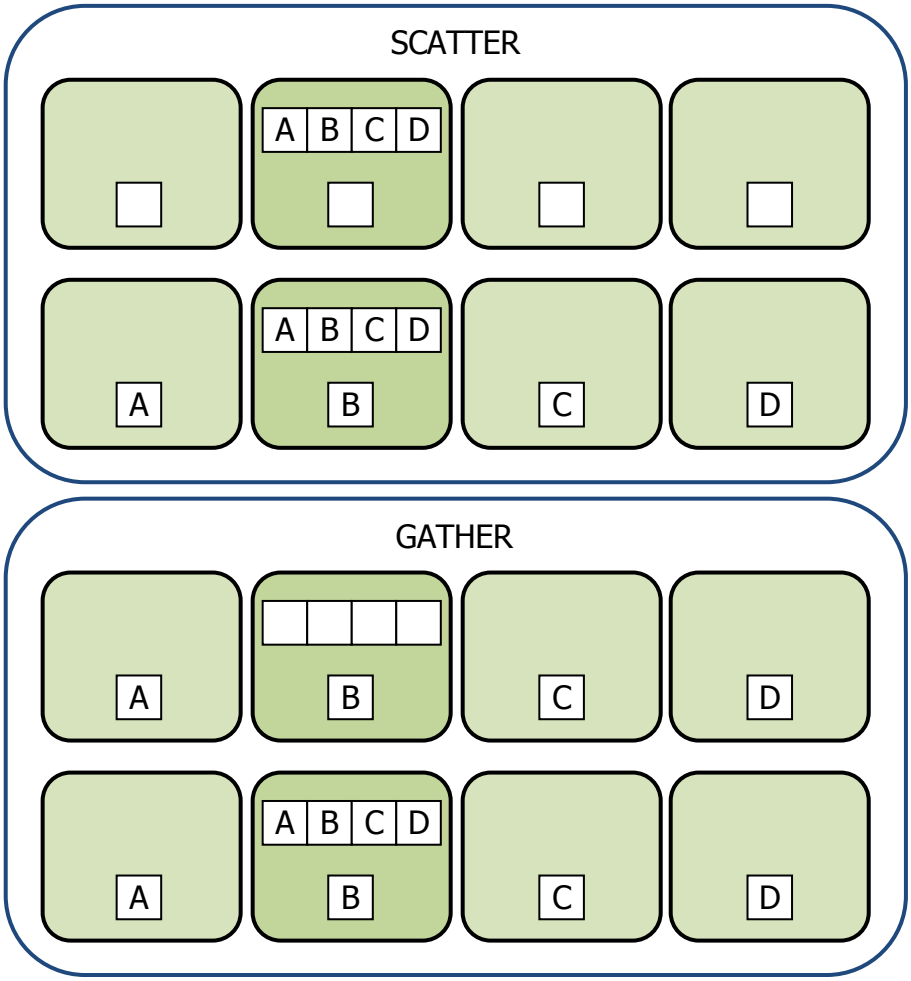

Figure 3.3: Graphical representations of the Scatter and Gather routines

For a collective reduce there are 2 routines with slightly different functions, the MPI_REDUCE and MPI_ALLREDUCE. The reduce function can be used to perform many types of reductions, such as a sum or finding the maximum or minimum of a collection. The 'all reduce' is particularly useful as every process involved is given the result of the communication, i.e. there is no root process. See figure 3.4.

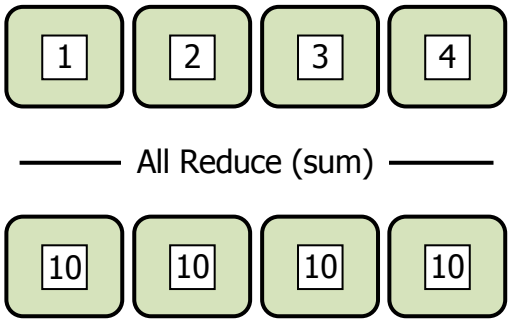

Figure 3.4: Graphical representation of the all reduce function

Many communications have 2 forms, blocking and non-blocking. This determines when a procedure calling a communication function will return, note this is not the same as a communication completing. The non-blocking form of a communication is paired with a 'wait' routine, this will then block progress through the code until the communication has completed. A non-blocking communication followed immediately by the corresponding wait is equivalent to the blocking form of the communication.

3.2.2 Synchronisation

Synchronisation is largely implicit, through the use of communications. When using synchronous point-to-point or collective communications, the processes involved are synchronised upon completing the communication. Threads involved in synchronous point-to-point communications must wait for a 'handshake' from the partner thread before proceeding to execute the message, ensuring synchronisation. In the case of collective communications although threads may enter the communication routine at different times, they have to wait for all other participating processes, before they collectively execute the communication. Data synchronisation will take place after all blocking communications, as once the communication completes the data used in the following computation will be correct. However, with the use of buffered messaging this does not ensure processes are synchronised in the execution of the code.

This data synchronisation must be carefully monitored when using non-blocking communications. When using non-blocking communications to interleave communication and computation, it is the programmer that becomes responsible for ensure the data being used in the computation is correct.

For explicit process synchronisation there is a collective barrier routine, which will only complete when all processes in the given communicator have entered. This will ensure that processes will be synchronised in point of execution but can have large overheads associated with it. This particular form of explicit synchronisation is rarely used in MPI programs, except in cases where a programmer uses it for diagnostic purposes, i.e. timing sections.

3.3 Hybrid

This chapter is based on information found at [6]. As discussed in the previous chapter, hybrid architectures are becoming more prominent in high performance computing; as such it follows that the hybridisation of the 2 previous programming models could be advantageous for such architectures.

In theory the hybrid scheme would be very successful on a clustered architecture. Where message passing is perfectly capable of being run across the PEs within the nodes in a cluster, it does not take advantage of explicitly sharing data and other useful functions provided by shared memory. The use of the shared memory model within a

node reduces the amount of explicit communication needed, and therefore the overheads incurred by such communication.

In the following section we will be discussing the hybrid model as implemented using the combination of MPI and OpenMP, as they were the APIs used for this project.

3.3.1 Hybrid Programming Model

In the hybrid the MPI nodes are used to distribute the workload whilst utilising threads to simultaneously compute any parallelisable code within the MPI process' workload. In its purest form the hybrid model consists of all of the PEs within a node being used with shared memory programming, whilst inter-node communication is implemented using the message passing model. A thread on the node will usually be responsible for communicating with other nodes by calling the MPI communication routines, usually this is the master thread. The master thread will then spawn additional threads were needed to utilise shared memory parallelism.

Whilst this may seem sensible it is often the case that this is not the most productive way to distribute MPI processes across the system. As described in chapter 2, shared memory architectures do not always have symmetric memory access but instead are often NUMA, this is particularly common for nodes used in clustered architectures. In this case it may be that a single MPI process may be placed in each NUMA region. The remaining PEs within that NUMA region will then be populated with threads spawned from the MPI process to fully utilise the performance of shared memory programming. Arranging processes and threads in this way ensures that all related threads have equal access to the shared memory resource.

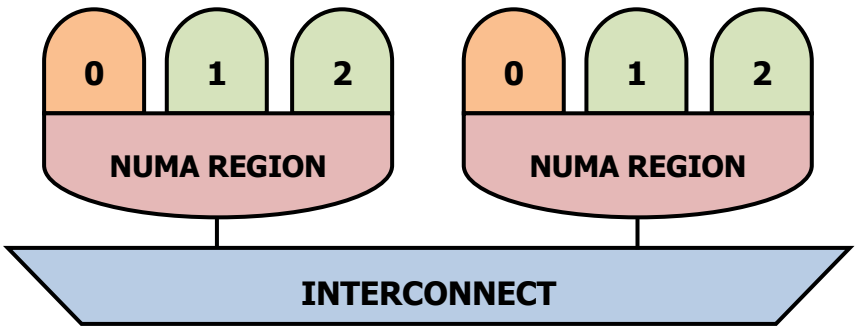

Figure 3.5: Threads running within a NUMA region, the master thread (orange PE, labelled 0) acting as the MPI process for the NUMA region

The thread performing inter-node communication via MPI does not have to be the master thread. Calling MPI does not need to be restricted to a single thread either, however implementing MPI communication on all of the threads adds considerable overheads. The various schemes available in hybrid programming can be summarised as follows:

- ***Master Only:*** All MPI communication takes place through the master thread outside of the OpenMP parallel regions.
- ***Funnelled:*** All MPI communication takes place through the master thread; this can be inside a parallel region.
- ***Serialised:*** MPI communication is performed by a single thread at any given time. In this case the sending/receiving threads can be distinguished using MPI tags or communicators.
- ***Multiple:*** MPI communication is performed simultaneously in multiple threads. It should be noted however, not all MPI implementations will support this form of hybrid parallelism.

The simplest schemes to implement are 'master only' and 'funnelled' as they can be implemented quickly on a code already parallelised using MPI by adding OpenMP directives. This allows shared memory parallelism to be added to sections of code with particular performance issues that may benefit from such additional parallelism, whilst keeping programmer overheads to a minimum. Code supplement 3.5 below shows how the 'master only' and 'funnelled' schemes differ in their implementation in code.

Master Only

```
1   !$OMP PARALLEL
2     computation
3   !$OMP ENDPARALLEL
4
5     call MPI_SEND(…)
6
7   !$OMP PARALLEL
8     computation
9   !$OMP END PARALLEL
```

Funnelled

```
!$OMP PARALLEL
  computation
!$OMP BARRIER
!$OMP MASTER
  call MPI_SEND(…)
!$OMP END MASTER
!$OMP BARRIER
  computation
!$OMP END PARALLEL
```

Code Supplement 3.5: Master Only and Funnelled hybrid schemes implemented in code.

This simplicity during implementation is not without cost however. Using either 'master only' or 'funnelled' hybrid parallelism, will leave the remaining threads idle whilst serial code is being executed. Also, having only a single thread performing MPI communication means the inter-node bandwidth may not be fully utilised.

3.3.2 Performance Considerations

Although in principle it is easy to create a hybrid code, particularly when starting from a code already parallelised using MPI, it can be difficult to reach a good level of performance. To fully utilise the addition of shared memory programming the entire code must implement threads, not just the key kernels. Often when moving from a solely MPI implementation to a hybrid implementation, the performance will dip. The programmer must then continue to work with the code to build the performance of the code up to a reasonable standard. This can occur for many reasons, such as the addition

of explicit synchronisation needed in shared memory programming, where with a strictly MPI implementation would only have the implicit synchronisation through messages. Having two levels of synchronisation also creates two levels of parallel overhead. It is therefore necessary to assess why a code may benefit from hybridisation. For this there are many aspects of a code that may indicate whether or not a hybrid code some be pursued. Here we will discuss a handful of common reasons for moving to a hybrid implementation.

MPI codes which use a lot of replicated data across the processes may benefit from hybridisation. The use of OpenMP allows the programmer to take advantage of shared memory, therefore reducing the amount of replicated data. Rather than having one copy of a key data structure per PE, we now require one copy per node or, depending on the distribution of MPI processes, per NUMA region. Now the data structure is shared, we can also reduce the amount of MPI buffer space reserved as intra-node messages are no longer required or, are significantly reduced.

If an MPI code scales poorly it may be the case that a hybrid code scales better. OpenMP may allow the code to scale better as the domain decomposition does not compliment MPI. This could be due to the load balancing of the domain, where OpenMP has much more flexible load balancing options. It is also possible that the domain is very simple and therefore a shared memory structure is more efficient, rather than a slew of messages to exchange large numbers of halos relative to the work load of an individual process.

Rather than the code being unable to scale to large numbers of MPI processes, it could be that the system being used doesn't give the capability to scale, or even that the implementation of the MPI library being used isn't able to handle the level of parallelism desired. This could be for many reasons, such as limit buffer space. The addition of OpenMP to an MPI code will in theory reduce the number of MPI processes required whilst maintaining similar levels of performance. Similarly whilst the MPI library implementation may be able to scale, it may not be optimised for the architecture on which it is being used, in this case a clustered architecture. Using the hybrid model, the programmer is effectively manually coding a more efficient way of performing, for example, an 'all reduce'. The threads within a node will locally reduce using shared memory, this is then followed by a reduction across the MPI processes, in doing so this decreases the number of messages that need to be sent, in theory reducing the overall overhead of the operation. This is true for many messages, aggregating smaller message into one large message to be sent between nodes, this reduces the effects latency has on the performance of the code.

Chapter 4

GS2: The Gyrokinetic code investigated

GS2 was the code investigated during this project. The development of GS2 was funded primarily by the United States Department of Energy, as part of the 'Scientific Discovery through Advanced Computing' program, and is currently used by researchers around the world. In this chapter we will cover a brief description of GS2 with more emphasis on the sections of the code worked with during the hybridisation process. We will also discuss the initial performance profiling of GS2, and how this provided the points of interest for the project.

4.1 GS2

Much of the information provided here is based on that found in [7], for a more complete description of GS2 please follow the link given in the references.

GS2 is used in several fields of physics to study low-frequency turbulence in magnetised plasma. These fields include natural plasmas, such as those found in astrophysical systems, and plasmas produced in a lab environment. It is more commonly used for laboratory purposes, assessing the microstability of the plasmas, along with calculating key properties of the turbulence which results from instabilities.

GS2 used Eulerian algorithms, these are a combination of spectral methods and grids in the 5-D phase space of gyrokinetics. It was the first code of its type able to "handle fully electromagnetic fluctuations with fully kinetic electrons in general non-circular tokamak geometry"[8]. It uses Eulerian algorithms and provides treatment of the multiple species, collisions and sheared flows found in the core region of tokamaks.

GS2 features fully gyrokinetic, nonlinear simulations and flexible simulation geometry. Flexible simulation geometry allows a range of assumptions to be made when carrying out linear and nonlinear calculations. When used on a parallel system, as the case is with this project, users can perform nonlinear simulations of fully developed turbulence. By taking advantage of the principles used in object-oriented programming, GS2 maintains portability. Each module may be exchanged for a system appropriate version. For example the communications routines used to implement the parallelism are contained within a single module, allowing different APIs to be used. Presently MPI and SHMEM are supported; along with the option of serial execution.

GS2 maintains good performance by independently calculating the linear and quasilinear proportions of the least damped eigenmode at a given wavenumber. Another factor in the impressive performance of GS2 is its efficiency when dealing with turbulent structures. Using coordinates following the magnetic field lines of the plasma, the structures can be resolved in a flux tube of modest extent, with very high efficiency. The program can also maintain this performance when scaling as separate optimisations are used depending on the size of the system used, see figure 4.1.

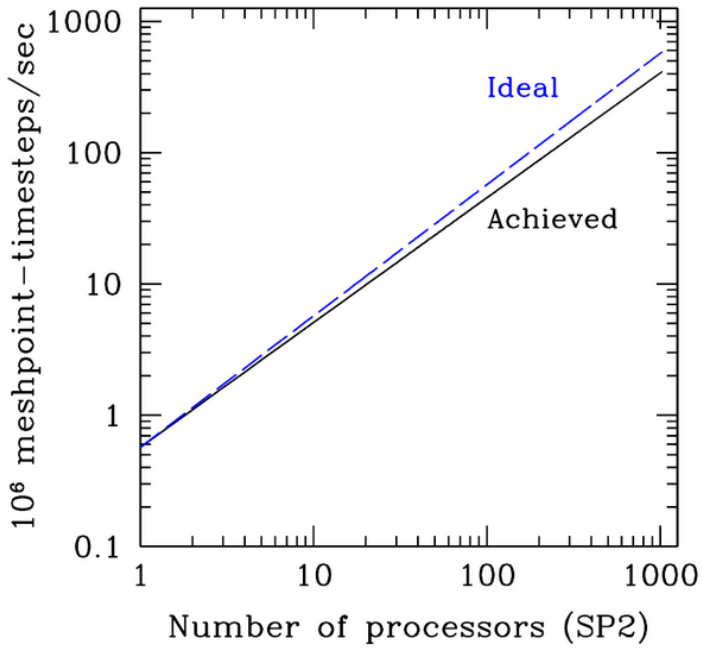

Figure 4.1: Graph comparing the achieved parallel performance of GS2 with the ideal parallel performance. Source for graph [9].

Running GS2 requires an input file with the extension '.in', the name of this file is used as the reference for a given simulation. The input file consists of a series of 'namelists', each namelist specifying various parameters. A comprehensive list of these namelists and their parameters can be found at [10]. Given the number of namelists it is often more sensible to take an existing input file and modify it to match the users requirements. An example input file can be found in appendix A.

4.2 The 'timeadv' routine

Our main concern throughout this project was with the 'timeadv' routine, found in the 'dist_fn' module. The main purpose of this module is to advance the discrete gyrokinetic equation by calculating the source and dealing with the parallel boundary conditions. Whilst the computational methods used in the timeadv function are beyond

the scope of this project, the structure and execution path of the function provide insight into areas of the code to which the addition of threaded parallelism is viable. Therefore this section will contain an overview of the structure of the timeadv routine.

The timeadv function calculates the distribution function at the next timestep of the simulation. The function is relatively short, consisting of 23 lines of "active" code, which is code not including comments, routine boundarys and blank lines. It is comprised of 4 main function calls; add_explicit_terms, invert_rhs, hyper_diff and vspace_derivatives. There purposes can be summarised as follows:

- ***add_explicit_terms:*** Calculates the explicit nonlinear terms.
- ***invert_rhs:*** Responsible for the actual evolution of the distribution function.
- ***hyper_diff:*** Adds hyper diffusions if present.
- ***vspace_derivatives:*** Adds collisions if present.

These functions are then followed by calls to redistribute and enforce parity if required.

Due to the modular, object-oriented style of GS2 each function call above continues to call subroutines down a call tree until reaching the routines containing the computational intensive sections, see figure 4.2.

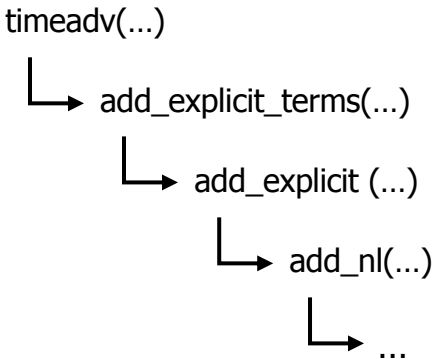

Figure 4.2: Example of a call tree in the timeadv routine

4.3 Initial Benchmarking and Profiling

We began by doing some basic benchmarking of GS2, to assess the general performance of the code, prior to any changes made. In order to discern routines in GS2 in which large amounts of execution time is spent, performance profiling was then undertaken. This would then shed some light on the sections of code that when parallelised using threads, would provide the most benefit to the performance of the code.

4.3.1 Benchmarking

We began by accessing the execution time relative to the number of processes used. These runs were performed on the HECToR system, see chapter 5 for details. The test case used throughout this project is provided as appendix A. Initial tests showed that execution time could vary slightly for identical runs, to provide more reliable data points, each run was performed 3 times and the values obtained where then averaged.

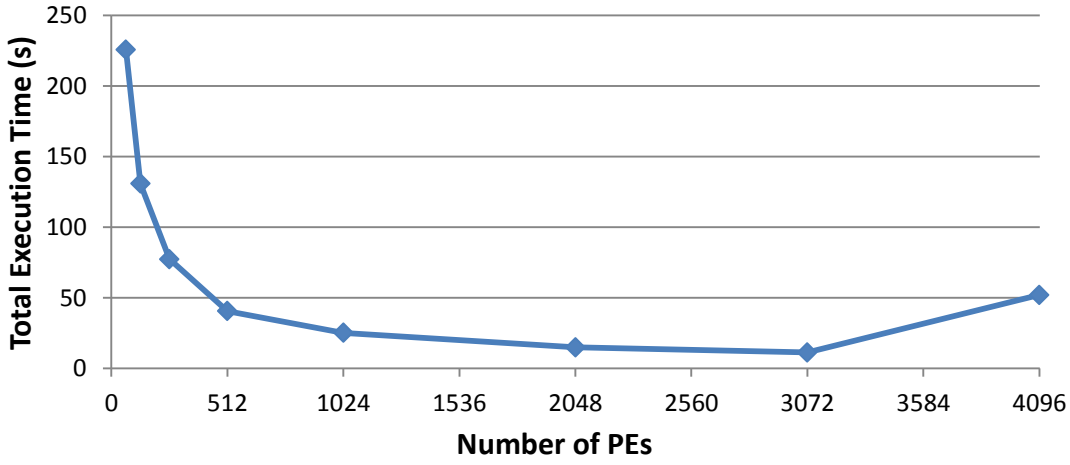

Figure 4.3: Graph showing the execution time of GS2 for 1000 time steps using varying numbers of PEs, initial data point is 64 PEs

Figure 4.3 shows us that the execution time for GS2 reduces as the PE number increases up to 2048 where we see a dramatic increase. Closer inspection of the data shows that for 4096 PEs the initialisation procedure continues to decrease as the trend would suggest. However, the execution time for advancing the time step and solving the fields increases.

The speed up factor and parallel efficiency of GS2 are shown in figures 4.4 and 4.5 below. Although the standard distribution of GS2 is capable of running in serial, we could not run this particular version of GS2 in serial. Therefore data points given for speed up where given by dividing the execution time for 64 PEs by the execution time gained for *P* PEs, rather than the time for running GS2 in serial on a single PE. Parallel efficiency is given by taking the speedup for *P* PEs and dividing it by *P*.

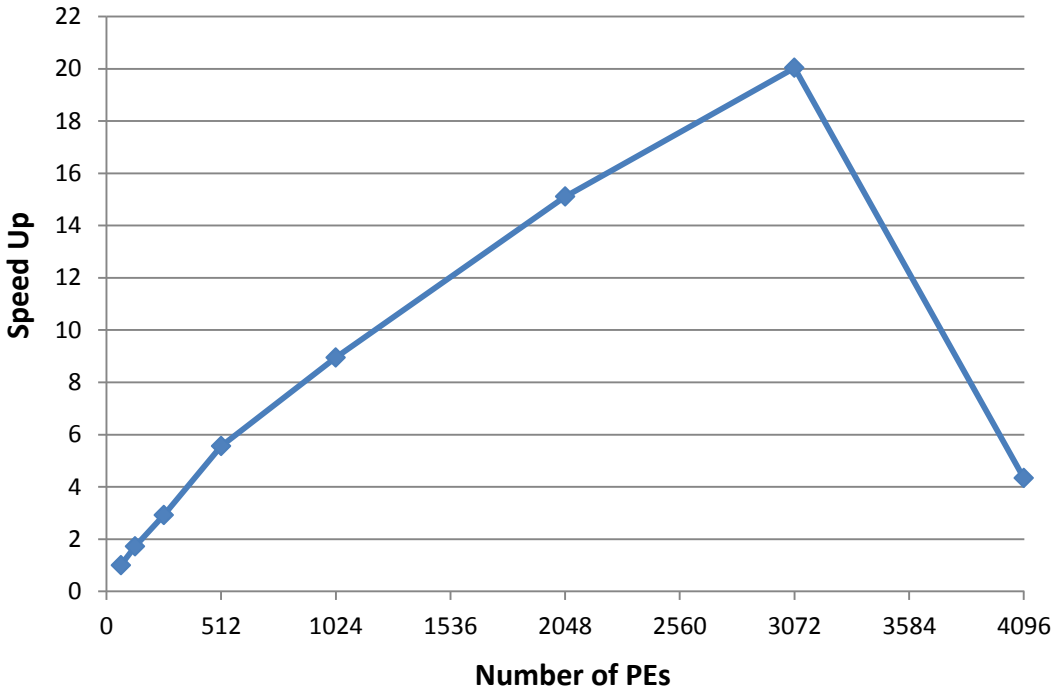

Figure 4.4: Graph showing the speed up of GS2 for 1000 time steps using varying numbers of PEs, initial data point is 64 PEs

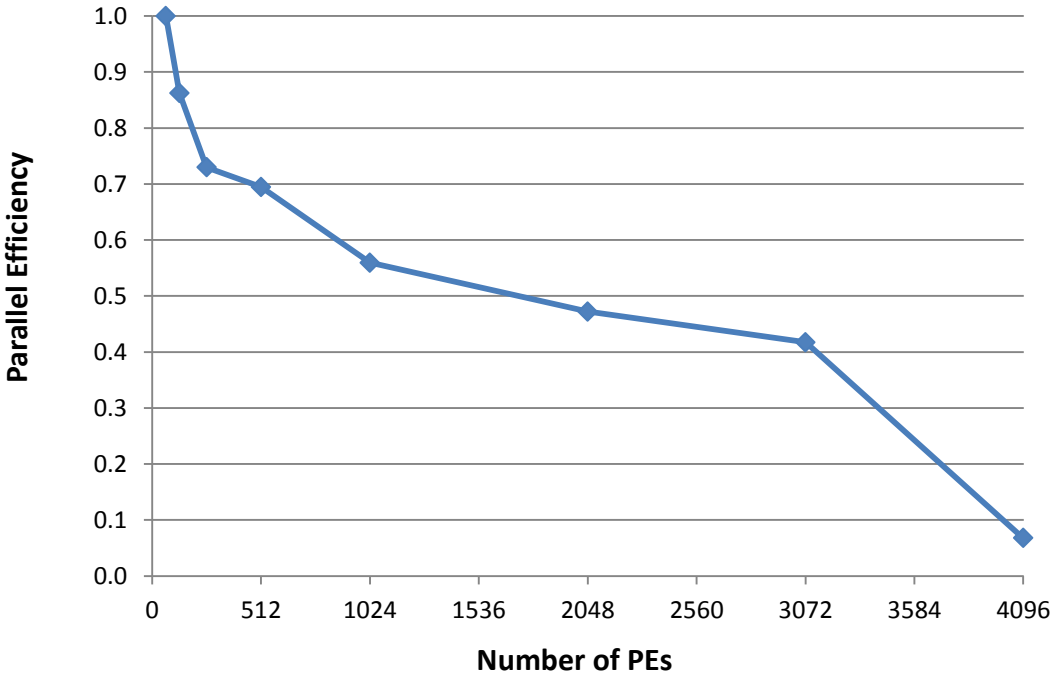

Figure 4.5: Graph showing the parallel efficiency of GS2 for 1000 time steps using varying numbers of PEs, data has been re-biased to begin at 1 for 64 PEs

4.3.2 Scalasca Profiling

GS2 comes with optional Scalasca support, activated by a variable in the primary makefile before building the code. This allows us to view a detailed performance profile, showing routines which are performance hotspots. Figure 4.6 shows that at the highest level of the call tree the execution time hotspots are contained in two separate functions, namely ‘init_fields’ and ‘advance’. It should be noted that the ‘init_fields’ routine, is part of the initialisation procedure for GS2. Therefore, it is only run once during the total execution time. Whereas the ‘advance’ procedure will be called proportionally to the number of steps in the simulation. With this in mind the ‘advance’ procedure will dominate execution time for real-world simulations, often involving a large number of steps.

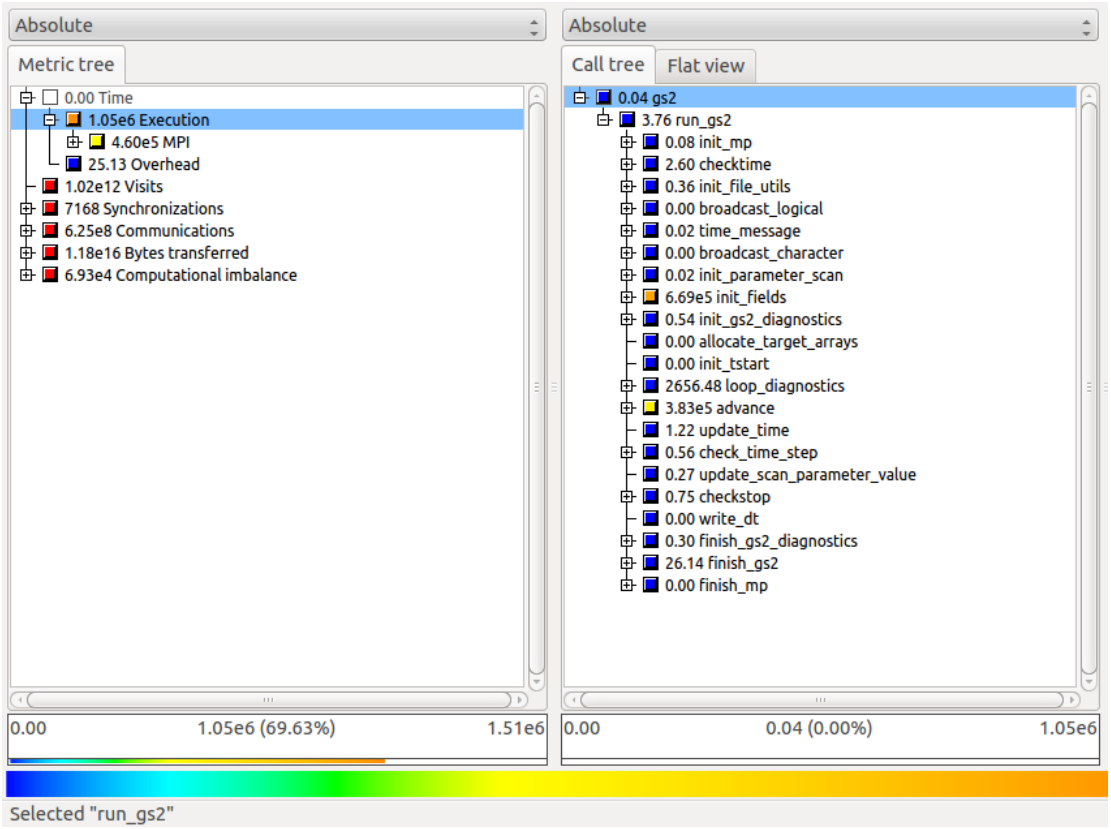

Figure 4.6: Top level Scalasca report, showing hotspots for execution time.

However, when the call trees are investigated further, we see that much of the execution time within these two functions can be attributed to the same subroutine, the ‘timeadv’ routine, see figure 4.7.

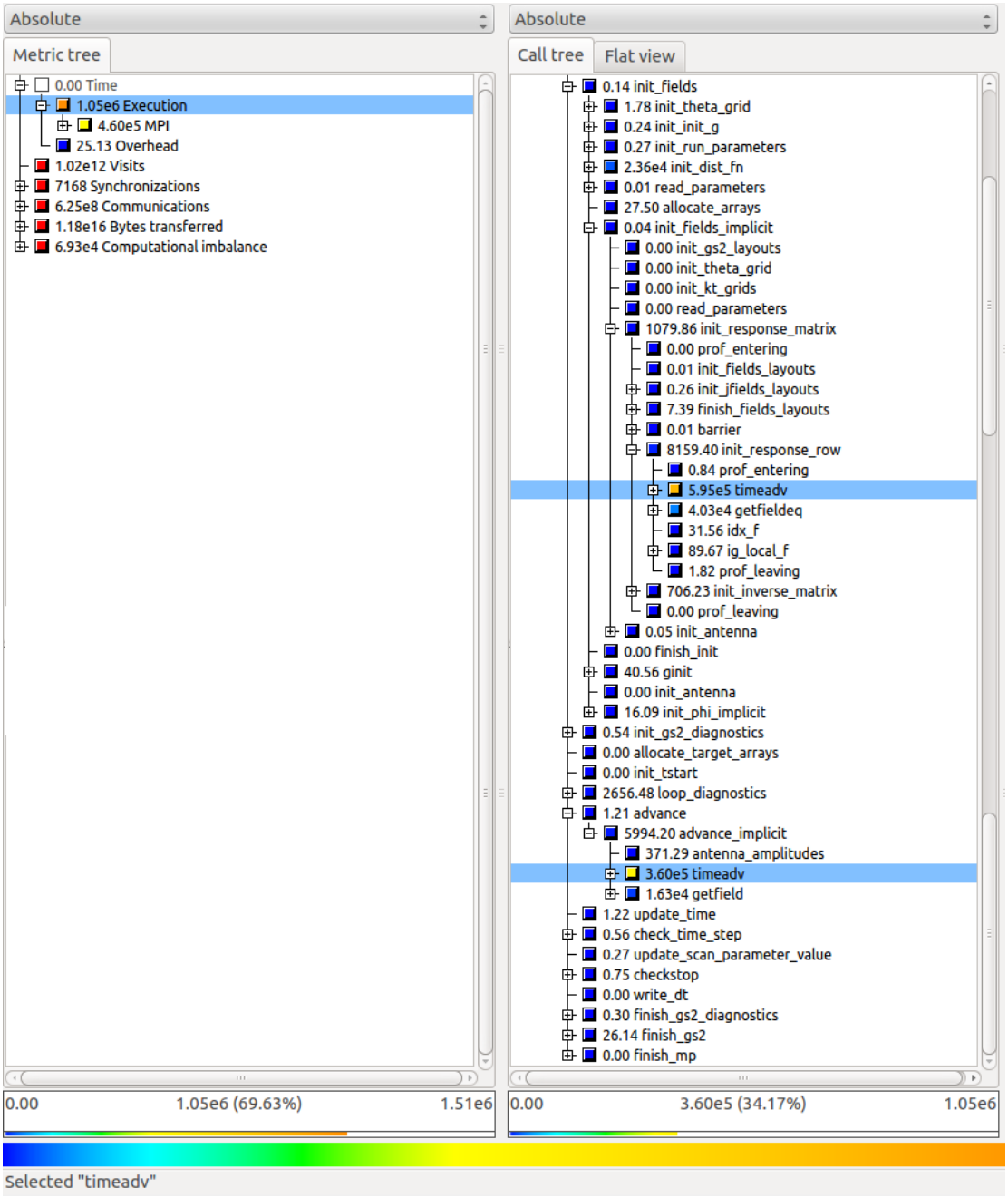

Figure 4.7: Extended call trees in Scalasca output, showing the 'timeadv' function responsible in both cases for the large execution time.

Moving further down through the Scalasca tree we find that the following routines take up the highest percentages of execution time:

- add_nl
- transform2_5d_accel
- get_source_term
- solfp_ediffuse
- conserve_diffuse
- solfp_lorentz
- conserve_lorentz

4.3.3 Communications and Data Distribution

We can see from figure 4.8 that MPI communications in GS2 take up 30% of the total run time. With the addition of OpenMP threads, there will be reduction in MPI processes. This should theoretically reduce the number of messages passed throughout the run time.

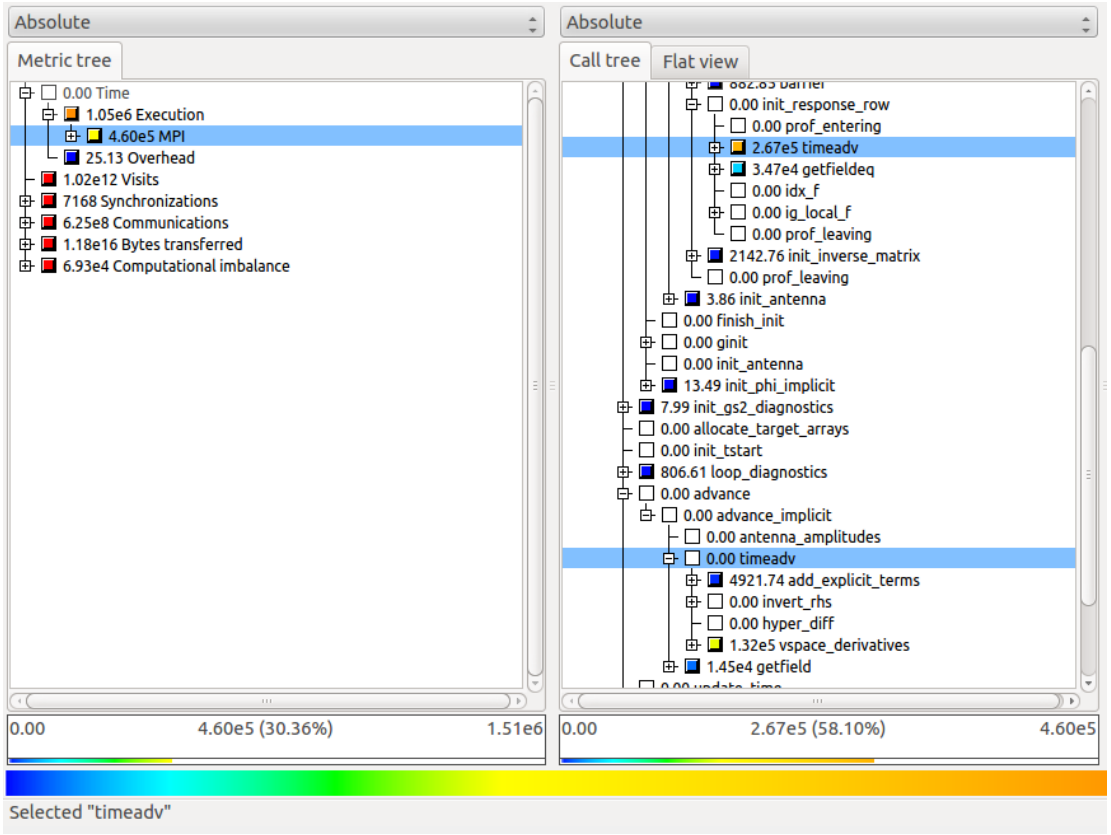

Figure 4.8: Scalasca screen showing the proportional time consumed by MPI communications, and routines in which communications are most commonly called.

Along with this, the figure 4.9 shows the amount of data transferred throughout the run time. With a reduction in messages there would also be a reduction in data transfer. With less data transferred, the program will be less susceptible to overheads caused by bandwidth constraints.

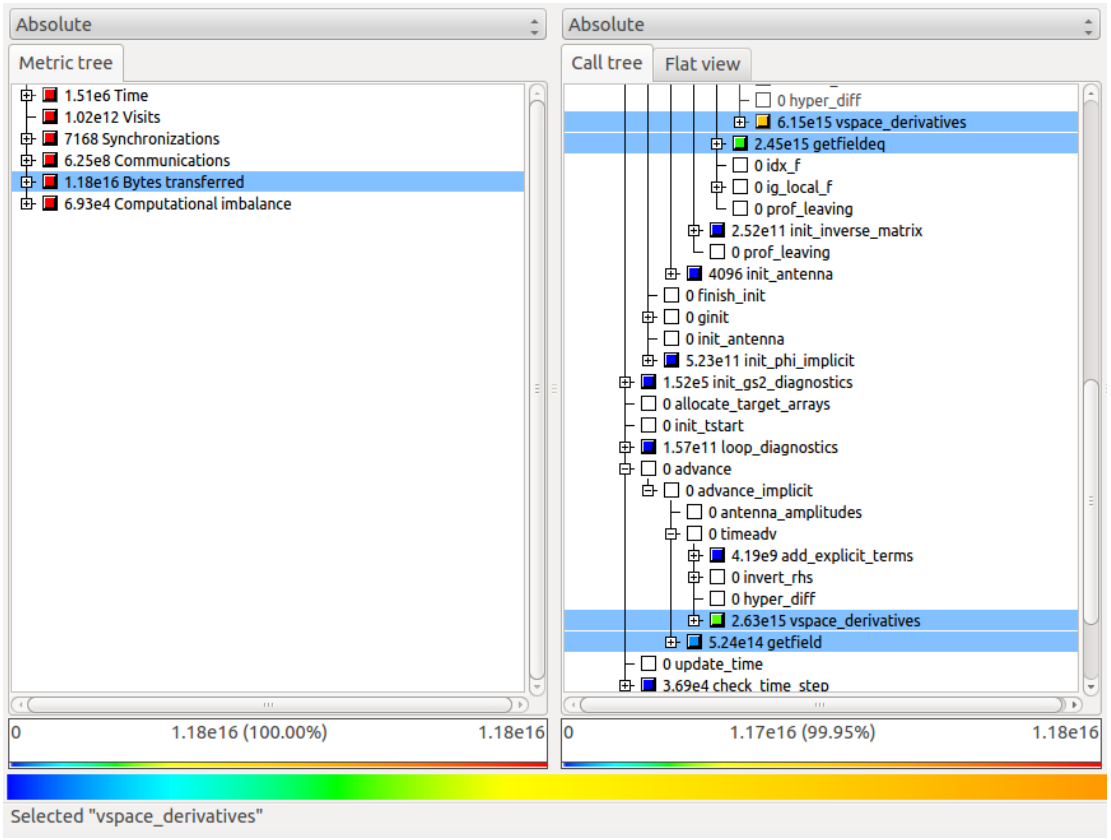

Figure 4.9: Scalasca screen showing the amount of data transferred throughout the run time of GS2, along with the routines involved in transferring the higher proportions of this data.

It has been shown that at large process counts the remote copy functionality of GS2 has a significant effect on performance.[11] This is one of the motivations to implement the hybrid model within GS2. As it will keep process counts lower, therefore there will be less data distribution across the system. However, unlike simply running the code with a lower number of processes, we retain the compute power in the form of additional threads. The larger chunks of the distributed data can then be work on in shared memory between all of the threads, requiring less message passing and there suffering less of the overheads associated with it.

Chapter 5

Implementation

This chapter describes the implementation of threaded parallelism in the GS2 code, along with the systems that were used to develop and test the additions. The Culham Centre for Fusion Energy development version of GS2 was used for this project. Given the time constraints of the project a complete threaded implementation of GS2 was out of the project scope. Work was therefore done on the sections of code which took large amounts of execution time; the hope was to reduce the overall execution time of the program. After examining the code in computationally heavy routines, sections that lend themselves to threaded programming where identified.

5.1 System Used

This section describes the high performance computing system, HECToR, used for this project. The description presented in the section is derived from [12], for a more information on the system please follow the link provided in the references. At time of writing, HECToR is the system used for the UK national supercomputing service. HECToR uses the Cray Linux Environment as its operating system, along with a variety of compilers including PGI, GNU and Cray compilers.

It is currently in Phase 3 and is a Cray XE6 system, with a total of 704 compute blades. With a total of 2816 compute nodes, each with 2 16-core AMD Opteron 2.3 GHz Interlagos processors. This gives a total of 90,112 cores, with a theoretical peak of over 800 TFlops. The interlagos processor is arranged into 2 NUMA regions. Each 16-core socket is paired with a Cray Gemini routing and communications chip. For every 2 XE nodes there is a Gemini router chip, each chip has 10 network links used to implement a 3D-torus.

The system has a total of approximately 90TB of memory, distributed amongst the processors as 16GB of memory per 16-core processor (8GB of memory per NUMA region). HECToR is equipped with over 1PB of high-performance RAID disks, accessible from any node using the Lustre distributed parallel file system. Along with this HECToR also has a tape based backup system, with a maximum capacity of approximately 1.02 PB.

5.2 Introducing OpenMP

We began introducing threads by using the OpenMP ‘parallel do’ to parallelise the do loops found in the previously mentioned chapter. This followed the master only scheme in hybrid programming. However, this proved to be inefficient as threads had to be repeatedly initialised, incurring additional overheads for each parallel region. We therefore decided to use the funnelled approach to hybrid programming and have a single parallel region per routine, implementing a master region where necessary for MPI communications along with code note suitable to parallel execution. This was largely successful as there are large bodies of GS2 involved loops which can be easily parallelised using threads.

In having only a single parallel region per routine we also took advantage of were to begin the region. With this in mind we positioned the parallel region as far into the routine, until reaching an appropriate parallelisable region. This was to ensure as much serial code could be executed as possible before initialising the threads and forcing the serial code into a master directive. This allows the PE on which the MPI process is running to take advantage of the extra resources available to it when its adjoining PEs are idle. Also it relieves the need for extra explicit OpenMP barrier routines at the end of a master region, therefore avoiding unnecessary overheads.

During the implementation of the OpenMP directives the decision was made to use the ‘default (none)’ clause for every parallel region. This prevented automatic variable casting to private or shared, and produced and error on compilation if a variable had not been explicitly declared within the shared or private clauses. This proved to be a useful tool in error detection, when a variable had been missed with the region. We also decided to by default the ‘schedule’ clause to each ‘do’ directive to allow for quick refactoring of the code in situations where different schedules are desired.

While ideally all of the hotspot routines mentioned in the previous chapter would be threaded. We successfully threaded all of these routines bar the ‘transform2_5d_accel’. Unfortunately, the main computational loop in this routine (shown in code supplement 5.1) contained a variable with a unique value for each loop iteration.

```
1   idx = g_lo%llim_proc
2   do k = accel_lo%llim_proc, accel_lo%ulim_proc
3    if (aidx(k)) then
4      ag(:,:,k) = 0.0
5    else
6      if (ik_idx(g_lo, idx) .ne. 1) then
7        do iduo = 1, 2
8          do itgrid = 1, 2*ntgrid +1
9            g(itgrid, iduo, idx) = 0.5 * g(itgrid, iduo, idx)
10           ag(itgrid - (ntgrid+1), iduo, k) = g(itgrid, iduo, idx)
11         enddo
12       enddo
13     else
14       do iduo = 1, 2
15         do itgrid = 1, 2*ntgrid+1
16           ag(itgrid-(ntgrid+1), iduo, k) = g(itgrid, iduo, idx)
17         enddo
18       enddo
19     endif
20     idx = idx + 1
21   endif
22  enddo
```

Code Supplement 5.1: The main computational loop within the transfrom2_5d_accel routine (not including comments)

The ‘idx’ variable is used in every iteration, and then continues to be used after it has been set for the final time at the end of the loop. As each thread would receive a different set of iterations the ‘idx’ variable would need to be set to the correct starting value for each thread. However, as we see in line 1 of code supplement 5.1, prior to entering the loop ‘idx’ is set to a value unknown until run time. While it may be possible to create a firstprivate version of ‘idx’ for the thread executing the first set of iterations, this would not suit for the other threads executing iterations further in the loop. Not only this but the variable does not simply increment with each iteration of the loop, it is part of the conditional statement, therefore threads executing later iterations could not assume what the value would be. Although I’m sure a solution could be conceived, due to time constraints this was not possible in this instance.

Another problematic routine was the ‘integrate_moment_c34’ function. It is repeatedly called in functions such as ‘conserve_lorentz’ in between the computational loops, and is involved in redistributing data across the processes by calling functions in the communication module. It would be highly inefficient to close the parallel region before each call to this function, to then have a separate parallel region within the function; as such we decided to attempt implementing orphaned OpenMP directives. Orphaned directives are directives that are used within the dynamic scope of a parallel region but are not in the lexical scope of a parallel region. However, by using orphaned

directives we have less direct control over declared variables as shared or private. This created a particular issue with the ‘total_small’, which we originally had as a reduction across a loop that had been threaded. However, for the reduction clause to be used a variable must be shared, from which private are copies are made for use within the work sharing region. By default variables declared inside a function with orphaned directives are private. We attempted to solve this problem in several ways, including declaring ‘total_small’ as a global variable, which are shared by default in when used in orphaned directives. But all attempts to rectify the issues gave erroneous results when tested, therefore we simply surrounded the calls to the integrate moment function with the master directive. This ensured communication continued through the master thread as well as keeping efficiency up by not having to enter and exit parallel regions.

5.2.1 Thread Ratio

Once threads where introduced to GS2 we proceeded to test the most effective ratio of OpenMP threads to MPI processes. We began by devoting an entire NUMA region to each thread; this is equivalent to one process to eight threads. We then continued to reduce this number by half until each process had two threads. Figure 5.1 gives a breakdown of the data gained.

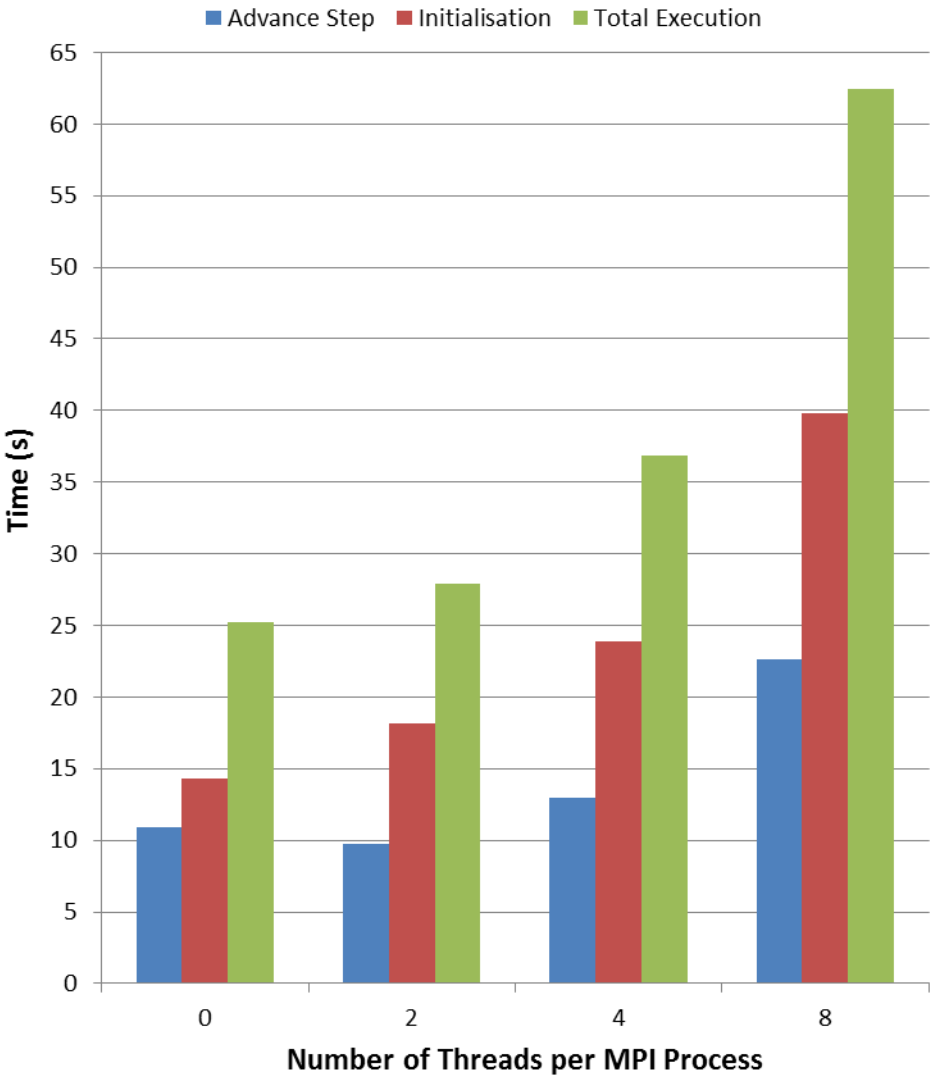

Figure 5.1: Chart showing time spent in advance steps, initialisation and total execution time against the number of threads per MPI process for the hybrid implementation of GS2 running 1000 time steps

As we can see from figure 5.1 that for all implementations using threads the total execution time for 1000 time steps is longer than the MPI only version of GS2. However, when run using two threads per process while the initialisation and total execution is longer, the time for advance steps is slightly shorter. Given that the initialisation is an overhead that should remain the same regardless of the number of time steps, and the advance step time will is proportional to the number of steps, the small reduction in advance step given by the hybrid implementation could cause a significant reduction in total execution time for runs with a large numbers of time steps. We therefore decided to proceed with two threads per process for further testing.

Chapter 6

Results and Analysis

This section covers the results gained from running the hybrid and MPI-Only versions of GS2 under varying workloads and across multiple numbers of PEs. These tests were run using the test case provided in appendix A as previously stated.

6.1 Scaling Performance

The following section discusses the results gained from strong scaling tests. Strong scaling involves the global volume to be fixed while increasing the number of PEs used to execute that workload. The performance results in this section were all gained using 1000 time steps.

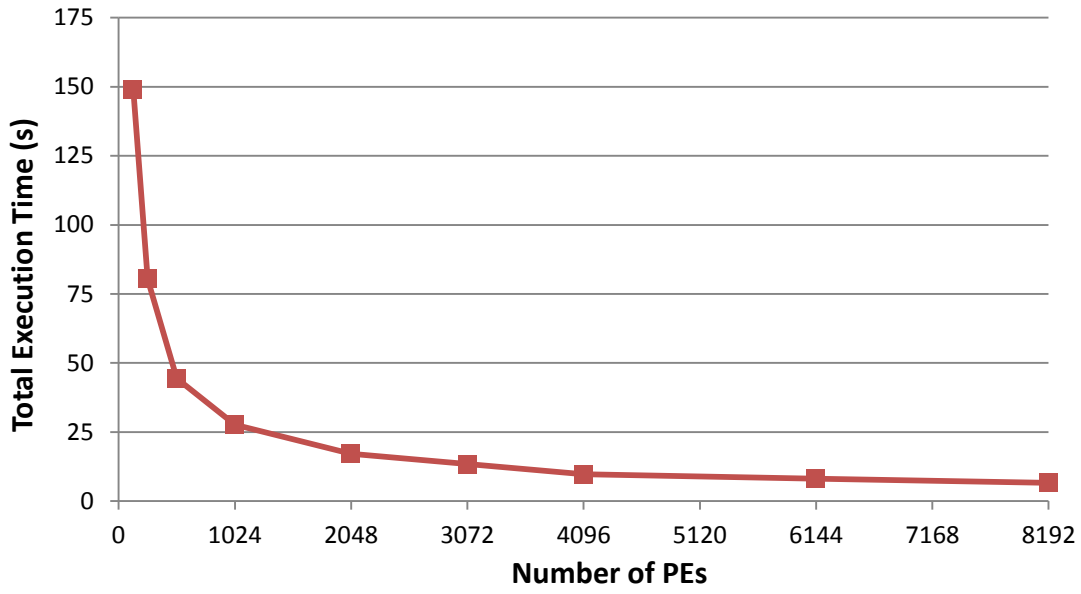

Figure 6.1: Graph showing the Total Execution time for Hybrid GS2 across varying numbers of PEs for 1000 steps

Figure 6.1 shows us that the hybrid implementation of GS2 continues to scale well to large number of PEs. When compared to figure 6.2 we can see that the MPI-Only version of GS2 begins to increase in execution time from 4096 PEs. We can also see that up until 4096 PEs the hybrid codes performance is, only slightly worse than the MPI-Only version.

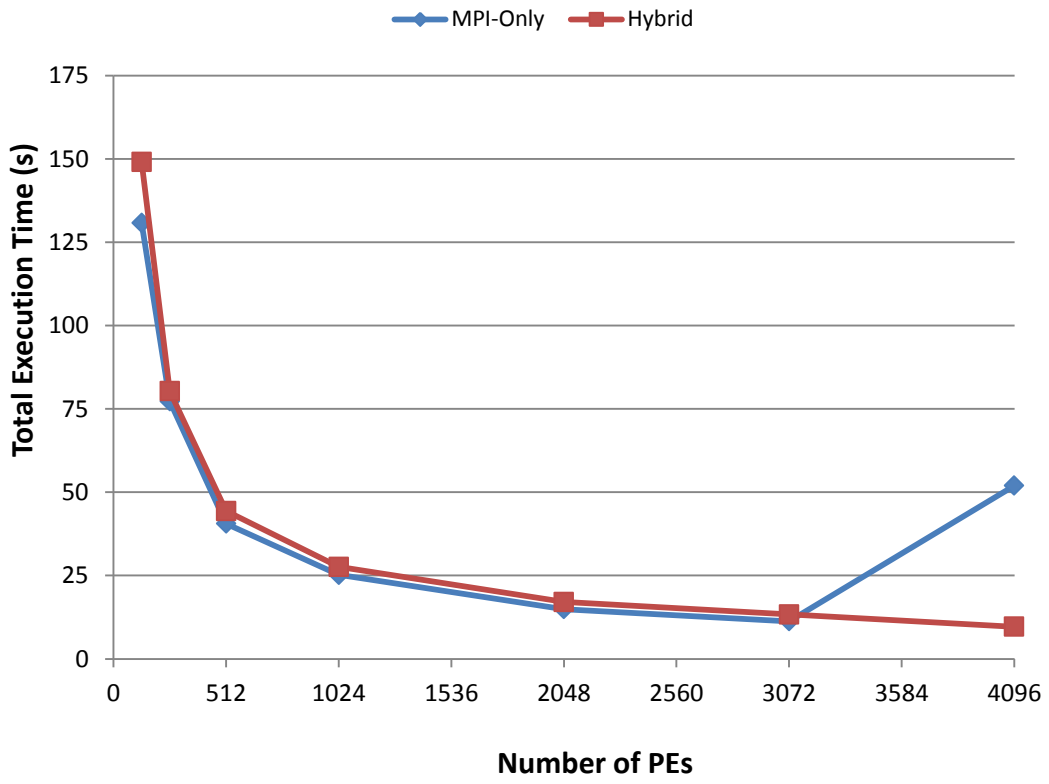

Figure 6.2: Graph showing the Total Execution time for both Hybrid and MPI-Only GS2 against the number of PEs used. Initial data point is at 128 PEs.

As we can see from figure 6.3, the speed up for the hybrid code remains fairly linear where the MPI-only code decreases dramatically at 4096 PEs.

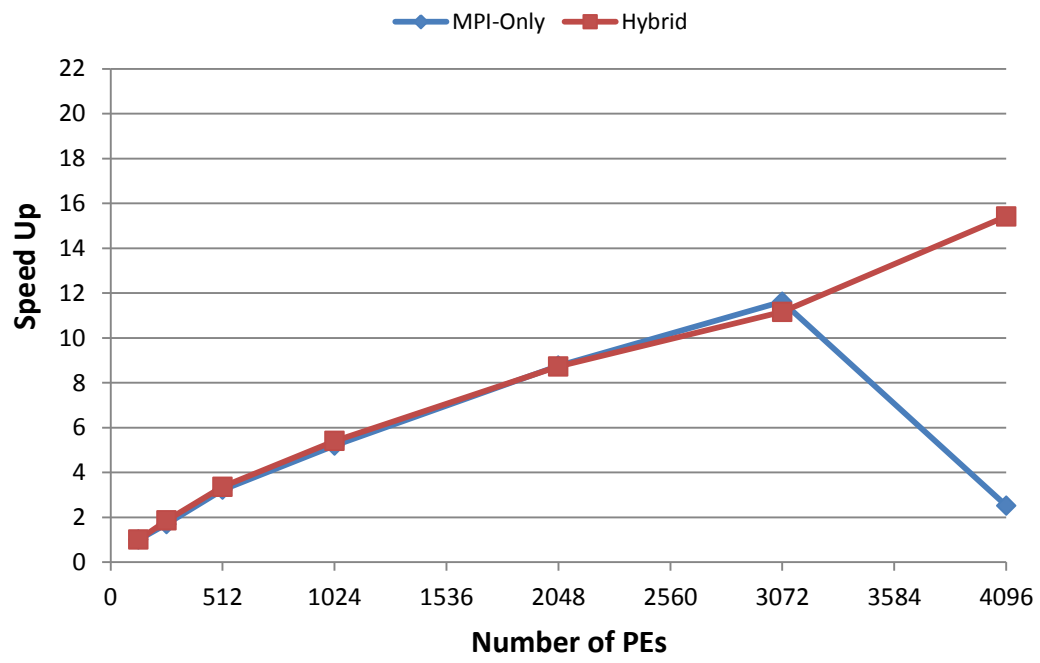

Figure 6.3: Graph showing the speed up against number of PEs used for both Hybrid and MPI-Only GS2. Initial data point is at 64 PEs.

We again see this pattern at 4096 PEs as figure 6.4 shows the parallel efficiency of the Hybrid code levelling, whereas the MPI-only code continues declining.

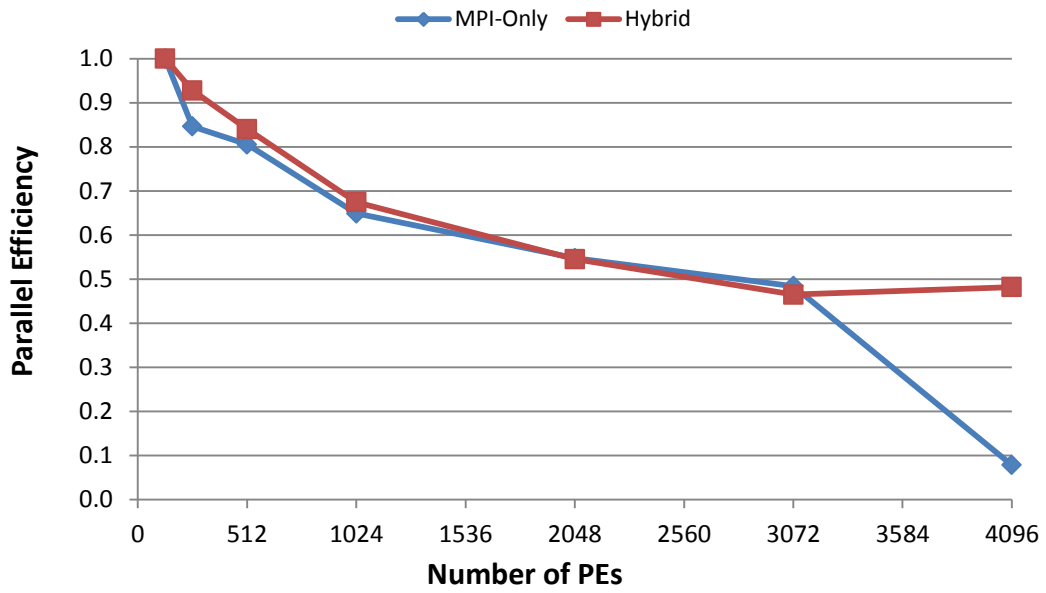

Figure 6.4: Graph showing Parallel Efficient against number of PEs used for both Hybrid and MPI-Only GS2. Initial data point is at 64 PEs, data has been re-biased to begin at 1.

Whilst figure 6.2 shows the comparison of performance based on the computing resources used, figure 6.5 gives a performance comparison based on the number of MPI processes used. It could be argued that this gives us a better comparison of performance as the data distribution amongst the MPI processes will be equal.

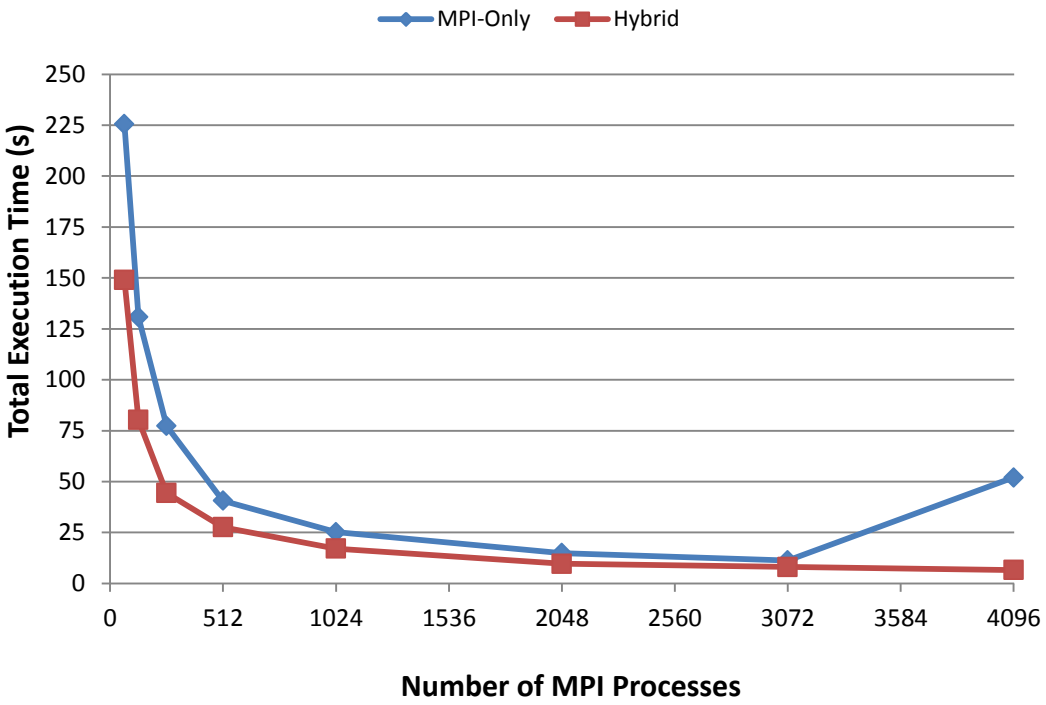

Figure 6.5 : Graph showing the Total execution time against the number of MPI processes used for both Hybrid and MPI-Only GS2. Initial data point is at 64 MPI processes.

As we can see in figure 6.5 the hybrid code consistently runs faster than the MPI-only code when using the same number of MPI processes. We can also see very similar patterns emerging from figures 6.6 and 6.7.

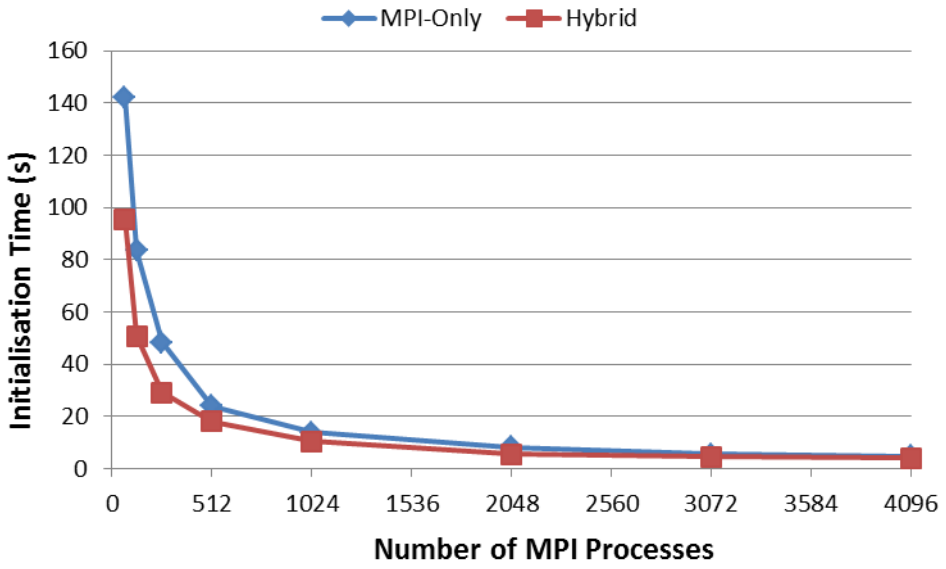

Figure 6.6: Graph Showing the Initialisation time against Number of MPI processes for both MPI-Only and Hybrid GS2.

In figure 6.6 we see that for lower MPI process counts, the hybrid code initialises faster than the MPI-only version. However, as they approach 4096 processes they converge. This indicates that the section of code allowing the hybrid implementation to outperform the MPI implementation is not in the initialisation phase of the program.

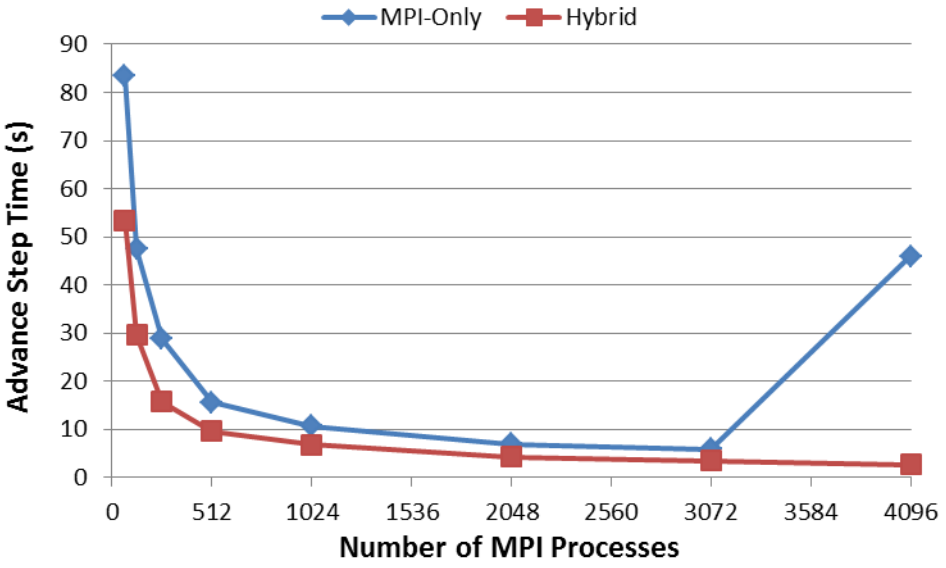

Figure 6.7: Graph Showing the Advance Step time against Number of MPI processes for both MPI-Only and Hybrid GS2.

Figure 6.7 provides some insight into why the hybrid code begins to drastically outperform the MPI-only code. As we can see moving towards 3072 processes, the two

implementations converge, with the hybrid implementation being slightly faster. However, upon reaching 4096 processes the MPI-only code drastically increases in advance step execution time, while the hybrid code continues to plateau. The data show in figure 6.7, along with the similarity in the graph shapes with figure 6.5, suggests that the advance step execution time is indeed a deciding factor in the total execution time of GS2, and when using equal numbers of MPI processes, the hybrid code outperforms the MPI-only code. Although, it may be the case that the addition of threads to the code provided this performance increase, it may also be the case that the performance was gained simply through under population of the node.

Under population of a node in clustered architecture is simply running on fewer than the total number of PEs on the node. Often this will be fraction of the node such as one half, and the work will be spread evenly across the node. This can often provide the active PEs with the resources normally used by an adjoining PE, such as the floating point unit or shared cache.

6.2 Workload Performance

In this section we will discuss the results of the workload testing we performed. The workload testing for this project took the form of varying the number of steps to be completed for a single execution of GS2. Whilst testing this aspect of the program, we used 1024 PEs for both the MPI-only and hybrid version. This meant that the hybrid version is running with 512 MPI processes, with each process being coupled with two OpenMP threads. For example submission scripts see appendix B. We test a range from 1000 to 10,000 steps, incrementing by 1000 until 5000, we then increase to 10,000.

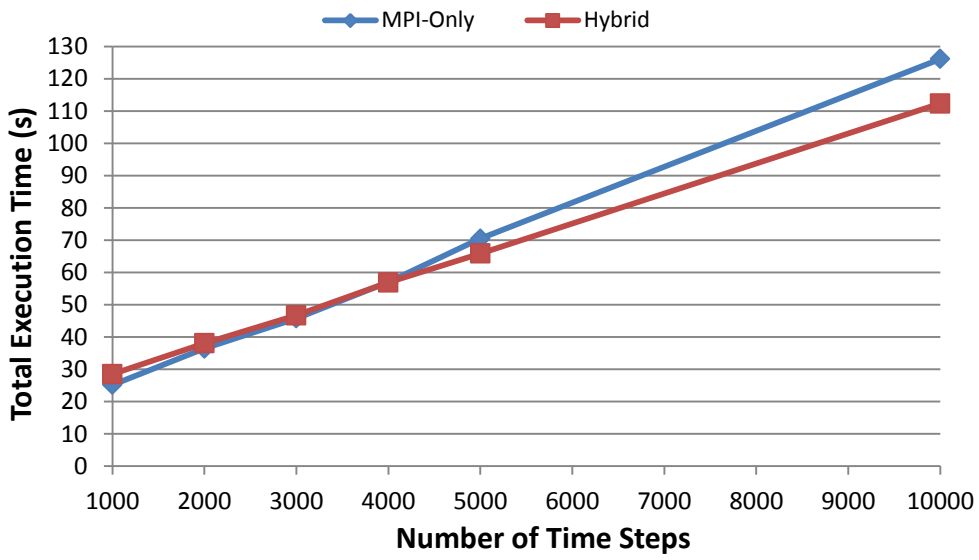

Figure 6.8: Graph showing Total Execution time against Number of time steps for both Hybrid and MPI-Only GS2.

We can see in figure 6.8 that from 1000 to 4000 the executions times converge, with the hybrid implementation being slightly slower than the MPI-only implementation. The execution times then begin to diverge, with the hybrid implementation outperforming the MPI-only implementation.

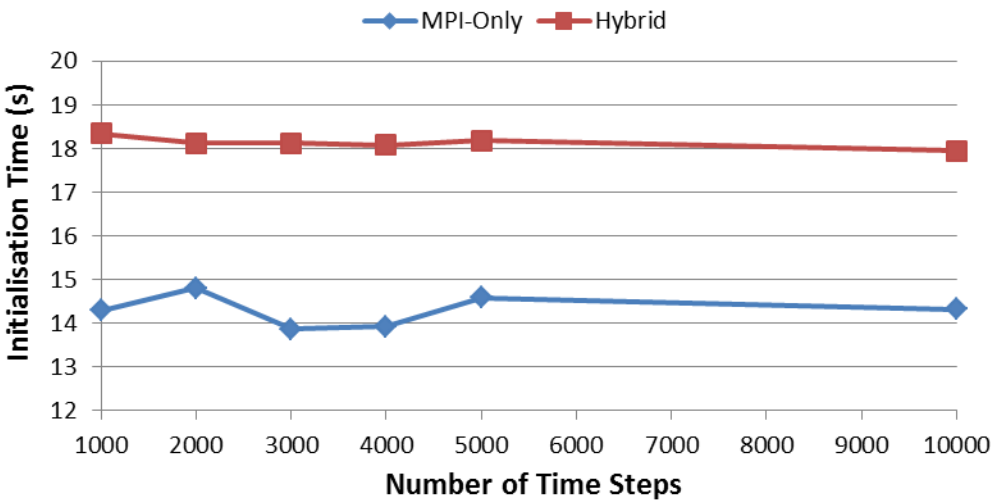

Figure 6.9: Graph showing the Initialisation time against the number of time steps for both Hybrid and MPI-Only GS2.

Figure 6.9 shows the time taken for the initialisation of GS2 across the varying number of time steps. The initialisation time is fairly stable across all step numbers for both versions of GS2. This is as we would expect as the initialisation is a single overhead. This data also indicates that initialisation is not proportional to the number of steps in a run. Figure 6.9 clearly shows that the initialisation for the hybrid implementation takes longer than for the MPI-only implementation. This coupled with what we see in figure 6.10 goes some way in explaining the graph we see in figure 6.8.

As we can see in figure 6.10 the advance step time for both the hybrid and MPI-only implementations are equal for 1000 steps, but immediately begin to diverge with the MPI-only implementation taking longer. The total execution time for the hybrid implementation would higher than the MPI-only implementation, until the gains given by the advance step code outweighs the additional initialisation overhead, which is what we see happening in figure 6.8 at 4000 steps.

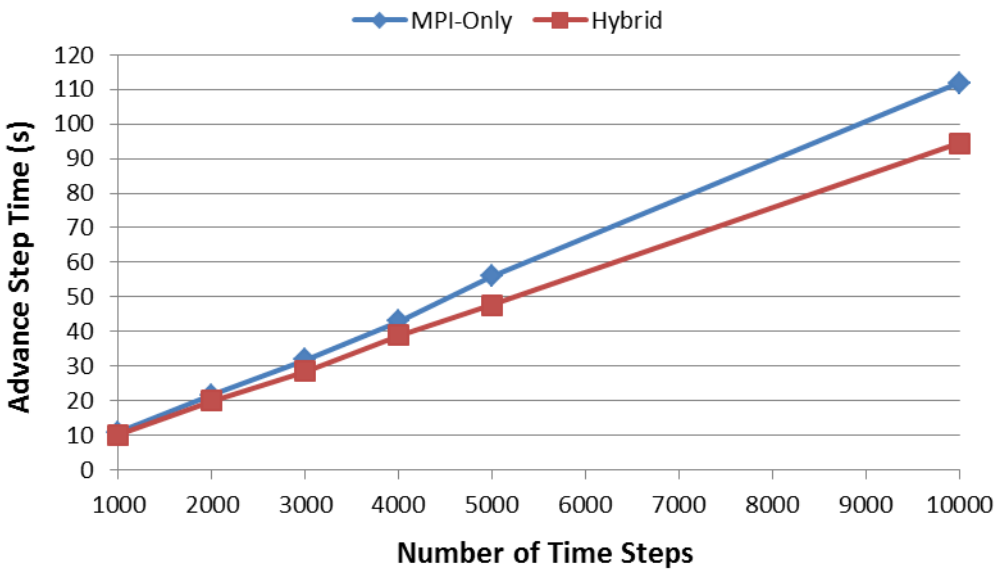

Figure 6.10: Graph showing the Advance Step time against the number of time steps for both Hybrid and MPI-Only GS2.

As we mentioned, figure 6.10 shows the advance step time slowly diverging, this indicates a marginal decrease in execution time for an individual time step. This would therefore produce a cumulative effect that gradually increases the performance gap. However, as previously stated, we are unable to tell whether this is a genuine increase due to the OpenMP additions or due to under population of the nodes with regards to the MPI processes.

Chapter 7

Conclusions

At the beginning of this project we set out to optimise the GS2 gyrokinetic simulation code through the use of the hybrid programming model. This involved adding threaded parallelism to key functions in the existing code base, which already used message-passing parallelism in the form of the MPI library. Our main aim was to access any improvements in performance that might occur due to the addition of OpenMP directives.

For the project we used the CCFE test version of GS2, which presented some issues in throughout the life cycle of the project. One such issue was the inability to run GS2 in serial. However, there were no problems serious enough to deadlock the project. After a detail performance profiling we isolated several routines that were responsible for large amounts of execution time, many of which were suitable for threaded parallelism (section 4).

Upon completing the addition of OpenMP directives to the code, we then set about testing the ratios of OpenMP threads to MPI processes. While it was initially thought that making use of an entire NUMA region would be beneficial, maximising the use of shared memory within a node, this was not the case. The best performing ratio was a ratio of 2:1 threads to processes. After looking into the data used in creating loop bounds, it was found that often the loops that had been parallelised would have very few iterations. This short fall in iterations would make it very inefficient to distribute over a large number of threads, and goes some way in explaining why we got the performance results we did.

Whilst we did see a slight improvement in the overall execution time for GS2 given a large enough workload and/or PEs used, I do not believe we can be certain that the additional parallelism is responsible. As we have stated previously in this report there are other factors at work that may explain the performance changes we have seen. That being said, I think many more routines in GS2 could successfully make use of threads. Given a more complete realisation of the hybrid programming model, throughout the code I think it would give a marked improvement to the overall performance of GS2.

7.1 Future Work

There are several aspects of the project that I would like to see revisited in future work. The most prominent of which is the introduction of OpenMP threads to the entire GS2 code. However, given the complex modular structure of the program, this would be very time consuming to implement, and for an unknown level of pay off. It would most definitely be subject to a cost-benefit analysis, from which it may not be deemed worth pursue.

As GS2 has built in support for the SHMEM single-sided communications library, I would be very interested in exploring the performance ramifications of adding OpenMP to a SHMEM implementation.

Along with a more extensive implementation of OpenMP, I would also like to see further testing take place, with a wide variety of test cases. A diverse group of test cases could uncover aspects of performance not seen by the example test case used during the project. Alongside this form of testing, I would also like to see a comparison of the hybrid implementation of GS2 with a similar code such as GKW, which reports to scale 2-4x further using the hybrid model.

Appendix A

Input Datafile

```
&theta_grid_knobs
 equilibrium_option='eik'
/


&theta_grid_parameters
 rhoc = 0.4
 ntheta = 30
 nperiod= 1
/


&parameters
 beta = 0.04948
 zeff =   1.0
 TiTe = 1.0
/


&collisions_knobs
 collision_model = 'default'
 !collision_model = 'none'
 !collision_model='lorentz'
/


&theta_grid_eik_knobs
 itor = 1
 iflux = 1
 irho = 3
 ppl_eq = .false.
 gen_eq =  .false.
 efit_eq = .true.
 gs2d_eq = .true.
 local_eq = .false.
 eqfile = 'equilibrium.dat'
 equal_arc = .false.
 bishop = 1
 s_hat_input = 0.29
```

```
  beta_prime_input = -0.5
 delrho = 1.e-3
 isym = 0
 writelots = .false.
/


&fields_knobs
 field_option='implicit'
/


&gs2_diagnostics_knobs
 write_ascii = .false.
 print_flux_line = .true.
 write_flux_line = .true.
 write_nl_flux = .true.
 write_omega = .false.
 write_omavg = .false.
 write_final_moments = .false.
 write_final_fields=.false.
 print_line=.false.
 write_line=.false.


 save_for_restart=.false.
 nsave=         1000


 nwrite=        100
 navg=          200


 omegatol=  1.0e-5
 omegatinst = 500.0
/


&le_grids_knobs
 ngauss = 8
 negrid = 8
/


&dist_fn_knobs
 boundary_option= "linked"
 gridfac=   1.0
/
```

```
&init_g_knobs
 !restart_file= "nc/input.nc"
 ginit_option= "noise"
 phiinit=   1.e-6
 chop_side = .false.
/


&kt_grids_knobs
 grid_option='box'
/


&kt_grids_box_parameters
 y0 = 10
 ny = 96
 nx = 96
 jtwist = 2
/


&knobs
 fphi= 1.0
 fapar= 0.0
 faperp= 0.0
 delt= 1.0e-4
 nstep= 1000
 wstar_units = .false.
/


&species_knobs
 nspec=  2
/


&species_parameters_1
 type  = 'ion'
 z     = 1.0
 mass  = 1.0
 dens  = 1.0
 temp  = 1.0
 tprim = 2.04
 fprim = 0.0
 vnewk = 1.0
 uprim = 0.0
/
```

```
&dist_fn_species_knobs_1
 fexpr  = 0.45
 bakdif = 0.05
/


&species_parameters_2
 type  = 'electron'
 z     = -1.0
 mass  = 0.01
 dens  = 1.0
 temp  = 1.0
 tprim = 2.04
 fprim = 0.0
 vnewk = 1.0
 uprim = 0.0
/


&dist_fn_species_knobs_2
 fexpr= 0.45
 bakdif=  0.05
/

&theta_grid_file_knobs
 gridout_file='grid.out'
/


&theta_grid_gridgen_knobs
 npadd = 0
 alknob = 0.0
 epsknob = 1.e-5
 extrknob = 0.0
 tension = 1.0
 thetamax = 0.0
 deltaw = 0.0
 widthw = 1.0
/


&source_knobs
/


&nonlinear_terms_knobs
nonlinear_mode='on'
cfl = 0.5
```

```
/

&additional_linear_terms_knobs
/

&reinit_knobs
 delt_adj = 2.0
 delt_minimum = 1.e-8
/

&theta_grid_salpha_knobs
/

&hyper_knobs
/

&layouts_knobs
 layout = 'yxles'
 local_field_solve = .false.
 unbalanced_xxf = .true.
 max_unbalanced_xxf = 0.5
 unbalanced_yxf = .true.
 max_unbalanced_yxf = 0.5
 opt_local_copy = .true.
 opt_redist_init = .true.
 opt_redist_nbk = .true.
 velint_subcom = .true.
/
```

Appendix B
Example Submission Scripts

B.1 MPI-Only Submission Script

```
#!/bin/bash --login
#PBS -N gs2
#PBS -l mppwidth=512
#PBS -l mppnppn=32
#PBS -l walltime=00:30:00
#PBS -A d45

cd $PBS_O_WORKDIR

cd /work/d45/d45/s1268782/diss/testcases/
cp /home/d45/d45/s1268782/diss/ccfe_opt_test/gs2 .

export NPES=`qstat -f $PBS_JOBID | awk '/mppwidth/ {print
$3}'`
export NPERNODE=`qstat -f $PBS_JOBID | awk '/mppnppn/
{print $3}'`

aprun -n $NPES -N $NPERNODE ./gs2 gs.in
```

B.2 Hybrid Submission Script

```
#!/bin/bash
#PBS -N hybrid_1024
#PBS -l mppwidth=1024
#PBS -l mppnppn=32
#PBS -l walltime=03:00:00
#PBS -A d45

cd $PBS_O_WORKDIR

cd /work/d45/d45/s1268782/diss/hybrid_1024/
cp /home/d45/d45/s1268782/diss/ccfe_opt_test/gs2 .

export NPES=`qstat -f $PBS_JOBID | awk '/mppwidth/ {print
$3}'`
```

```
export NPERNODE=`qstat -f $PBS_JOBID | awk '/mppnppn/
{print $3}'`
export PSC_OMP_AFFINITY=FALSE
export OMP_NUM_THREADS=2

echo "OMP_NUM_THREADS" $OMP_NUM_THREADS

aprun -n 512 -N 16 -d $OMP_NUM_THREADS -S 4 ./gs2 gs.in
```

References

[1] A. Jackson, S. Booth, M. Bull, A. Gray, "HPC Architectures" course slides, EPCC, The University of Edinburgh, 2012.

[2] M. Bull, A. Krause, F. Reid, "Threaded Programming" course slides, EPCC, The University of Edinburgh, 2012.

[3] "OpenMP Website". [Online]. Available: http://openmp.org/ . [Accessed: Aug-2013].

[4] D. Henty, "Message-Passing Programming" course slides, EPCC, The University of Edinburgh 2012.

[5] "Message Passing Interface Forum". [Online]. Available: http://www.mpi-forum.org/ . [Accessed: Aug-2013].

[6] M. Bull, D. Henty, "Hybrid Programming", Advanced Parallel Programming course, EPCC, The University of Edinburgh, 2012.

[7] "Gyrokinetic Simulations Wiki". [Online]. Available: http://sourceforge.net/apps/mediawiki/gyrokinetics/index.php?title=Gyrokinetics_Wiki_Home . [Accessed: Aug-2013].

[8] "The GS2 Algorithm". [Online]. Available: http://sourceforge.net/apps/mediawiki/gyrokinetics/index.php?title=The_GS2_Algorithm . [Accessed: Aug-2013].

[9] "An Introduction to GS2". [Online]. Available: http://sourceforge.net/apps/mediawiki/gyrokinetics/index.php?title=An_Introduction_to_GS2 . [Accessed: Aug-2013].

[10] "GS2 Input Parameters". [Online]. Available: http://sourceforge.net/apps/mediawiki/gyrokinetics/index.php?title=GS2_Input_Parameters . [Accessed: Aug-2013].

[11] A. Jackson, "Improved Data Distribution Routines for Gyrokinetic Plasma Simulations". [Online]. Available: http://www.hector.ac.uk/cse/distributedcse/reports/GS202/GS202.pdf . [Accessed: Aug-2013].

[12] "HECToR Hardware". [Online]. Available: http://www.hector.ac.uk/service/hardware/ . [Accessed: Aug-2013]

[13] A. Jackson, M. Serigo Campobasso, "Optimised Hybrid Parallelisation of a CFD

code on Many Core Architectures," *CoRR*, vol. 1304.7654, 2013.

[14] "JUQUEEN – Configuration". [Online]. Available: http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/Configuration/Configuration_node.html . [Accessed: Aug-2013].