Mixed Mode Programming in a Star Formation Code

Nikolaos Tryfonidis

August 22, 2012
Abstract

Hybrid architectures have become the standard in HPC systems during the last decade; modern supercomputers contain nodes that consist of multiple cores with shared memory. Mixed-mode programming is a programming model that is perfectly suited for this hybrid architecture in theory, but a definite answer cannot be provided for whether or not it leads to better performance over MPI-only implementations.

MG is a Computational Fluid Dynamics code that has been parallelized with MPI. In the present project, the addition of OpenMP threads to MG is investigated, in order to transform it to a mixed-mode implementation and compare its performance to the original MPI-only version. Moreover, since the code uses linked-list traversals, it provides a good opportunity to test the performance of the relatively new OpenMP task construct, which was the biggest addition of OpenMP 3.0.

It was found that the manual parallelization of the linked-list traversal led to significant performance benefits for large numbers of PEs, whereas the OpenMP tasks implementation resulted in bad performance. Furthermore, the mixed-mode code showed significantly less memory requirements.
# Contents

1 Introduction 1

2 Architectures 3
   2.1 Distributed Memory ............................................. 3
   2.2 Shared Memory .................................................. 4
   2.3 Clustered Architecture ......................................... 6

3 Programming Models 7
   3.1 Distributed Memory - MPI ....................................... 7
      3.1.1 Communication ............................................. 8
      3.1.2 MPI Synchronization ....................................... 9
      3.1.3 MPI Example ............................................... 10
   3.2 Shared Memory - OpenMP ....................................... 11
      3.2.1 Work Distribution ........................................ 11
      3.2.2 Communication ............................................ 13
      3.2.3 Synchronization ........................................... 13
      3.2.4 OpenMP example .......................................... 14
      3.2.5 OpenMP Tasks ............................................. 16
   3.3 Mixed-Mode MPI+OpenMP Programming .......................... 18
      3.3.1 The Mixed-Mode Programming Model ...................... 19
      3.3.2 Mixed Mode Example ..................................... 20
      3.3.3 Advantages and Disadvantages ............................ 21

4 MG: The CFD Code Investigated 23
   4.1 MG ............................................................... 23
   4.2 PARC ............................................................ 25
   4.3 The “step” Function ............................................ 26
   4.4 Initial Benchmarks and Profiling .................................. 28
      4.4.1 Benchmarks ................................................. 28
      4.4.2 Profiling .................................................. 32

5 Implementation 34
   5.1 Computer Systems Used ........................................ 35
      5.1.1 Ness ....................................................... 35
      5.1.2 HECToR .................................................. 35
# Table of Contents

5.2 Introducing Threads .................................................. 36
   5.2.1 Replacement of "while" Loops ................................. 36
   5.2.2 Parallelizing the "for" Loops With OpenMP ................. 39
   5.2.3 Find Maximum: Scaling Investigation ....................... 40
   5.2.4 Loop Scaling With OpenMP .................................... 43
5.3 OpenMP Tasks ......................................................... 46
   5.3.1 Loop scaling with OpenMP tasks .............................. 47
5.4 Correctness Testing .................................................. 49
   5.4.1 Testing the OpenMP Tasks Version ......................... 53

6 Results ......................................................................... 54
   6.1 Mixed-Mode Benchmarks ............................................ 55
      6.1.1 2 Threads per MPI Process ................................. 56
      6.1.2 4 Threads per MPI Process ................................. 59
      6.1.3 8 Threads per MPI Process ................................. 62
      6.1.4 Comparison .................................................... 65
   6.2 Memory Usage ......................................................... 67

7 Conclusion ................................................................. 69
   7.1 Problems Encountered .............................................. 70
   7.2 Future Work .......................................................... 71

A Benchmark Data ............................................................ 72
   A.1 Initial MG Benchmarks ............................................. 72
   A.2 Find Maximum: Scaling Investigation ....................... 73
   A.3 Loop Scaling With OpenMP ..................................... 74
   A.4 Loop Scaling With OpenMP tasks ............................. 75
   A.5 Mixed-Mode Benchmarks ......................................... 75
      A.5.1 2 Threads Per MPI Process ............................... 75
      A.5.2 4 Threads Per MPI Process ............................... 76
      A.5.3 8 Threads Per MPI Process ............................... 77
# List of Figures

2.1 Illustration of a distributed memory system of 10 processors ............... 3  
2.2 Illustration of an SMP shared memory system, with four cores accessing their shared memory via a single bus ............................. 4  
2.3 Illustration of a NUMA shared memory system, with four cores. Cores have fast access to their local memory and slower access to remote memory. ......................................................... 5  
2.4 Illustration of a clustered architecture, with 2 nodes of 4 cores each. ....... 6  
3.1 Point to point communication (send and matching receive) between two processes ................................................................. 8  
3.2 Broadcast collective communication; a process sends a message to all other processes in the communicator ........................................ 8  
3.3 "All reduce" collective communication; a variable is summed and the result is sent to all processes ................................. 9  
3.4 Distribution of a large dataset to a number of processes ................. 10  
3.5 OpenMP threads executing a parallel region. The master thread creates the other threads at the start of the parallel region, and terminates them at the end. ....................................................... 11  
3.6 Four OpenMP threads operating on a shared memory array, using a static schedule without a specified chunk size. ....................... 15  
3.7 Four OpenMP threads operating on a shared memory array, using a static schedule with a chunk size of one. ............................ 15  
3.8 Mixed-mode programming in a NUMA region. One MPI process (dark blue) is placed inside each NUMA region. ......................... 19  
3.9 Mixed mode example. An arrays is distributed to two MPI processes which then spawn three threads each to operate on their smaller arrays in parallel. ..................................................... 21  
4.1 An example of AMR. The grid size is reduced in regions where the solution varies rapidly, giving better accuracy (reproduced by [13]). ........ 23  
4.2 PARC Broadcast function ......................................................... 25  
4.3 PARC "Synchronize Array" function ......................................... 26  
4.4 MG, execution time versus number of MPI processes (size: \(100^3\)) .......... 29  
4.5 MG, execution time versus number of MPI processes (size: \(200^3\)) ........ 30  
4.6 MG, execution time versus number of MPI processes (size: \(300^3\)) ......... 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.7</td>
<td>MG, speedup versus number of MPI processes (all problem sizes)</td>
<td>31</td>
</tr>
<tr>
<td>4.8</td>
<td>MG, efficiency versus number of MPI processes (all problem sizes)</td>
<td>31</td>
</tr>
<tr>
<td>4.9</td>
<td>MG, percentage of time spent in MPI functions versus number of MPI processes (size: $200^3$ cells)</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>&quot;Step&quot; function: The main parts of the first order (a) and second order (b) steps.</td>
<td>36</td>
</tr>
<tr>
<td>5.2</td>
<td>Original linked-list traversal (a) and replacement of &quot;while&quot; loop with a &quot;for&quot; loop (b)</td>
<td>37</td>
</tr>
<tr>
<td>5.3</td>
<td>Finding the maximum rates in parallel: Execution time versus number of threads, using three different methods</td>
<td>43</td>
</tr>
<tr>
<td>5.4</td>
<td>Finding the maximum rates in parallel: Speedup versus number of threads, using three different methods</td>
<td>43</td>
</tr>
<tr>
<td>5.5</td>
<td>Loop execution times versus number of threads (size: $100^3$ cells)</td>
<td>44</td>
</tr>
<tr>
<td>5.6</td>
<td>Loop speedup versus number of threads (size: $100^3$ cells)</td>
<td>44</td>
</tr>
<tr>
<td>5.7</td>
<td>Loop execution times versus number of threads (size: $200^3$ cells)</td>
<td>45</td>
</tr>
<tr>
<td>5.8</td>
<td>Loop speedup versus number of threads (size: $200^3$ cells)</td>
<td>45</td>
</tr>
<tr>
<td>5.9</td>
<td>Tasks version: loop execution times versus number of threads (size: $100^3$ cells)</td>
<td>47</td>
</tr>
<tr>
<td>5.10</td>
<td>Tasks version: loop speedup versus number of threads (size: $100^3$ cells)</td>
<td>48</td>
</tr>
<tr>
<td>5.11</td>
<td>Tasks version: loop execution times versus number of threads (size: $200^3$ cells)</td>
<td>48</td>
</tr>
<tr>
<td>5.12</td>
<td>Tasks version: loop speedup versus number of threads (size: $200^3$ cells)</td>
<td>48</td>
</tr>
<tr>
<td>5.13</td>
<td>Evolution of density along the x axis, for 2x10 computational steps (size: $20^3$ cells), using the original MG code.</td>
<td>50</td>
</tr>
<tr>
<td>5.14</td>
<td>Evolution of density along the x axis, for 2x10 computational steps (size: $20^3$ cells), after parallelizing the linked-list loops. The test run was made using 4 MPI processes with 4 threads each.</td>
<td>51</td>
</tr>
<tr>
<td>5.15</td>
<td>Density along the x axis for 10 computational steps, using the original MG code (size: $20^3$ cells)</td>
<td>52</td>
</tr>
<tr>
<td>5.16</td>
<td>Density along the x axis for 10 computational steps, using the original MG code and the threaded version on the same window - plots coincide (size: $20^3$ cells)</td>
<td>52</td>
</tr>
<tr>
<td>6.1</td>
<td>MG benchmark: mixed-mode (2 threads/MPI process) and MPI-only (size: $100^3$ cells)</td>
<td>56</td>
</tr>
<tr>
<td>6.2</td>
<td>MG benchmark: mixed-mode (2 threads/MPI process) and MPI-only (size: $200^3$ cells)</td>
<td>57</td>
</tr>
<tr>
<td>6.3</td>
<td>MG benchmark: mixed-mode (2 threads/MPI process) and MPI-only (size: $300^3$ cells)</td>
<td>58</td>
</tr>
<tr>
<td>6.4</td>
<td>MG benchmark: mixed-mode (4 threads/MPI process) and MPI-only (size: $100^3$ cells)</td>
<td>59</td>
</tr>
<tr>
<td>6.5</td>
<td>MG benchmark: mixed-mode (4 threads/MPI process) and MPI-only (size: $200^3$ cells)</td>
<td>60</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.6</td>
<td>MG benchmark: mixed-mode (4 threads/MPI process) and MPI-only (size: $300^3$ cells)</td>
<td>61</td>
</tr>
<tr>
<td>6.7</td>
<td>MG benchmark: mixed-mode (8 threads/MPI process) and MPI-only (size: $100^3$ cells)</td>
<td>62</td>
</tr>
<tr>
<td>6.8</td>
<td>MG benchmark: mixed-mode (8 threads/MPI process) and MPI-only (size: $200^3$ cells)</td>
<td>63</td>
</tr>
<tr>
<td>6.9</td>
<td>MG benchmark: mixed-mode (8 threads/MPI process) and MPI-only (size: $300^3$ cells)</td>
<td>64</td>
</tr>
<tr>
<td>6.10</td>
<td>Speedup vs PEs, for all combinations of threads per MPI process (size: $100^3$)</td>
<td>65</td>
</tr>
<tr>
<td>6.11</td>
<td>Speedup vs PEs, for all combinations of threads per MPI process (size: $200^3$)</td>
<td>66</td>
</tr>
<tr>
<td>6.12</td>
<td>Speedup vs PEs, for all combinations of threads per MPI process (size: $300^3$)</td>
<td>66</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Profiling with small numbers of MPI Processes ......................... 32
4.2 Profiling with large numbers of MPI Processes ......................... 32

6.1 MG memory usage for MPI-only and mixed-mode (size: 100³) ........ 68
6.2 MG memory usage for MPI-only and mixed-mode (size: 200³) ........ 68
6.3 MG memory usage for MPI-only and mixed-mode (size: 300³) ........ 68

A.1 MG, initial benchmark (size: 100³) .................................. 72
A.2 MG, initial benchmark (size: 200³) .................................. 73
A.3 MG, initial benchmark (size: 300³) .................................. 73
A.4 Find maximum: execution times with single command inside critical region .................................................. 73
A.5 Find maximum: execution times with loop inside critical region .... 73
A.6 Find maximum: execution times with temporary array, no critical region .................................................. 74

A.7 Loop execution times with manual OpenMP implementation (size: 100³ cells) .................................................. 74
A.8 Loop execution times with manual OpenMP implementation (size: 200³ cells) .................................................. 74
A.9 Loop execution times with the OpenMP tasks implementation (size: 100³ cells) .................................................. 75
A.10 Loop execution times with the OpenMP tasks implementation (size: 200³ cells) .................................................. 75

A.11 MG Mixed-mode benchmarks: Execution times with 2 threads per MPI process (size: 100³) .............................. 75
A.12 MG Mixed-mode benchmarks: Execution times with 2 threads per MPI process (size: 200³) .............................. 76
A.13 MG Mixed-mode benchmarks: Execution times with 2 threads per MPI process (size: 300³) .............................. 76
A.14 MG Mixed-mode benchmarks: Execution times with 4 threads per MPI process (size: 100³) .............................. 76
A.15 MG Mixed-mode benchmarks: Execution times with 4 threads per MPI process (size: 200³) .............................. 76
A.16 MG Mixed-mode benchmarks: Execution times with 4 threads per MPI process (size: 300³) .............................. 77
A.17 MG Mixed-mode benchmarks: Execution times with 8 threads per MPI process (size: $100^3$) ........................................ 77
A.18 MG Mixed-mode benchmarks: Execution times with 8 threads per MPI process (size: $200^3$) ........................................ 77
A.19 MG Mixed-mode benchmarks: Execution times with 8 threads per MPI process (size: $300^3$) ........................................ 77
Acknowledgements

I would like to thank my main supervisor, Dr. Chris Maynard, for his continuous help and guidance during the project and for keeping my stress levels to a minimum. I wish him the best in his new quest to figure out the British weather at the UK Met Office.

I would also like to thank my second supervisor, Dr. Mark Bull, for taking over the supervision for the final part of the project, and for providing useful advise and help.

I am also very thankful to Professor Sam Falle, from the Department of Applied Mathematics at the University of Leeds, for providing the MG code and for his valuable help throughout the project.
Chapter 1

Introduction

Shared memory architectures, multicore CPUs with a single-address shared memory space, have become the standard in computer systems during the last decade. This has been the result of advances in microprocessor technology, allowing for more transistors on a CPU chip and, on the other hand, power limitations at higher clock rates. Consequently, trends have moved from increasing the clock rates of a CPU to increasing the number of cores on a CPU, while keeping their clock rates relatively low.

Nowadays, it is rare to find a computer system with a single-core CPU. Most personal computers have multicore CPUs and supercomputers consist of nodes of multiple cores. HECToR, the UK national supercomputer, which is a Cray XE6 system, consists of 90,112 nodes, each of which has two 16-core processors, resulting in a total of 32 cores per node.

This wide adoption of this hybrid architecture leads to the question of what kind of programming model can be used in order to exploit its characteristics. MPI is certainly portable to hybrid systems and can be used to communicate between nodes. However, whether or not MPI is the ideal choice for intra-node communication is unclear, as it does not take advantage of shared-memory. OpenMP, a programming model designed for shared-memory programming, is a good candidate for computations inside a node. Using MPI to communicate between nodes and OpenMP to parallelize code inside the nodes fits this hybrid architecture very well as a concept.

An issue that may arise in this context is transforming existing codes that only use MPI, to mixed-mode MPI+OpenMP codes. In the present project, the introduction of OpenMP threads to an existing scientific code (MG) has been investigated. The code, provided by Professor Sam Falle (the author of the code) from the Department of Applied Mathematics at the University of Leeds, is a Computational Fluid Dynamics code which has been parallelized using MPI. MG (and the assorted libraries) was developed by the Mantis Numerics company, of which Professor Sam Falle is the director.
According to information provided by Professor Sam Falle, MG has been used extensively in Astrophysics, but is also being used to model high pressure releases from pipelines carrying liquid CO2, non-ideal detonations, chemo-taxis and groundwater flow.

After introducing threads into the code, benchmark tests were performed in order to find out whether mixed-mode programming can offer better performance for this particular program. Moreover, the nature of the code offered the opportunity to use a relatively new feature of OpenMP the task construct. Since the loops that were parallelized with OpenMP are linked-list traversals, they are very well suited for OpenMP tasks. This provides a good opportunity to compare their performance to a manual parallelization of the linked-list loops.

Relevant background information about this project is contained in Chapters 2 and 3. HPC architectures are discussed in Chapter 2, and the different programming models are described in Chapter 3. Chapter 4 introduces MG and PARC, describing their use and showing the initial benchmarks and profiling of the program. Chapter 5 describes the implementation work done, before showing the performance results of the mixed-mode implementation in Chapter 6.
Chapter 2

Architectures

Mixed mode programming is closely tied to the heterogeneous nature of clustered architectures. A review of the different HPC architectures is thus desirable as an introduction to this hybrid programming model. This chapter will provide a brief introduction to the characteristics of these architectures, mainly with regard to the relationship between processors and memory. Information in this section is based on [1] and [2].

2.1 Distributed Memory

Distributed memory systems consist of a number of processors, each with its own local memory, connected with a form of interconnect. Each processor runs its own separate operating system and operates on its local memory, independently from other processors. Access to remote memory is accomplished by communication via the interconnect, using explicit message-passing.

Figure 2.1: Illustration of a distributed memory system of 10 processors
This simple architecture has many advantages. First of all, a distributed memory system can be built from widely available commodity components, making it cost-effective and simple to assemble. Secondly, it is a highly scalable architecture, since memory (and bandwidth) can be easily increased by adding more processors to the system. Finally, since each processor operates on its memory individually, cache utilization takes place effectively, without the complications of cache coherency mechanisms in shared memory systems.

However, the explicit communication needed for this architecture requires careful planning by the programmer and can thus be prone to errors and overheads, affecting performance. Moreover, the high scalability of distributed memory systems relies heavily on good interconnect mechanisms. Finally, since jobs for execution are placed in certain processors and remain there until the end of the application, load imbalance can become an issue.

### 2.2 Shared Memory

Shared memory systems, as the name suggests, consist of a number of processors and memory that is shared among them. This memory is accessible by all processors and it is viewed as a global address space. Changes to shared memory by one processor are thus visible by all other processors in the shared memory system. Communication is, in this way, implicit and it is accomplished by reads and writes to shared memory. Processors maintain their caches updated by automatic cache coherency mechanisms, which are hardware-implemented.

Shared memory systems can be divided between Symmetric Multiprocessing systems (SMPs) and Non-Uniform Memory Access (NUMA) systems. Symmetric multiprocessing systems provide equal access to memory; all processors have the same latency and bandwidth to access any shared memory location.

Figure 2.2: Illustration of an SMP shared memory system, with four cores accessing their shared memory via a single bus
Non-uniform access shared memory systems also provide a single address memory space. However, in NUMA systems access to that shared memory is not equal; a group of processors has fast access to its *local* part of the shared memory, and slower access to remote parts of shared memory. This can remove the stress on the bus that may result by all cores waiting to access some part of the shared memory.

![Illustration of a NUMA shared memory system, with four cores. Cores have fast access to their local memory and slower access to remote memory.](image)

As far as the user is concerned, a shared memory system can be viewed as a single processor. Apart from the single address space and the implicit communication, a single operating system is responsible for the function of all cores and for moving tasks among them. This results in simplicity of development and maintenance. Moreover, data sharing can be fast, due to memory being physically close to all processors (in contrast to distributed memory systems).

However, communication can be a bottleneck in shared memory systems, as accesses to shared memory can cause too much traffic on the bus. While NUMA architectures can minimize this effect for systems of hundredths of processors, it remains a limiting factor for scalability in shared memory systems. Apart from that, the implicit nature of communication means that the programmer is responsible for ensuring the correctness of data sharing (thread safety).
2.3 Clustered Architecture

A combination of distributed systems and shared memory systems results in a clustered architecture. A clustered system is made up as a distributed system whose nodes consist of shared memory systems, as shown in Figure 2.4.

![Clustered Architecture Diagram](image)

Figure 2.4: Illustration of a clustered architecture, with 2 nodes of 4 cores each.

A clustered system combines the features of its parent architectures. Each node has its own memory address space and communicates with other nodes using explicit message passing. Cores that belong to the same node share their memory as a single address space.

Clustered systems are thus easily expandable; simply adding more nodes increases the total memory and number of cores of a cluster. However, the higher complexity of this architecture means that it can be difficult to take advantage of its heterogeneous nature in order to analyze and achieve performance.
Chapter 3

Programming Models

In this chapter, a brief review of the two programming models that make up mixed mode programming will be given. Afterwards, a more detailed description of mixed mode programming will take place, to provide some insight into the motivation behind this parallel programming paradigm.

For the scope of the present dissertation, the focus will be on the models that will be used; MPI and OpenMP for distributed and shared memory respectively. However, this is not restrictive, as these two implementations are the most widely used for their respective models.

3.1 Distributed Memory - MPI

Message passing is a model that fits the distributed memory parallel architecture. Since each processor is only aware of its own memory, sending messages is a natural way to communicate between processors. Parallelization is achieved by having processors operate on different parts of the program’s dataset concurrently. In this way, the workload is distributed among processes and the program (hopefully) finishes faster.

In message passing, the same program is executed by a number of processors. Each such instance of the program is referred to as a process. Processes are grouped together into communicators. It should be noted that while all processes execute the same code, execution paths of different processes may vary according to the role that the programmer assigns to them.

The Message Passing Interface (MPI) is a message-passing library interface specification, which is used to write message passing implementations. In the rest of this section, a brief discussion of the basic aspects of MPI will take place. For more detailed descriptions of MPI, the reader is referred to [3] and [4].
3.1.1 Communication

Communication in MPI can generally be divided into two categories: *point-to-point* and *collective* communication functions. As the names suggest, point-to-point communications involve messages being sent and received between two processes. Collective communications, on the other hand, involve *all* processes present in the specified communicator.

Point-to-point communications are basically "send" and "receive" operations between *two* processes. A simple description of such a procedure would be a process 1 sending a message to process 2. In order for this communication to complete, process 1 must call the "send" function and process 2 must call the "receive" function.

It should be noted that both processes participating in a point-to-point message exchange must wait until the message has been successfully received before continuing. An exception is the *non-blocking* version of point-to-point communication functions, where processes must wait at a specified waiting point before proceeding. This behaviour is very important for the appropriate synchronization of MPI processes.

![Figure 3.1: Point to point communication (send and matching receive) between two processes](image1)

Collective communications involve *all* processes in a specified communicator. There is a variety of collective communication functions, for different purposes. An example can be the *broadcast* function, which sends data from one process to all other processes, as shown in Figure 3.2.

![Figure 3.2: Broadcast collective communication; a process sends a message to all other processes in the communicator](image2)
Another important collective communication function is the "All Reduce" function, which applies a reduction operation (i.e., a sum) for a value across all processes in a communicator and sends the result to all processes, as shown in the next figure.

Figure 3.3: "All reduce" collective communication; a variable is summed and the result is sent to all processes.

Collective communication functions in MPI are implicit synchronization points; all processes must wait for the collective function to complete before proceeding.

### 3.1.2 MPI Synchronization

An important factor in MPI is synchronization. Synchronization in MPI programs is usually implicit, and is accomplished by arranging computation with regard to the completion of the appropriate communication functions. In this way, the programmer can be sure that some data in a process is valid at a certain point through execution.

The implicit nature of communication in MPI is a relatively simple concept. As discussed in Section 3.1.1., point-to-point communication consists of two processes sending and receiving a message. Generally, this means that both participating processes must wait until the communication is complete before continuing to the execution of further code.

In the case of non-blocking communication functions, the programmer is responsible to correctly indicate the waiting point of the communication; processes must then wait at the specified point until the appropriate communications have completed.

While explicit synchronization (barriers) is available in MPI, it is usually used for secondary purposes, such as timing certain parts of the program. This is necessary in order to correctly time parts of an MPI program, because otherwise the process responsible for timing may be timing its own execution time instead of the time it took for all processes to finish a certain code region.
3.1.3 MPI Example

A simple example will be useful to showcase the general idea of parallel programming in MPI. Consider a serial program that reads input from a file into a single array, operates on all elements of that array and finally writes the updated array into an output file.

A parallel version of this program written for the message passing model may involve one process (master) reading the input data into a single array. Afterwards, this array would be divided equally into smaller arrays, each of which would be sent as a message to another process respectively, as shown in the next figure. Each process would then operate on its small dataset and send the resulting array back to the master process. The master process would then reassemble the array and write the output file.

![Figure 3.4: Distribution of a large dataset to a number of processes](image)

In the example above, there are two distinct roles a process can play; a master process and a worker process. The master process, apart from performing computational work, is also responsible for the input/output phases of the program. Other processes receive their respective datasets, operate on them and finally send their new data back to the master process.

This example is simple, but it is sufficient to show a basic structure of a parallel program implemented with message passing. For instance, parallel input and output can be dealt with in a more elegant and efficient way, using MPI-IO, in which processes would be able to read and write to files concurrently. However, a discussion of MPI-IO is outside the scope of this project.
3.2 Shared Memory - OpenMP

OpenMP is an Application Program Interface (API), which is defined by a group of software and hardware vendors [5]. It provides a portable and scalable way for the development of parallel applications that take advantage of shared memory systems. The OpenMP Architecture Review Board (ARB) owns the OpenMP brand and is responsible for overseeing and approving new versions of the OpenMP specification. In this section, a brief review of OpenMP will be given. A detailed description of OpenMP can be provided in [6] and [7].

The fundamental element of parallelism in OpenMP is the concept of threads. Threads can be thought as processing elements that can execute parts of code in parallel; in a sense, they are the shared-memory equivalents of MPI processes. OpenMP uses the model of fork-join to implement parallelism. Programs using OpenMP start as serial code, with only one thread (the master thread) executing the code. At some point in the program, parallelism is introduced with the use of compiler directives. The part of the code that is indicated as parallel is called a parallel region. At the point indicated by the directive, the master thread spawns a number of threads that execute the denoted part of the code in parallel, as illustrated in Figure 3.5.

![Figure 3.5: OpenMP threads executing a parallel region. The master thread creates the other threads at the start of the parallel region, and terminates them at the end.](image)

Additional clauses after the directive can specify other attributes, such as the data scope of variables inside the parallel region. The most basic differentiation is between shared and private variables. Shared variables, as the name suggests, are accessible by all threads. On the other hand, a private variable is copied to every thread’s private memory, where it can only be accessed by the respective thread.

3.2.1 Work Distribution

Inside the parallel region, each thread can be identified by its individual thread ID, which is a non-negative integer. Work can be assigned to each thread manually in this
way. For example, consider a program where four threads operate on an array of four elements. By storing each thread’s ID to a private integer variable named \texttt{threadID}, and declaring the array as shared, each thread could operate to the respective element of the array with the following code:

```c
/* OpenMP directive */
#pragma omp parallel private(threadID) shared(array)
[
  threadID = omp_get_thread_num(); /* store ID in private variable */
  array[threadID] = threadID; /* operate on shared array */
]
```

Listing 3.1: Threads writing their IDs to the corresponding elements of a shared array

Apart from manually distributing work, directives that invoke work-sharing methods are available. These work-sharing directives distribute the iterations of a loop to different threads automatically:

```c
#pragma omp parallel private(i) shared(array)
[
  #pragma omp for schedule(type[,chunk])
  for (i=0; i<N; i++) {
    array[i] = i;
  }
]
```

Listing 3.2: A simple work-sharing schedule

Here, the iterations of the loop are assigned automatically to different threads, according to the schedule chosen inside the parentheses. Available schedules are:

1. \textbf{Static}: Chunks of size \texttt{chunk} are assigned statically to threads.
2. \textbf{Dynamic}: Chunks of size \texttt{chunk} are assigned dynamically to threads.
3. \textbf{Guided}: Chunks of decreasing size are assigned dynamically to threads.
4. \textbf{Runtime}: The schedule is chosen by setting an environment variable before execution.
5. \textbf{Auto}: The schedule is chosen by the compiler or system.

As a general rule, static schedules are more suitable for problems with a balanced workload. Work is distributed evenly among threads, saving the large overhead of dynamic distribution during execution. For imbalanced workloads, however, dynamic schedules can be more efficient. Despite their higher overhead, distributing work dynamically can save threads from being idle.
3.2.2 Communication

Communication in OpenMP is implicit. Since memory is accessible by all threads, there is no need for explicit message passing. Communication is instead achieved by accessing *shared* memory elements. Consider again the previous simple example in Listing 3.1. There, each thread writes its own ID to the corresponding element of an array; we can say that each thread has an *affinity* with its corresponding element of the array. The array is visible after the parallel region by the master thread, which is able to process its elements in any desirable way. This is a form of communication, since memory elements that were processed by separate threads are ready to be accessed by the master thread. The message passing equivalent would involve a procedure similar to that described in Section 3.1.2.

Apart from that, a different form of implicit communication in OpenMP involves threads reading from or writing to elements of shared memory that other threads have affinity to. A variation on the previous code sample could be as follows:

```c
#pragma omp parallel private(threadID) shared(array1, array2)
{
    threadID = omp_get_thread_num();
    nThreads = omp_get_num_threads();
    if (threadID != 0 && threadID != nThreads - 1) {
        array1[threadID] = array2[threadID - 1] + array2[threadID + 1];
    }
}
```

Listing 3.3: Threads reading adjacent memory elements

Executing the code above with four threads, the two "middle" threads (1 and 2) will access elements of array2 that have affinity to other threads, in line 6. This type of communication is similar to halo regions being sent and received in message passing programs. The OpenMP equivalent may appear to be more simple; however, the programmer must be very careful to ensure that the shared data is accessed correctly.

3.2.3 Synchronization

At the end of a parallel region there is an implied barrier; all threads must reach that point before the master thread continues executing the code after the parallel region. This is an implicit type of synchronization of OpenMP. Explicit synchronization is also available in the form of *barriers* inside parallel regions, where all threads must arrive before proceeding further.
#pragma omp parallel private(threadID) shared(array) 
{
  threadID = omp_get_thread_num(); /* store ID in private variable */
  array[threadID] = threadID; /* operate on shared array */
  #pragma omp barrier /* Threads synchronize here */
  if (threadID == 0) {
    /* (...) Master thread prints all elements of array */
  }
} 

Listing 3.4: A barrier ensures that thread 0 will print the array elements after they have all been set.

In the previous code sample, all threads must reach the barrier before proceeding. In this way, a thread can safely access elements of the shared array, since all threads have finished writing to their respective array elements. If the barrier was not used, it could be possible that thread 0 would print an array element before its value had been set by the appropriate thread, leading to errors. Such cases are called race conditions and require careful planning by the programmer.

An important synchronization feature of OpenMP is the critical region, which can be defined inside a parallel region, again with the use of a directive. Critical regions signify parts of code that should be executed only by a thread at a time. In other words, when a thread is inside a critical region, another thread that reaches that point must wait for the first thread to complete the critical region before entering. Critical regions are very useful in OpenMP, in cases where shared variables have to be updated by threads concurrently. As writing to the same memory location with multiple threads at the same time produces indeterminate results, critical regions can ensure that accesses to a shared memory location is done safely.

### 3.2.4 OpenMP example

A simple example to illustrate a program parallelized with OpenMP could be similar to the one described for MPI in Section 3.1.2. An obvious difference with OpenMP is that there would be no need to actually partition the large array into smaller ones. Also, explicit messages would not be required.

Instead, the large array would be a shared variable, accessible by all threads. The main computational loop would then be declared as part of a parallel region. The iterations of the loop could be assigned to different threads by using a work-sharing directive and one of the schedules described in the previous section.
In Figure 3.6, without specifying a chunk size, equal contiguous chunks of iterations are distributed to threads. This is very similar to the MPI decomposition discussed in the MPI example. Alternatively, in Figure 3.7, we see that a chunk size of 1 results in iterations being distributed in a round-robin fashion to threads. In case there is any spatial imbalance among iterations (for example, the first half of the iterations being significantly more heavy), choosing a small chunk size can ensure that the heavy part of the workload is divided among all threads.

A choice of a dynamic schedule of a certain chunksize would result in similar types of...
distributions, with the basic difference that new groups of iterations would be assigned
dynamically to threads when they finish their previous work.

3.2.5 OpenMP Tasks

The task construct is a relatively new feature of OpenMP, being a major addition in
OpenMP 3.0 (2008) [8]. The task construct is important to the code of the present
project, as will become evident after the description of the code in the next chapter. It
is thus necessary to describe this new feature of OpenMP; this subsection will briefly
introduce the general concept of tasks in OpenMP, based on [6], [8] and [9].

A task is basically a piece of code along with its corresponding data. Inside a parallel
region, a thread encountering a task region will generate a task from the associated
code-block. Some thread, at a later point in the parallel region, will then execute the
task. In other words, tasks constitute "pieces" of work that are defined by an encoun-
tering thread and then completed by one of the threads in the parallel region. For more
details on the task construct, including clauses and restrictions, see [6].

```
#pragma omp parallel
{
  #pragma omp task
  {
    /* (Task Code) */
  }
}
```

Listing 3.5: A task inside a parallel region.

Tasks in OpenMP are useful because they enable parallelization of a more wide variety
of loops. As described in Section 3.2.1, loops need to have well-defined bounds in
order to be automatically parallelized by OpenMP, using the work-sharing constructs.
Even in cases where work is manually distributed to threads, the programmer needs to
know the first and last iterations of a loop. A "for" loop in C, with known bounds, is an
example of a good candidate for OpenMP parallelization. A "while" loop, however is
not; this can be a major restriction in codes that utilize such loops.

A good example where parallelizing with OpenMP may prove difficult is a linked-list
traversal. Linked list traversals generally involve a "while" loop that iterates through
items of a linked list until reaching the final item:
The previous example is not straightforward to parallelize in OpenMP. A manual solution would be to replace the "while" loop with a "for" loop. In order to do this, however, additional steps have to be taken before the loop. First, the number of iterations has to be counted and then an array of pointers (of size equal to the number of iterations) must be allocated. Then, each pointer must be pointed to the respective list item. Finally, a "for" loop can be executed, by pointing to the appropriate list item in each iteration, as shown below.

```
/* Point to first item of list */
p = firstItem;

/* Go through items until reaching the last one */
while (p != NULL) {
    calculate(p);
    p = p->nextItem;
}
```

Listing 3.6: Linked list traversal

The previous example is not straightforward to parallelize in OpenMP. A manual solution would be to replace the "while" loop with a "for" loop. In order to do this, however, additional steps have to be taken before the loop. First, the number of iterations has to be counted and then an array of pointers (of size equal to the number of iterations) must be allocated. Then, each pointer must be pointed to the respective list item. Finally, a "for" loop can be executed, by pointing to the appropriate list item in each iteration, as shown below.

```
/* Count */
while (p != NULL) {
    counter ++;
    p = p->nextItem;
}

/* Allocate array of pointers */
array_ptr = (struct typeP **) malloc(counter*sizeof(struct typeP *));

/* Point */
p = firstItem;
for (i = 0; i < counter; i++) {
    array_ptr = p;
    p = p->nextItem;
}

/* Original loop */
for (i = 0; i < counter; i++) {
    /* Set p to "i" element of pointer-array */
    p = array_ptr[i];
    calculate(p);
}
```

Listing 3.7: Linked list traversal with a "for" loop

The last loop of the code sample above is the original "while" loop, transformed to a "for" loop. It can now be parallelized, with every thread getting different values of the iteration variable and executing the code for the corresponding items of the list. It is
obvious, however, that this method involves a lot of additional code. Apart from extra work, the possibility of introducing bugs and the reduced readability of the code, this is also a non-elegant method.

On the contrary, a more elegant and straightforward implementation can be accomplished by using tasks. The basic idea is as follows; the "while" loop will be enclosed in a parallel region, with the code inside the loop being specified as a task. A single thread will then go through iterations of the loop, packaging them into different tasks. Other threads present in the parallel region (or the packaging thread itself) will then execute the tasks. The following code shows the described concept:

```c
#pragma omp parallel
{
  #pragma omp single
  {
    p = firstItem;
    while (p != NULL) {
      #pragma omp task
      {
        calculate(p);
      }
      p = nextItem;
    }
  }
}
```

Listing 3.8: Linked list traversal with tasks

It is obvious that using tasks in this case is not only easier and straightforward, but also results in more elegant code. The task construct is an addition that increased the flexibility of OpenMP, making it able to parallelize a wider variety of loops.

### 3.3 Mixed-Mode MPI+OpenMP Programming

The clustered architecture, as was described in the previous chapter, fits nicely with a similarly hybrid programming model. Since nodes have a single memory space, OpenMP can be used for computations inside a node and MPI can be utilized for communication between different nodes. This combination is commonly referred to as hybrid or mixed-mode programming.

The motivation behind this combination is simple. While MPI is perfectly suitable for shared memory systems, it can be argued that it does not take any advantage of the shared memory of the nodes; message-passing overheads may be particularly expensive, whereas simple shared memory accesses might be more appropriate. In this section, the mixed-mode programming model will be described, followed by a discussion on its possible advantages and disadvantages.
Information in this section is based on [10] and [11].

3.3.1 The Mixed-Mode Programming Model

The general characteristic of the mixed-mode programming model is the utilization of both MPI processes and OpenMP threads. This translates to a problem being partitioned into separate smaller problems and distributed to different MPI processes. Each MPI process then spawns its own team of threads and executes its share of the problem utilizing threaded parallelism.

Generally, MPI is used for communication between nodes, and OpenMP is used within each node’s shared memory. In such an implementation, nodes may communicate by having one of their cores (for example the master thread) calling MPI message-passing functions outside parallel regions. After the communication is complete, the master thread will spawn the available threads (the other cores of the node) in order to perform the appropriate calculations using shared-memory parallelism. In other words, a single MPI process may be placed inside each node and the remaining cores are threads which are spawn when necessary.

However, as memory in shared memory multiprocessors is not always symmetric, as described in chapter 2, NUMA multiprocessors may require a slightly different treatment in order to reach their full potential. The fact that memory is not symmetric may lead to excessive overheads when threads access remote memory areas. In that case, it may be more appropriate to place one MPI process per NUMA region, which will then spawn the remaining cores of its own NUMA region as threads. In this way, adjacent threads will reside in the same NUMA region, making memory accesses faster.

![Diagram of Mixed-mode programming in a NUMA region](image)

Figure 3.8: Mixed-mode programming in a NUMA region. One MPI process (dark blue) is placed inside each NUMA region.

Depending on the *where* and by *which thread* MPI calls take place in the program, there can be variations on the simple mixed-mode example described above. In summary, the possible alternative mixed-mode programming models are:
• **Master-Only:** In this model, calls to MPI functions take place only outside OpenMP parallel regions. This makes it the simplest model to develop and maintain. Work is distributed initially to different MPI processes, which then spawn threads to tackle the most intensive parts of the code (i.e. expensive loops). Halo regions and input-output communication takes place before or after the OpenMP parallel regions.

• **Funneled:** In the funneled model, calls to MPI functions can take place inside parallel regions. However, MPI functions are only called through the same thread, which is usually chosen to be the master thread.

• **Serialized:** In the serialized model, calls to MPI functions are made only by one thread at a time.

• **Multiple:** The multiple model is the most complex model; calls to MPI functions are made by all threads and can take place simultaneously.

Generally, the master-only model is the simplest model conceptually, since it is basically an MPI code that also uses OpenMP threads in certain parts of the code. It can be thought of as an MPI code with certain parts that are "accelerated" with further share memory parallelism. This also means that an existing MPI code may be easily expanded to a hybrid code without too much effort.

However, this simplicity also has its downsides since, as we read in [12], the fact that only one thread performs the calls to MPI functions, inter-node bandwidth is not taken advantage to its full potential. Moreover, while the master thread performs the MPI calls, other threads remain idle, basically wasting CPU cycles. Finally, with MPI communication being made outside parallel regions, it is not possible for the programmer to incorporate overlapping computation and communication techniques, in order to perform calculations on non-halo parts of the data while waiting for halo data to arrive.

The other mixed-mode programming models, that do involve MPI calls being made from parallel regions, can overcome the above limitations. However, implementing mixed-mode code following those models can be significantly more difficult.

### 3.3.2 Mixed Mode Example

A mixed mode example will illustrate this parallel programming model clearly. For the scope of the present project, emphasis will be given to the master-only model. However, the general idea is similar for all mixed-mode variations, as discussed in the previous section.

In a similar fashion with the examples described for MPI and OpenMP, let us assume that once again the program involves computations on a big array. Supposing we use a system of two nodes with four shared-memory processors each, the array would have to be split into two smaller arrays, much like in the MPI example. After splitting and sending the array to the other node, each MPI process would then spawn the remaining
threads of its node and proceed to execute the calculations in parallel. The procedure is shown in Figure 3.9 below.

![Figure 3.9](image)

Figure 3.9: Mixed mode example. An arrays is distributed to two MPI processes which then spawn three threads each to operate on their smaller arrays in parallel.

As discussed in the previous subsection, one MPI process is placed on each node or NUMA region. An obvious difference from a pure MPI implementation is the smaller number of halo communication needed. Since generally two message passing functions are required per neighbouring process (one "send" and one "receive"), a pure MPI implementation running with 8 MPI processes would require sixteen calls to MPI functions per halo exchange. On the other hand, the mixed-mode example described, that also uses 8 parallel elements (PEs), will only require 4 calls to MPI functions per halo exchange, since only 2 MPI processes are used.

### 3.3.3 Advantages and Disadvantages

In theory, the mixed-mode programming model appears to be ideally suited to clustered architectures. Whether or not this is actually the case, however, is a complex question with a non-definite answer. In this subsection, the advantages and disadvantages of mixed-mode programming will be discussed.

First of all, an obvious case where mixed-mode programming may have an advantage is on poorly scaling MPI codes. MPI codes may scale poorly due to a variety of reasons, which generally end up in processes wasting too much time in communication functions. This may be due to load imbalance issues (which are difficult to tackle in MPI) or simple domain decompositions (one-dimensional instead of more, resulting in communication overheads). By using OpenMP, load imbalance can be dealt with easily and
simple domain decompositions are less harmful, since a smaller number of messages is required.

Another obvious case where mixed-mode programming may have an advantage is at programs that use replicated data strategies; in such programs each MPI process has its own copy of a large data structure. This can easily lead to memory limitations as well as to excessive communication costs, since such replicated data would require all-to-all communications. In a mixed-mode implementation of a replicated data algorithm, only one copy of the replicated data structure would be required per node (or NUMA region), thus saving considerable amounts of memory. Moreover, since halo regions are a form of replicated data between neighbouring processes, the resulting reduction in memory requirements applies to a wide variety of applications. Apart from memory savings, communication costs would be greatly reduced, since all-to-all communications between $P$ processes involve $P(P-1)$ total messages. Therefore, the reduction of MPI processes translates to a considerable reduction in communication overheads.

There are also cases of applications that have certain limits to the number of MPI processes that can be executed. In this case, by running the program with the maximum number of processes along with the introduced threaded parallelism, more parallelism than previously allowed will be possible.

Finally, MPI implementations may not be appropriately optimized for intra-node (shared memory) execution. This can have a bad effect on performance, since the shared memory in nodes is not utilized properly. A mixed-mode code may help to alleviate this problem.

However, as was mentioned, "upgrading" a pure MPI program to a mixed-mode version may not always be profitable for performance. A major disadvantage that the introduction of threads can bring is the additional synchronization that may be necessary to ensure thread safety. Synchronization from barriers or critical regions that was not present in the "serial" parts of the pure MPI code may seriously affect performance. Introducing parallel regions into the MPI code also adds the implicit synchronization mechanisms of OpenMP.

Although reducing the number of messages exchanged may be good for performance, having only one process per node exchanging inter-node messages may mean that the maximum bandwidth of the system is not being used. As mentioned in Section 3.3.1, this can be overcome by following a mixed-mode model where all threads call MPI functions. However, implementing these models can be significantly more difficult.

As usual, whether or not mixed-mode performs better than pure MPI depends on a lot of factors, from the application in question to the system that it is going to execute it.
Chapter 4

MG: The CFD Code Investigated

MG is the code that was investigated for the present project. MG was developed by Mantis Numerics and was provided by Professor Sam Falle (University of Leeds, Department of Applied Mathematics). In this chapter, a brief description of the MG code will take place, emphasizing more on the parts that are relevant to the application of mixed-mode programming in the code. Afterwards, a number of initial benchmarks and profiling will be presented, to provide some insight into the performance characteristics of the code.

4.1 MG

MG is a general purpose Computational Fluid Dynamics code that solves the equations of compressible flow, using a conservative upwind finite volume scheme. It uses Adaptive Mesh Refinement, in order to provide greater resolution in spaces where the solution varies rapidly. This is done by halving the grid size in such places, in order to improve the accuracy of the numerical solution.

![Figure 4.1: An example of AMR. The grid size is reduced in regions where the solution varies rapidly, giving better accuracy (reproduced by [13]).](image)
Generally, MG creates a domain of cells, which are the elements of the computational grid used. In serial mode, a single processor has the whole grid in its memory. In parallel, the grid is divided among processors in a one-dimensional decomposition; this is done by cutting the domain across the last dimension entered in the initial conditions specification, as mentioned below.

When initiated, MG uses its own command prompt, providing interactive use. The first step in order to execute any computational work using MG is to provide the initial conditions, depending on the problem being studied. The initial conditions are entered at the MG command prompt in the following fashion:

```
1 START "1D sod test" ! START command and title
2 LEVELS 1 ! Number of levels (numbered from 0)
3 XYZ 1 ! Geometry and number of dimensions
4 X 100 -1 1 FREE ! X No of cells, domain and boundary condition
5 CASE SOD ! Case
6 SCALARS 2 ! No of scalars (optional, default=1)
7 END ! End of START command
```

Listing 4.1: Example initial conditions for MG - Sod problem

Alternatively, a more useful and efficient way is to write the commands above into a text file (i.e. "sod.dat") and simply read the commands from the file. MG will echo the commands as it executes them, setting up the problem. A simple user session in MG could then be:

```
1 MG_G> $read sod
2 MG_G>march 1
3 MG_G>write file
4 MG_G>q
```

Listing 4.2: A sample user session in MG. Initial conditions are read from a "sod.dat" file. A single computational step is performed and finally the result is written to an output file with the name "file" before quitting.

In the previous example, the user types one command at a time. First, the program reads the initial conditions. Then, one step is performed (more steps can also be performed at once) and the output is written to a file. After each of the previous two steps, the program reports the time it took to complete each action. Finally, the user exits the program.

When the program is executed by a single processor (serially), the "write" command will write two output files named "file.000.dsb" and "file.h.dsb". In parallel, running with n processors, the output files are "file.h.dsb" and n output files named "file.i.dsb", where i is the processor number (from 000 to n).

MG also supports graphics, using the included "muvi" library to interpret the graphics commands. Solution variables such as speed and density can be plotted at any time during execution (for example between different steps of the solution). For more details, see [14].
4.2 PARC

MG handles communication between different processors using PARC, a communications library written by Mantis Numerics. PARC was provided by Professor Sam Falle, as part of the MG code. In this section, a brief description of PARC will take place, based on the PARC documentation [15].

PARC provides a higher-level approach to communication between MPI processes for the programmer; it accomplishes this by offering collective communication functions, which the intention of emulating shared-memory systems. PARC offers three communication functions, which will be described next.

First, PARC has a broadcast function that can be used to send data from one PE to all other PEs:

```c
parc_broadcast (a, ndim, dims, ios);
```

The result of this function is to send the element "a" (of "ndim" dimensions of "dims" size each) from the master MPI Process (rank 0) to all other PEs. The last argument (ios) is an error reporting variable, not important to the use of the function.

![Figure 4.2: PARC Broadcast function](image)

The PARC broadcast function can handle arrays of multiple dimensions of different sizes; "a" in the previous figure can be a single variable or a three-dimensional array.

Another useful function in PARC is the "synchronize array" function:

```c
parc_syncarray (array, ndim, dims, ios);
```

Assuming that PEs operate on different parts of a single array but need the whole array for their computations, the `parc_syncarray` function updates the array on every PE.
In the previous figure, we see that every PE operates on different columns of the "shared" array. After calling the parc_syncarray function, all PEs get the updated array, much like a shared-memory scenario where a shared array is visible to all threads.

The final PARC function is the "global data exchange" function, which is a more general communication function, since it can be used to send and receive data from any PE to any other PE:

```
parc_xdata (size, outs, outbuf, ins, inbuf, ios);
```

With this function, a process can send to another process with rank `ipe`, a number of elements specified by `outs[ipe]`, stored in the `outbuf` array. The same PE then receives `ins[ipe]` elements from that rank, stored in the `inbuf` array.

### 4.3 The "step" Function

The main computational function of MG is a function called "step". This function determines the stable time step and finds the solution over that time step. While the details of the computational method used by the function are not relevant for the scope of this project, it is useful to outline its structure, in order to form a basic idea of the parts of the code that are candidates for threaded parallelization. In this section, the structure of the "step" function will be briefly described.

"Step" has a size of approximately 500 lines of code (comments included, no blank lines). The most important parts of the function are the loops where it goes through all `cells` or `joins`, updating each one according to the numerical scheme used; these are the parts that threaded parallelism can be introduced. The loops that go through the cells or joins are in the form of a linked list; a pointer is set to point to the first element in the list and, after performing the work for that particular element, the pointer is set to the next element on the list. After the final element has been processed, the pointer is set to
NULL and the loop breaks. The following pseudocode illustrates the type of loop that is used in the "step" function:

```c
/* Point to the first cell */
pointer = firstCell;

/* Loop */
while (pointer != NULL) {
  /* Work on pointer */
  (...........)
  pointer = pointer->nextCell;
}
```

Listing 4.3: An example of a linked-list loop in "step" function

There are three types of linked-list loops in "step":

1. **cptr loop**: A loop that goes through cells, using pointers named *cptr*.
2. **jptr loop**: A loop that goes through joins, using pointers named *jptr*.
3. **clist loop**: A loop that goes through halo cells, using pointers named *clist*.

The loops above will be referred to using their type names from now on. The code in "step" function is dominated by "cptr" and "jptr" loops. In the following listing, the structure of the step function will be described.

1. **"First" cptr loop**
2. **Communication** (Time step, global max rates)
3. **First order time step**
   i. Halo communication
   ii. For each dimension: **jptr loop**
   iii. **cptr loop**
   iv. if order == 1: Advance Solution, Return
4. **Second order time step**
   i. **cptr loop 2.1**
   ii. Halo Communication
   iii. For each dimension: **cptr loop 2.2**
   iv. **clist loops (last dimension only)**
   v. **jptr loop 2**
   iv. **cptr loop 2.3**
   v. Advance Solution, Increment time, Return

Listing 4.4: Structure of "step" function.
4.4 Initial Benchmarks and Profiling

In order to gain some insight into the performance of the MG code with different problem sizes and increasing numbers of MPI processes, a number of benchmarks was performed initially. Afterwards, the program was profiled, in order to specify the most intensive parts of the code and test its behaviour with large numbers of MPI processes. In this section, the performance results obtained will be described.

4.4.1 Benchmarks

Benchmarks for MG were performed on HeCTOR, without any changes to the code. The only change was made to the header files "struct.h" and "parc.h", in order to allow for bigger numbers of processors, as initially the program allowed a maximum number of 200 processors.

The problem (initial conditions) that was used for the benchmarking purposes was suggested by Professor Sam Falle, and is a three-dimensional version of the two-dimensional "circle" problem, which was included in the code provided. An input file (circle3D.dat) was created, containing the following commands:

```
1 r t o l 0 . 0 1
2 d e r e f c r i t 0 . 2
3 r e f c r i t 0 . 3
4 d i f f b a d 5
5 f a s t
6 S T A R T " c i r c l e x y z "
7   L E V E L S 1
8   X Y Z 3
9   X 1 0 0 2 . 0 2 . 0  F R E E
10  Y 1 0 0 2 . 0 2 . 0  F R E E
11  Z 1 0 0 2 . 0 2 . 0  F R E E
12   S C A L A R S 1 A D V E C T 1
13   C A S E C I R C L E X Y Z
14 E N D
```

Listing 4.5: Circle3D: The initial conditions used for benchmarking

Finally, the following code was added to the `usreqis` function, as suggested by Professor Sam Falle, in order to create the initial conditions for this problem:
```c
else if (strcmp(case, "CIRCLEXYZ") == 0) {
    if (xc[0]*xc[0] + xc[1]*xc[1] + xc[2]*xc[2] <= 1.0) {
        pa[iqd] = 1.0;
        pa[iqe] = 10.0;
        pa[iqal0+1] = 1.0;
    }
    else {
        pa[iqd] = 0.125;
        pa[iqe] = 0.01;
        pa[iqal0+1] = 0.0;
    }
    if (inten)
        pa[iqie] = pa[iqe] / powf(pa[iqd], g);
}
```

Listing 4.6: Circle3D: Code to create the initial conditions

Three different problem sizes were used for the benchmarks that will be described (100, 200 and 300 cells in each dimension). The benchmarks were performed doing 10 computational steps (MG command "march 10"), which means that the "step" function is called 10 times for each benchmark.

The program’s performance was tested with an increasing number of MPI processes, up to the point at which each problem size demonstrated worse performance with the increase of MPI processes. Execution times are summarized in the following three figures, for each of the problem sizes tested respectively.

Figure 4.4: MG, execution time versus number of MPI processes (size: $100^3$)
From the execution times shown in the previous figures, we can see that performance generally improves with the increase of MPI processes, until reaching a plateau and finally becoming worse. This happens at 256, 512 and 1024 MPI processes for $100^3$, $200^3$ and $300^3$ sizes respectively.

It is also evident that the program does not scale ideally; this will be illustrated better by examining the speedup of the program, in Figure 4.7:
It is obvious that, even for the largest problem size, the speedup is far from ideal. The program’s efficiency, as we may see in the following figure, is below the ideal number of 1, and it suffers particularly for the larger numbers of PEs.

This, almost linear, decrease in efficiency with the number of MPI processes is an indication that communication could be dominating the execution time for the larger numbers of PEs.
4.4.2 Profiling

Further examination of the program’s performance characteristics was done by profiling, using the Cray Performance Analysis Tools suite (CrayPAT) in HeCTOR. Analysis of the sampling results was done using the "Automatic Program Analysis" option.

In order to examine the percentage of time spent in communication and quantify the increase of communication costs with the number of MPI processes, profiles were made with different numbers of PEs. Four profiling runs were performed, with 2, 4, 128 and 256 PEs, representing the smaller and larger numbers of PEs. A size of $200^3$ cells was chosen for the profiling runs and 10 steps were performed. The results of the profiles are shown in the following tables; other than MPI functions, only the five most time-consuming functions of MG are shown.

<table>
<thead>
<tr>
<th>2 MPI Processes</th>
<th>4 MPI Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Spent (%)</td>
<td>Function Name</td>
</tr>
<tr>
<td>16.8</td>
<td>step</td>
</tr>
<tr>
<td>15.4</td>
<td>hflux</td>
</tr>
<tr>
<td>7.4</td>
<td>connect_cell</td>
</tr>
<tr>
<td>7.4</td>
<td>usrfxp</td>
</tr>
<tr>
<td>6.9</td>
<td>find_cell</td>
</tr>
<tr>
<td>MPI Functions</td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td>MPI_Recv</td>
</tr>
</tbody>
</table>

Table 4.1: Profiling with small numbers of MPI Processes

<table>
<thead>
<tr>
<th>128 MPI Processes</th>
<th>256 MPI Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Spent (%)</td>
<td>Function Name</td>
</tr>
<tr>
<td>8.8</td>
<td>step</td>
</tr>
<tr>
<td>5.2</td>
<td>hflux</td>
</tr>
<tr>
<td>4.3</td>
<td>connect_cell</td>
</tr>
<tr>
<td>4.0</td>
<td>find_cell</td>
</tr>
<tr>
<td>3.0</td>
<td>rem_cell</td>
</tr>
<tr>
<td>MPI Functions</td>
<td></td>
</tr>
<tr>
<td>23.7</td>
<td>MPI_Recv</td>
</tr>
</tbody>
</table>

Table 4.2: Profiling with large numbers of MPI Processes

In table 4.1, we see that, for small numbers of PEs, communication costs are relatively low. However, in table 4.2 it is evident that increasing the PEs creates a bottleneck in communication; with 256 processes, nearly one-third of the time is spent in the MPI_Recv function.

The next figure visualizes the increase of communication costs with the increase of MPI processes, showing the percentage of time spent in all MPI functions (not just MPI_Recv) versus the number of PEs.
It is clear that communication certainly affects the program’s performance when increasing the number of PEs. It is thus interesting to investigate the impact that the introduction of threaded parallelism will have in the performance of MG.
Chapter 5

Implementation

The purpose of this project was to get a "real-world" code, unfamiliar to anyone working on this project, and try to introduce OpenMP threads to transform it into a mixed-mode code. The code chosen for this project was based on a short discussion between the author of the code, Professor Sam Falle, and the dissertation project supervisor, Dr. Chris Maynard. It was thus a part of the project to get the code and read the documentation and the code, in order to get an basic idea about the code, before doing any implementation.

As the initial project proposal was based on the communication library that MG uses (PARC), and specifically on the fact that it uses MPI to emulate shared-memory systems, initial investigation went into the code of the PARC library. This was a non-trivial task, as the library consists of 12 source files, containing approximately 2000 lines of code. After getting a basic idea of the structure of the communication functions, it was realized that introducing threads inside PARC would not be possible. After a discussion with the project’s supervisor, it was decided to leave PARC as the communication library for the program and introduce threads in MG itself.

MG consists of 23 source files, containing approximately 11000 lines of code. A basic idea of its structure was acquired after going through the code and the basic parts of the computational functions were identified. Porting the code to the two systems that were used (Ness and HECToR) proved to be rather hard; the help of Dr. Chris Maynard was crucial to this process at the beginning of the project, along with help from HECToR’s helpdesk support.

In this chapter, the implementation work done will be presented, after a short description of the computer systems that were used (Ness and HECToR) in Section 5.1. Two different implementations were done; first, the linked-list loops of the "step" function were parallelized manually, as shown in Section 3.2.5. Then, a different approach was taken, using OpenMP tasks. These two versions will be described in Sections 5.2 and 5.3 respectively. The tests that were used in order to ensure correctness will be described in Section 5.4.
5.1 Computer Systems Used

In this section, the HPC systems that were used for this project will be described. Two different HPC systems were used; Ness, a small shared-memory system and HECToR, the UK national supercomputer.

5.1.1 Ness

Ness is a small shared-memory machine, mainly used for the purposes of the MSc in High Performance Computing in EPCC. Ness was chosen for developing, testing and performing small benchmarks, because of its ease of use and availability compared to a large machine, such as HECToR. Continuous compilation, short interactive test runs and jobs submitted by script were far easier to make on Ness, as the queues were virtually empty compared to HECToR.

Ness consists of the front-end and the back-end. The front-end is used for logging on, editing and compiling programs, and it consists of a dual-processor system. It can also be used to run executables, but only for short test runs. The back-end consists of two 16-core shared-memory boxes, and it is where larger jobs are submitted via the job scheduling system. All processors on Ness are 2.6 GHz AMD Opteron processors, with 2 gigabytes of memory each.

The operating system used on Ness is GNU/Linux 2.6.18 and the distribution is "Scientific Linux". PGI and GNU compilers are available on Ness; however, compilation of MG was only possible using gcc, the GNU C compiler, which is not the ideal choice for performance. The job submission system on Ness is the Sun Grid Engine (SGE).

5.1.2 HECToR

HECToR, the UK national supercomputer, is a Cray XE6 system (known as phase 3). HECToR consists of 2816 XE6 nodes, with each node containing two AMD 2.3 GHz 16-core processors, resulting in a total of 90,112 cores. Each node has 32 GB of shared memory available for its 32 cores, resulting in a total of 90 TB of memory. Particularly important for mixed-mode applications is the division of each shared-memory node into NUMA regions. Each 32-core node consists of 4 NUMA regions, with 8 cores each. The login nodes of HECToR consist of a dual-core 2.6 GHz AMD Opteron processor and 8 GB of RAM. Processors in HECToR are connected via a high-bandwidth interconnect, using Cray Gemini communication chips arranged on a three-dimensional torus. Every communication chip is shared between two XE6 nodes.

The operating system used on HECToR is the Cray Linux Environment. It consists of Compute Node Linux for the compute nodes, which is a stripped down version of Linux,

---

1Information about the systems can be found in their respective user sites: http://www2.epcc.ed.ac.uk/ness/ and http://www.hector.ac.uk/support/documentation/userguide/
designed to be lightweight in order to minimize the operating system interruptions to the compute process. The service nodes (login, I/O and system) feature a full Linux distribution.

The compilers available on HECToR are GNU, PGI and Cray. However, as mentioned before, compilation was only possible with gcc, which was again the compiler used on HECToR.

5.2 Introducing Threads

In this section, the introduction of OpenMP to the MG code will be presented. The general concept of parallelizing the linked-list loops will be described and any relevant information regarding this process will be discussed. A visualization of the structure of "step" function is presented in Figure 5.1, to help make the relevant parts of the code more clear and provide easy reference for the discussion of this chapter.

![Figure 5.1: "Step" function: The main parts of the first order (a) and second order (b) steps.](image)

5.2.1 Replacement of "while" Loops

As shown in Section 4.3, there are three different kinds of linked-list loops in the "step" function. The implementation process was as follows. The different kinds of loops were approached separately, dealing with "cptr" loops first, "clist" loops later and "jptr" loops last. Each time OpenMP threads were introduced to a loop, correctness tests took place; these tests will be described in section [FILL LATER].

As discussed in Section 3.2, loops without well-defined boundaries need some extra preparatory work in order to be parallelized by OpenMP. The concept is the same for all linked-list loops in the "step" function. The addresses of the linked-list elements
must be stored in an array; then, the "while" loop can be replaced by a "for" loop, which can finally be parallelized with OpenMP worksharing constructs.

As shown in Figure 5.2 (a), originally a linked-list loop goes through every element of the list and, after completing the work associated with that particular element, it points to the next element of the list. The process repeats until the loop reaches the final element.

```
cptr = lev[ilev]->fcell;
while (cptr != NULL) {
    /* Do Work */
    /* (...). */
    cptr = cptr->nxt;
}
```

Listing 5.1: Example of an original linked-list loop in "step" function.

In order to parallelize such a loop with OpenMP, it is first necessary to replace the "while" loop with a "for" loop, so that it can have known boundaries at runtime. This can be achieved by using an array of pointers that will point to the elements of the linked-list; a "for" loop will then go through these pointers and perform the work associated with every item of the list, as shown in Figure 5.2 (b).

This replacement of loops requires some preparatory work. For every linked-list loop, a certain number of steps need to be taken:

1. Count the number of linked-list elements.
2. Allocate an array of pointers to the elements.
3. Point each pointer of the array to a corresponding linked-list element.

The preparation process is shown in the following listing:
This procedure was followed for every linked-list loop in "step" function. The types of data involved in the computation taking place in "step" are data structures, defined in a separate file ("struct.h"). The two main structures are namely cells and joins; clist loops go through cell_list structures, which point to cell structures. As we can see from the previous listing, the data structure allocated for each loop depends on the structure that loop works with. This means that the array of pointers allocated for each loop must be of the same data type as its corresponding loop.

Finally, the "while" loops were replaced with "for" loops, iterating through the number of elements that was counted in the first step:

In the previous listing, we see that the only additional code needed inside the loop is to point the cp ptr pointer to the current pointer of the array (cp ptr[t]). Moreover, the last line of the "while" loop (cp ptr = cp ptr->nxt;) is no longer necessary, since now the loop iterates through elements of the array of pointers, using the iterator t. After all work involving the array of pointers is finished, the array can be freed.
5.2.2 Parallelizing the "for" Loops With OpenMP

After the replacement of the "while" loops with "for" loops, OpenMP can be introduced to execute the new "for" loops in parallel, with threads. The parallelization process involves placing the loops inside parallel regions and applying a work-sharing directive immediately before the for loop. In this way, the master thread will spawn its team of threads at the start of the parallel region and the iterations of the loop will be automatically distributed to different threads by the work-sharing directive.

```c
#pragma omp parallel shared(cptr_ptr, ...) 
    private(t, cptr, ...) 
    firstprivate(cptr_counter, ...) 
    default(none)
{
    #pragma omp for schedule(type, chunk) 
    for (t=0; t<cptr_counter; t++) 
        /* Point cptr to current array element */
        cptr = cptr_ptr[t];
        /* Do Work */
        /* (...) */
}
```

Listing 5.4: Parallelization of a "for" loop, shown for a "cptr" loop as an example.

The general concept of parallelizing the "for" loops with OpenMP, as seen in Listing 5.4 is simple; threads will be assigned different iterations of the "for" loop, which represent different elements of the array of pointers. In other words, each thread will get a different subset of the loop’s working data structure (cptr, jptr or clist), and threads will go through their subsets in parallel.

Assigning the data scope of the variables requires some careful thought. First of all, certain variables form a common theme in loops; the pointers to data (cptr and cptr_ptr in Listing 5.4) and the loop-related variables (t and cptr_counter in Listing 5.4). The data scope of these variables is as follows:

- **cptr_ptr**: The array of pointers is declared as shared. In this way, threads will be able to access their parts of the single array.
- **t**: The iterator is declared as private. Each thread gets an uninitialized copy of t, and uses it to iterate through its part of the loop, according to the work-sharing schedule.
- **cptr**: The working pointer of the loop is declared as private. Each thread gets an uninitialized copy of this pointer variable, which then points to the appropriate data in every iteration (Listing 5.4, line 9).
- **cptr_counter**: The counter of the linked-list elements has the purpose of the upper loop boundary. This means that its value is set before the parallel region and
remains unchanged. It is also needed by all threads; consequently, it is declared as a firstprivate variable.

There are of course other variables used inside the various loops of the "step" function. Their data scope has been chosen after careful examination of the code; the factors considered regarding their scope are summarized below:

- **Private**: Variables that are not initialized before a loop and their value is set in each iteration are set as private variables.

- **Shared**: Variables that are set before a loop, needed after the loop and are accessed by all threads are set as shared variables. With the exception of the first cpctr loop in "step" function (as will be described in Section 5.2.3), only the arrays of pointers have been set as shared variables.

- **Firstprivate**: Variables that are initialized before loops and their value is needed for computations inside a loop are set as firstprivate. These variables are not altered inside the loops, but their value is used to perform calculations on the data structures.

After introducing OpenMP to a loop, correctness tests took place, in order to ensure that the parallelization process was correct. These tests will be described in Section 5.4.

### 5.2.3 Find Maximum: Scaling Investigation

An interesting part of the single cpctr loop right before the first order step, is the determination of maximum rates (variables hrate and vrate). This part can be implemented in two ways in parallel; in this subsection these two solutions will be described and their performance will be investigated.

As the loop goes through each cell, it determines the value of the hrate variable. Then, it checks if the newly calculated value is bigger that the maximum value so far and, if so, it replaces it with the new value. The procedure is shown in Listing 5.5:

```c
for (t=0; t<cpctr_counter; t++) {
    cpctr = cpctr_ptr[t];
    /* Work (...) */

    /* Maximum rate */
    for (id = 1; id <= nd; id++) {
        divdx = cpctr->rjoin[id-1]->area/cpctr->vol;
        hrate = max(hrate, (mabs(cpctr->pa[iqu0+id]) + ss)*divdx);
    }
    /* Work Continued (...) */
}
```

Listing 5.5: Maximum rate in the first cpctr loop
In serial, this algorithm is simple; going through one cell at a time, the maximum value for \( h_{rate} \) is determined after the loop has gone through all cells. In parallel, however, this procedure is different, since each thread goes through its own subset of cells. Finding the maximum among all cells now needs some extra caution.

There are two main ways to achieve the determination of these maximum values in the loop. The first and most straightforward way is to declare the rate variables (\( h_{rate} \) and \( v_{rate} \)) as shared variables. Afterwards, the lines where the rates are set (line 7 in Listing 5.5) will be enclosed inside a critical region, in order to avoid race conditions. In this way, each thread will update the shared rate variables with the largest value found at each particular point in the loop. While this is a simple solution, the use of critical regions is generally bad for performance.

```c
for (t=0; t<cptr_counter; t++) {
    cptr = cptr_ptr[t];
    /* Work (...) */

    /* Maximum rate */
    for (id = 1; id <= nd; id++){
        divdx = cptr->rjoin[id-1]->area/cptr->vol;
        #pragma omp critical
        {
            hrate = max(hrate, (mabs(cptr->pa[iqu0+id]) + ss)*divdx);
        }
    }
    /* Work Continued (...) */
}
```

Listing 5.6: Finding the maximum in parallel using critical regions (\( nd \): number of dimensions)

The second way to find the maximum in parallel, is to use a shared temporary array for each of the two rate variables. In each element of these arrays, each thread will write the maximum value it has encountered in its own subset of cells. Finally, after the end of the parallel region, the master thread will go through the temporary arrays and determine the total maximum, writing it to the original rate variables (\( h_{rate} \) and \( v_{rate} \)). This solution involves additional code than the previous one, however it avoids the critical region. The concept of this solution is shown in Listing 5.7.
```c
#pragma omp parallel
{
  int myid = omp_get_thread_num();
#pragma omp for
  for (t=0; t<cptr_counter; t++)
    cptr = cptr_ptr[t];

    /* Work (...) */
  for (id = 1; id <= nd; id++)
    divdx = cptr->rjoin[id-1]->area/cptr->vol;
    hrate_temp[myid] = max(hrate_temp[myid], (mabs(cptr->pa[iqu0+id]) + ss)*divdx);

    /* Work Continued (...) */
}/*! End of parallel region */
for (i=0; i<MAX_THREADS; i++)
  if (hrate_temp[i] > hrate) {
    hrate = hrate_temp[i];
  }
```

Listing 5.7: Finding the maximum in parallel using temporary arrays

Comparing the performance of the two solutions will show which of the two scales better with an increasing number of threads and will help to choose the best solution too implement for the final code. Two versions of the first solution were tried; one having only the single "max" command (line 5 in Listing 5.5) inside the critical region, and one having the whole "rate" loop (lines 5-8 in Listing 5.5) inside the critical region.

The idea behind placing the whole "rate" loop inside the critical region is to reduce the effect of the critical region on performance. When only the single "max" command is inside the critical region, the size of the critical region is smaller, but this also means that there is effectively one critical region for each loop iteration. By placing the whole loop over \( nd \) in the critical region, each thread may finish its loop faster, taking advantage of its cache memory and resulting in less time spent waiting outside the critical region.

The cptr loop was timed and performance tests were made on \textit{Ness} for the three cases mentioned above, for two different sizes \((100^3\) and \(200^3\) cells), with 1, 2, 4 and 8 threads and a single computational step. The execution times and speedup for the different sizes are shown in Figures 5.3 and 5.4 respectively.
We see that both variations of the method involving critical regions have bad scaling, resulting in increasingly worse performance for the increasing number of threads tested. On the other hand, the method involving the temporary shared arrays results in better performance, which is more evident for the larger size ($200^3$ cells); however, the method does not scale ideally. Because of the obvious better results of the temporary array method, it was the one chosen for parallelizing the $cptr$ loop before the first order step.

### 5.2.4 Loop Scaling With OpenMP

After parallelizing the loops in "step" function, as described in the previous section, it is interesting to investigate their scaling with different numbers of threads. Each loop was timed using the OpenMP library timing function ($omp_get_wtime()$), and the code ran with a single MPI process and 1, 2, 4 and 8 threads.

These loop scaling tests were made on Ness, before porting the program on HECToR, as these are not final performance tests, but rather small performance runs to see how the program responds to the introduction of OpenMP parallelization.
The results of the scaling investigation are shown in Figures 5.5 - 5.8, for two different problem sizes (100$^3$ and 200$^3$ cells) respectively. For easy reference, the loops were divided into two groups; the first and second order loops, as shown in Listing 4.4 and Figure 5.1. The first cpdr loop, before the first order step, was included in the first group.

![Figure 5.5: Loop execution times versus number of threads (size: 100$^3$ cells)](image)

![Figure 5.6: Loop speedup versus number of threads (size: 100$^3$ cells)](image)

For the smaller test size (100$^3$ cells), seen in Figures 5.5 and 5.6, we see that the three most computationally expensive loops demonstrate reasonably good scaling. Loop cpdr 2.2 and jptr loops in both order steps achieve ideal speedup for two threads and almost linear for four. They still manage to get better performance with eight threads, but evidently less than ideal.

The more light loops, on the other hand, do not scale as well. The first cpdr loop does achieve shorter execution times, but its scaling is far from ideal. Loops cpdr 2.1 and 2.3 scale well up to four threads, but fail to achieve better performance for eight threads, with cpdr 2.1 actually performing slightly worse.
For the larger problem size ($200^3$ cells), seen in Figures 5.7 and 5.8, loops follow a similar pattern with the smaller size, discussed previously. It is noteworthy that the two jptr loops now achieve even better speedup for eight threads.

These scaling tests for the parallelized loops indicate that the program responds well to the introduction of OpenMP threads. However, when running with many MPI processes in mixed-mode, cells will be distributed among them, resulting in less work for each process. As we have seen from the previous scaling investigation, the most expensive loops achieve better threaded performance for larger loop sizes. It is thus interesting to perform the mixed-mode performance tests and see what the overall performance of the program will be.
5.3 OpenMP Tasks

The second way to parallelize the linked-list loops is to use OpenMP tasks. In this section, the process of parallelizing the loops of the "step" function will be described.

Needless to say, linked-list traversals are very well suited for tasks, as discussed in Section 3.2.5, as no replacement of the "while" loops is necessary, resulting in a more elegant implementation. The general concept of parallelizing the linked-list loops with tasks is similar for all three kinds of linked-list loops in the "step" function, as shown in listing 5.8.

```c
#pragma omp parallel private(cptr, myid, ...) shared(...) firstprivate(...) default(none)
{
    #pragma omp single
    {
        cpdr = lev[ilev]->fcell;
        while (cpdr != NULL) {
            #pragma omp task private(myid, ...) shared(...) firstprivate(cpdr, ...) default(none)
            {
                /* Do Work - Task code */
                myid = omp_get_thread_num();
                /* (...) */
            }
            cpdr = cpdr->nxt;
        }
    }
}
```

Listing 5.8: Parallelizing a linked-list loop in "step" function using tasks.

In Listing 5.8, we see that a single thread inside the parallel region assigns the tasks. First it points the cpdr pointer to the first item of the list (line 8). Then it enters the while loop, packaging the current iteration as a separate task, to be executed by any thread of the team. Exiting the task code, it finally points the cpdr pointer to the next item on the list. The process is repeated until the final item of the list has been reached.

What is important in the task version of the code is the fact that the working pointer (cpdr in the example shown) needs to be declared as firstprivate in the task clause. This is because tasks are distinguished by the list item they operate on. The thread that packages tasks, does so by pointing the cpdr pointer to consecutive items of the linked list. The thread that will execute the task needs to know which element it will be working on. Thus, by declaring the cpdr pointer as firstprivate, the executing thread will get the address of the appropriate list item for the task it is going to execute.
Another important thing to note is the scope of the variable that holds the number of the task-executing thread (\textit{myid}). This variable is declared as \textit{private} in the task clause, and its value is set \textit{inside} the task (Listing 5.8, line 16). By doing so, each thread that executes a task will set its own ID to \textit{myid}. If the variable was declared as firstprivate, all instances of \textit{myid} would contain the number of the thread that packages the tasks.

The scope of other variables inside the linked-list loops was declared according to the same concepts described in Section 5.2.2. Again, the \textit{default(none)} clause was used in order to explicitly declare every variable and avoid mistakes.

\section*{5.3.1 Loop scaling with OpenMP tasks}

The development of the OpenMP tasks version of the code took place on HECToR, for reasons that will be explained later (Section 5.4.1). After implementing the tasks version, the scaling properties of the loops was investigated, by running the program with a single MPI process and 1, 2, 4 and 8 threads. Two different problem sizes were investigated (100$^3$ and 200$^3$ cells).

The performance tests took place on HECToR, so execution time comparison with the manual version (performed on Ness) is not meaningful. However, the general scaling behaviour of the loops will become evident, especially from the speedup results. Results for the smaller problem size are shown in Figures 5.9 and 5.10. Loops are again divided into two groups (orders 1 and 2).

\begin{figure}[h]
\centering
\begin{subfigure}[b]{0.45\textwidth}
\includegraphics[width=\textwidth]{fig5_9a}
\caption{First order loops}
\end{subfigure}\hspace{0.5cm}
\begin{subfigure}[b]{0.45\textwidth}
\includegraphics[width=\textwidth]{fig5_9b}
\caption{Second order loops}
\end{subfigure}
\caption{Tasks version: loop execution times versus number of threads (size: 100$^3$ cells)}
\end{figure}

47
From the results in the previous figures, it is evident that the OpenMP tasks version of loops do not scale well, actually performing worse for the increasing numbers of threads. The larger problem size ($200^3$) exhibits a similar behaviour, as seen in Figures 5.11 and 5.12:
It is obvious that parallelizing the linked-list loops with OpenMP tasks seriously hampers the performance of the code, which actually becomes worse when more threads are spawned. The reason is probably the large amount of tasks generated, along with the fact that each task is not particularly heavy.

For instance, regarding the smaller size of $100^3$ cells, the program uses 1,000,000 cells and 3,030,000 joins. This means that there are 1,000,000 tasks generated for every $cptr$ loop and 3,030,000 tasks generated for every $jptr$ loop. Distributing such numbers of tasks to all threads and synchronizing their work seriously affects the program’s performance.

Using the smaller size ($100^3$ cells) again as an example, consider one of the loops; for instance, the first $cptr$ loop. This loop, as shown in Section 5.2.4 (Figure 5.5 - a), takes approximately one second to complete in serial (one thread). For $10^6$ iterations, this means that each iteration takes roughly $10^{-6}$ seconds to complete. However, according to [16], the overhead for creating a single task when running with only one thread is at the order of $10^{-6}$ seconds, and becomes significantly larger as the number of threads increases. In other words, the overhead for creating each task, as shown in Listing 5.8, is equal or greater than the time it requires to complete.

### 5.4 Correctness Testing

As mentioned in Section 5.2, after every step of the implementation, correctness tests were performed. The tests, being part of the implementation process, were performed on *Ness*. In this section, the tests and testing process will be discussed.

The testing process was suggested by Professor Sam Falle, and involved checking the results using the interactive graphics of the program. Initially, the original program (without any alterations) was executed, using the suggested initial conditions (*circle3D.dat*). The solution was then written to output files and stored in a separate directory for testing. In this way, the solution could be read by the program at a later time and visualized by the program’s graphics.

In order to produce graphics, a graphics window is first set up using the following command that executes the appropriate commands from file *x.dat*, provided with the code:

```
MG_G> $read x
```

Once the window is set up, variables of the solution can be visualized. Professor Falle’s suggestion was to visualize the density along each of the three dimensions. Drawing along one dimension requires "slicing" along the other two, since this is a three-dimensional problem. In other words, to visualize the density along dimension "x", the other three dimensions are sliced at zero. This is achieved by the "mask" command in MG. The whole process involves the following commands:
Listing 5.9: Plotting density along the x axis on MG

First, coordinates "y" and "z" are masked, choosing to plot any chosen variable along the "x" dimension. Then, the variable to be plotted is chosen (rho stands for density). Finally, "p" plots the density of the current problem state. After plotting along "x", it is possible to plot along other directions, by using the same commands, after erasing previous plots from the graphics window and then unmasking the coordinates:

Listing 5.10: Erasing previous plots and unmasking all coordinates

Using the original code, the evolution of the density of the problem is shown for 20 computational steps in Figure 5.13, visualized at two steps at a time. This involved performing two steps and then plotting, repeated for ten times.

Figure 5.13: Evolution of density along the x axis, for 2x10 computational steps (size: $20^3$ cells), using the original MG code.

The main correctness checking process involved testing the code after any alteration, with identical runs (same initial conditions, problem size and number of steps). During the process of manually introducing threads, as described in Section 5.2, tests were performed both after the replacement of every "while" loop and after the parallelization of each loop.
A first test was to reproduce the density evolution with the altered code, and ensure that the solution was, at least qualitatively, identical to the solution of the original program (shown in Figure 5.13). An example of one such qualitative test is shown in Figure 5.14, executed with four MPI processes and four threads, after all loops were parallelized with OpenMP (as described in Section 5.2):

![Figure 5.14: Evolution of density along the x axis, for 2x10 computational steps (size: $20^3$ cells), after parallelizing the linked-list loops. The test run was made using 4 MPI processes with 4 threads each.](image)

We can see that the solution is identical to the original program, showing that the results are correct. However, the final test to ensure that the introduction of threads is correct involves plotting both versions on the same window. If both plots coincide, then the solution is identical. The result of the original code is shown in Figure 5.15.
Figure 5.15: Density along the x axis for 10 computational steps, using the original MG code (size: $20^3$ cells)

In Figure 5.16, both versions are plotted on the same window. This is achieved by running both versions and saving the problem state to output files. Then, MG is started and a graphics window is initiated. Finally, the two models are read consecutively, without erasing the plot.

Figure 5.16: Density along the x axis for 10 computational steps, using the original MG code and the threaded version on the same window - plots coincide (size: $20^3$ cells)

We see that the two versions have the same results; the parallelization of the linked-list loops was thus correct. It should be noted that the tests described were performed for all three dimensions.
5.4.1 Testing the OpenMP Tasks Version

The tests described so far took place during the implementation process, on Ness. However, it was not possible to implement the OpenMP tasks version of the code on Ness; the gcc compiler available is an older version (version 4.1.2) that does not support OpenMP 3.0 - the version of the OpenMP specification that first introduced tasks.

The code was then ported to HECToR. However, while the newer version of gcc (version 4.6.3) available in HECToR supports OpenMP 3.0, it is also incompatible with the output functions of the MG code. Specifically, the functions that write the output to memory use deprecated code, resulting in compiler warnings 2 when using the new version of gcc. While the program initiates and runs without problem, it does not write output.

After unsuccessful attempts to fix this problem, it was realized that this would involve a lot of work in the output parts of the code, that were of course not relevant to the project’s aim. It was thus decided that such work was outside the scope of the project, especially due to the relatively limited time of the dissertation period. As a result, any implementation involving OpenMP tasks cannot be tested for correctness on HECToR.

However, the data scope of the variables inside the parallel regions remains generally the same, with the exceptions that were described in section 5.3. A test was made to ensure that the appropriate number of tasks is generated at every loop, by counting the number of tasks generated and printing their number. Also, another test was performed to ensure that the tasks are distributed to all available threads, by printing the thread ID of the thread executing each task. Those two tests are not sufficient to show that the task version of the code is correct; however, they show that the correct amount of work is performed, and it is executed in parallel.

---

2 Compiler warning: "deprecated conversion from string constant to 'char *'", for functions in source files plot.c, mvcmd.c, readxq.c, modelrw.c, top.c, usrc
Chapter 6

Results

In this chapter, the benchmark results of the mixed mode implementation of the MG code will be presented. First, the performance results (execution time, speedup) will be shown; afterwards, a discussion regarding memory usage will take place.

As described in Section 5.2.3, the scaling of the parallelized loops was investigated for 2, 4 and 8 threads. Generally, the best scaling behaviour for all loops was encountered at 4 threads. At 8 threads, the most computationally intensive loops continued to scale well, but the other loops did not result in considerable benefit (Figures 5.6 - b, 5.8 - b). Moreover, larger problem sizes showed better scaling than smaller ones (100^3 vs 200^3 cells). Also, regarding the MPI parallelization of the program, we have seen that the code suffers from large communication costs at larger MPI process numbers (Figure 4.9).

Taking the above into account, there appears to be an interesting trade-off in mixed-mode. More threads per MPI process will result in less MPI processes in total, which is good for reducing communication costs at high numbers of PEs. At the same time, having less MPI processes means that the problem size is divided to fewer, larger pieces; each MPI process gets a bigger personal chunk of the problem. However, running with more than 4 threads per MPI process does not take advantage of the best possible scaling for all loops.

To investigate the behaviour of the code with the above trade-off in mind, the code was tested in mixed-mode with 2, 4 and 8 threads per MPI process, corresponding to 4, 2 and 1 MPI processes placed per NUMA region. Tests with more threads per MPI process were not performed, since they did not show good performance in scaling tests.

The scaling investigation in Section 5.3.1 showed that the OpenMP tasks version of the code, while being more elegant and straightforward, resulted in significantly bad performance results. Consequently, the manual OpenMP threads implementation was chosen for the final benchmarks in this chapter. Performing a whole series of benchmarks for the tasks version would not produce any meaningful results, other than showing the bad performance of the code and consuming CPU time.
Benchmarks were performed on HECToR, using the manual OpenMP implementation described in Section 5.2. The benchmarks were similar to the original MG code benchmarks (Section 4.4.1), for comparison purposes; 10 computational steps for three problem sizes ($100^3$, $200^3$ and $300^3$ cells), using the three dimensional circle problem ($circle3D.dat$). Benchmarks for each problem size were executed for large numbers of PEs, starting before the PE count for which each problem size shows worse performance (Figures 4.5 - 4.7).

Using the `aprun` options, it was ensured that the threads spawned by each MPI process would belong to the same NUMA region as their master thread, by running the program as shown below:

```
aprun –n 4 –N 4 –d 8 –S 1 ./mg
```

Listing 6.1: An example of running MG in mixed mode using aprun

The command shown in Listing 6.1 initiates MG with 4 MPI processes total (`-n 4`), placing all four in a single node (`-N 4`). It will also assign 8 cores as threads to each MPI process (`-d 8`). Finally, the `-S 1` option places only one MPI process per NUMA region. In a similar fashion, the `aprun` command can provide the desirable combination of threads per MPI process for each benchmark.

### 6.1 Mixed-Mode Benchmarks

In the following subsections, the benchmarks for each of the three combinations of threads per MPI process will be presented, compared to the original MPI-only version of the code. The three different combinations will be presented successively, showing execution time and speedup for each of the three problem sizes examined. Finally, all three combinations will be presented together for comparison.
6.1.1 2 Threads per MPI Process

Execution times and speedup for the three problem sizes, using 2 threads per MPI process, are shown in Figures 6.1 - 6.3 below.

![Execution Time Graph](image1)

(a) Execution Time

![Speedup Graph](image2)

(b) Speedup

Figure 6.1: MG benchmark: mixed-mode (2 threads/MPI process) and MPI-only (size: $100^3$ cells)
Figure 6.2: MG benchmark: mixed-mode (2 threads/MPI process) and MPI-only (size: \(200^3\) cells)
As seen from the figures above, there is a common behaviour for all sizes examined. The mixed-mode implementation performs equally or slightly better than the original MPI-only code for small PE counts for each problem size. As the number of PEs increases, the mixed-mode implementation also starts to lose performance, though still giving better results compared to the MPI-only implementation.
6.1.2 4 Threads per MPI Process

Execution times and speedup for the three problem sizes, using 4 threads per MPI process, are shown in Figures 6.4 - 6.6 below.

Figure 6.4: MG benchmark: mixed-mode (4 threads/MPI process) and MPI-only (size: $100^3$ cells)
Figure 6.5: MG benchmark: mixed-mode (4 threads/MPI process) and MPI-only (size: $200^3$ cells)
Running with 4 threads per MPI process we see that, for the small PE counts, the original MPI-only version of the code performs slightly better than the mixed-mode implementation; the communication costs for these small numbers of PEs do not significantly affect performance. However, as the PE number increases for each size, we see that the mixed-mode implementation continues to achieve better performance, even for the larger PE counts.
6.1.3 8 Threads per MPI Process

Execution times and speedup for the mixed mode implementation are shown for the three problem sizes examined, in Figures 6.7 - 6.9, along with the original MPI-only implementation results for comparison.

Figure 6.7: MG benchmark: mixed-mode (8 threads/MPI process) and MPI-only (size: $100^3$ cells)
Figure 6.8: MG benchmark: mixed-mode (8 threads/MPI process) and MPI-only (size: $200^3$ cells)
With 8 threads per process, the mixed-mode version of the code performs significantly worse than the original version for the smaller PE counts examined for each problem size. However, towards the larger numbers of PEs for each size, the mixed-mode implementation again keeps achieving better performance when the MPI-only version slows down.
6.1.4 Comparison

After showing the different combinations of threads per MPI process individually compared to the MPI-only implementation, the speedup results of all combinations will be presented together, in the following figures, for comparison.

Figure 6.10: Speedup vs PEs, for all combinations of threads per MPI process (size: $100^3$)

For the problem size of $100^3$ cells, seen in Figure 6.10, we see that the mixed-mode implementation with 4 threads per MPI process achieves the higher speedup for 256 PEs, followed by the 8 thread mixed-mode code. For the smaller PE counts, the performance of the mixed-mode code with 2 threads and 4 threads is roughly the same, gaining significant ground on the MPI-only code for 128 PEs. This is not the case for 8 threads per MPI process; for 32 and 64 threads this combination performs worse than all other versions, and only becomes equal to the original MPI code for 128 PEs.
The scaling behaviour of the mixed-mode code for the size of $200^3$ is similar to the smaller size of $100^3$ cells, as we can see in Figure 6.11. It is worth noting that the bigger problem size resulted in larger speedup numbers overall. The mixed-mode code clearly outperforms the MPI-only code for 512 PEs. The best performance is achieved again when running with 4 threads per MPI process, followed by 8 and 2 threads.

Figure 6.12: Speedup vs PEs, for all combinations of threads per MPI process (size: $300^3$)
The largest problem size of $300^3$ cells resulted in even better speedup results, as seen in Figure 6.12. Again, the mixed-mode implementation outperforms the original code for the larger number of PEs (1024), for all numbers of threads per MPI process. In general, the behaviour of the three different combinations of threads per MPI per process is similar with the smaller sizes.

Overall, the three combinations of threads per MPI process respond differently to different counts of PEs. For 2 threads per MPI process, the mixed-mode code performs equally or better than the MPI-only code for all numbers of PEs investigated; however, it shows slow-down for the largest PE counts. Using 4 threads per MPI process, the mixed-mode code achieves the best performance at the largest numbers of PEs, for all sizes, but performs worse than the original code for the smaller numbers of PEs.

For 8 threads per MPI process, the mixed-mode code performs significantly worse than the MPI-only code for small numbers of PEs. This is reasonable since, as seen in Section 5.2.4, the linked-list loops did not show significant performance benefits when increasing the number of threads from 4 to 8. However, scaling with 8 threads is generally good and it performs particularly well for the largest PE counts, being second only to the 4-thread version.

The effects of the trade-off, discussed in the beginning of the present chapter, are thus evident. In the PE counts where the original MPI implementation suffered particularly bad performance due to excessive time spent in communication, the mixed mode implementation manages to achieve better performance. Using less MPI processes, MPI communication time is reduced, while the linked-list loops are executed in parallel by threads.

On the other hand, in the lower PE counts, the mixed mode implementation does not have similarly good performance compared to the MPI-only version. Since the original code spends less time in communication for small PE numbers, the performance benefit gained from threading the loops is not enough to account for the less MPI processes utilized.

### 6.2 Memory Usage

Apart from achieving better performance in the larger numbers of PEs, the mixed-mode implementation also requires less memory for the same number of PEs (MPI Processes × threads). As discussed in Section 3.3.3, mixed-mode programming can save memory in cases of replicated data. In the case of MG, the halo regions are a form of replicated data, as different MPI processes have copies of the same data as halos. By introducing threads and thus using less MPI processes, less halo regions are needed, which translates to smaller memory requirements.
Using the MG command "stats", the program is able to provide information about its memory usage. In Tables 6.1, 6.2 and 6.3 below, the total memory usage for the three different problem sizes is shown, up to 128 PEs. The combination of 8 threads per MPI per process was chosen for the memory requirements investigation, as this version offers the most significant advantage for the mixed-mode code. The two other combinations, with 2 and 4 threads per MPI process, will also require less memory than the MPI-only implementation, but the difference will not be so apparent.

<table>
<thead>
<tr>
<th>Size: 100^3</th>
<th>PEs</th>
<th>MPI Memory (Gb)</th>
<th>Mixed-mode Memory (Gb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.992</td>
<td>0.992</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1.340</td>
<td>1.03</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1.910</td>
<td>1.07</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>3.140</td>
<td>1.16</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: MG memory usage for MPI-only and mixed-mode (size: 100^3)

<table>
<thead>
<tr>
<th>Size: 200^3</th>
<th>PEs</th>
<th>MPI Memory (Gb)</th>
<th>Mixed-mode Memory (Gb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.45</td>
<td>5.45</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>8.97</td>
<td>8.04</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>10.0</td>
<td>8.17</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>13.5</td>
<td>8.44</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: MG memory usage for MPI-only and mixed-mode (size: 200^3)

<table>
<thead>
<tr>
<th>Size: 300^3</th>
<th>PEs</th>
<th>MPI Memory (Gb)</th>
<th>Mixed-mode Memory (Gb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10.1</td>
<td>10.1</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>29.0</td>
<td>18.4</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>31.2</td>
<td>18.5</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>36.5</td>
<td>18.5</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3: MG memory usage for MPI-only and mixed-mode (size: 300^3)

The smaller memory needs of the mixed-mode implementation are obvious; for all cases, the mixed mode implementation requires less memory. This becomes more obvious for the largest problem size, where 128 PEs in mixed-mode require almost half the memory of their MPI-only equivalent. It should be noted that because the memory is reported in exponential form with only three significant digits, in some cases there may be some loss of information. For example, the mixed-mode memory for 300^3 cells (Table 6.3) appears to be the same for 64 and 128 PEs, (8 MPI processes × 8 threads and 16 MPI processes × 8 threads respectively).
Chapter 7

Conclusion

The present project involved examining an existing code, parallelized with MPI, and investigating the addition of OpenMP threads to it, in order to transform it to a mixed-mode implementation and assess any possible performance benefits that this might have.

Benchmarking and profiling the original code showed that communication affected performance at high numbers of MPI processes, taking up large percentages of the execution time (Section 4.4). This showed promise that the code might benefit from mixed-mode programming, by using less MPI processes and thus less communication, while executing the computationally intensive parts in parallel using threads in each MPI process.

The main computational loops of the code, which were parallelized with threads, are linked-list traversals. This allowed for two different implementations; one manual, which involved transforming the loops (Section 5.2), and one using OpenMP tasks (Section 5.3), a relatively new addition to OpenMP (OpenMP 3.0). The manual implementation achieved good loop scaling (Section 5.2.4), whereas the tasks version, while being more elegant, resulted in significantly bad performance (Section 5.3.1).

The manual OpenMP parallelization of the code was thus chosen to investigate the final mixed-mode performance of the code. Three combinations of threads per MPI processes were investigated, as discussed in Section 6.1. The benchmarks showed that for every problem size and number of PEs, there was a combination of threads per MPI process of the mixed-mode code that performed equally or better than the original code. More importantly, for the higher PE counts, where the original MPI-only implementation exhibits slow-down, the mixed-mode code has been shown to achieve better performance. Apart from performance, mixed-mode was shown to require significantly less memory over MPI-only (Section 6.2).
7.1 Problems Encountered

A significant part of the project involved acquiring a scientific code that neither the student nor the supervisor were familiar with and, after reading the documentation and examining the code, finding how to introduce mixed-mode programming to it. As mentioned before, the choice of program was based on a short discussion between the supervisor (Dr. Chris Maynard) and the creator of the code (Professor Sam Falle). While such work is a good challenge, it inevitably leads to certain problems.

First of all, during the project preparation process, the work required for the project was not defined. The general scope of the project was mixed-mode programming, but it was part of the preparation to find how threads could be introduced to the code. Initially, in the project description, the PARC library was targeted (without knowledge of its function or structure, but mainly because it was described as an emulation of shared-memory systems). After spending a good amount of time in analyzing the structure and function of this fairly complicated MPI communication library, it was realized that introducing threads to it was not possible. The direction of the project then changed, deciding to work with MG and implement a mixed-mode version of it. The analysis of the PARC library, while being challenging and interesting, was unfortunately too irrelevant to the scope of the project to include in the present report.

Porting the code to both systems used (Ness and HECToR) also proved to be quite difficult and time consuming. The installation process suggested in the documentation would not be very efficient for the development process, requiring continuous compilation of all libraries. Separate makefiles were written for the two machines, mainly with the help of Chris Maynard. However, compiling the program successfully on both machines required significant effort and special thanks go to Chris Maynard and the HECToR helpdesk for their help.

The most important problem encountered is probably the issue described in Section 5.4.1, regarding compatibility with the new version of gcc. The consequence of this problem is that no output can be read when running on HECToR; this is especially important because the tasks implementation cannot be tested effectively, other than the small tests described in 5.4.1. Unfortunately, this problem was not discovered until rather late into the project, because Ness was used for development and testing initially. As a result, it was decided that the relatively short dissertation period would not allow enough time to deal with this problem, especially because it is outside the project’s scope.
7.2 Future Work

Apart from trying to fix the problem described in Section 5.4.1, the code can provide some opportunities for further work. As described in Section 5.3.1, the OpenMP tasks version of the code gave very poor results, because of the large number of small tasks generated. A future implementation can involve grouping more iterations of the linked-list loops into each task. In this way, a smaller number of tasks will be created, possibly resulting in better scaling. This implementation will require more additions to the code, not having the elegance of the tasks code; however, it will still be less than the additions of the manual implementation.

Another possible direction, outside the scope of mixed-mode programming, could be to deal with PARC, the communications library. First of all, PARC uses blocking MPI sends and receives, which are not ideal candidates for good performance. The PARC code can thus be investigated in order to see if the blocking functions can be replaced with non-blocking equivalents and, if so, examine if any performance can be gained from such alteration.

Moreover, the parc_xdata communication function is similar to the MPI_Alltoallv function. Since the parc_xdata function is used extensively in the MG communication functions, its replacement with the MPI_Alltoallv function can be investigated and, if it is possible, the performance of the two can be compared.

Regarding the addition of OpenMP threads to the original code, additional work can be done in order to produce a more elegant code. For the scope of this project, the two different OpenMP versions (manual and tasks) were implemented as alternative "step" functions inside the mg.c source file. In order to choose any of the two threaded implementations, these functions have to be renamed and the code has to be recompiled. A more elegant approach would involve applying conditional compilation and letting the user define the desired version during compilation.
Appendix A

Benchmark Data

In this section, the data from the benchmarks that were performed throughout this project will be presented.

A.1 Initial MG Benchmarks

<table>
<thead>
<tr>
<th>MPI Processes</th>
<th>Execution Time (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>54.8</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>46.3</td>
<td>1.18</td>
</tr>
<tr>
<td>4</td>
<td>26.1</td>
<td>2.10</td>
</tr>
<tr>
<td>8</td>
<td>16.9</td>
<td>3.25</td>
</tr>
<tr>
<td>16</td>
<td>9.91</td>
<td>5.53</td>
</tr>
<tr>
<td>32</td>
<td>6.61</td>
<td>8.29</td>
</tr>
<tr>
<td>64</td>
<td>4.98</td>
<td>10.96</td>
</tr>
<tr>
<td>128</td>
<td>4.64</td>
<td>11.81</td>
</tr>
<tr>
<td>256</td>
<td>6.66</td>
<td>8.23</td>
</tr>
</tbody>
</table>

Table A.1: MG, initial benchmark (size: $100^3$)
### A.2 Find Maximum: Scaling Investigation

<table>
<thead>
<tr>
<th>Size</th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>8 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>100³</td>
<td>1.01</td>
<td>1.56</td>
<td>2.65</td>
<td>5.11</td>
</tr>
<tr>
<td>200³</td>
<td>8.76</td>
<td>12.0</td>
<td>22.0</td>
<td>44.6</td>
</tr>
</tbody>
</table>

Table A.4: Find maximum: execution times with single command inside critical region

<table>
<thead>
<tr>
<th>Size</th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>8 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>100³</td>
<td>1.04</td>
<td>1.55</td>
<td>2.6</td>
<td>5.02</td>
</tr>
<tr>
<td>200³</td>
<td>8.64</td>
<td>12.2</td>
<td>22.1</td>
<td>45.5</td>
</tr>
</tbody>
</table>

Table A.5: Find maximum: execution times with loop inside critical region
Table A.6: Find maximum: execution times with temporary array, no critical region

<table>
<thead>
<tr>
<th>Size</th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>8 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>100³</td>
<td>0.92</td>
<td>0.63</td>
<td>0.55</td>
<td>0.56</td>
</tr>
<tr>
<td>200³</td>
<td>8.36</td>
<td>5.75</td>
<td>4.82</td>
<td>3.65</td>
</tr>
</tbody>
</table>

A.3 Loop Scaling With OpenMP

Table A.7: Loop execution times with manual OpenMP implementation (size: 100³ cells)

<table>
<thead>
<tr>
<th>Loop</th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>8 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>First cptr</td>
<td>0.94</td>
<td>0.66</td>
<td>0.56</td>
<td>0.52</td>
</tr>
<tr>
<td>jptr 1</td>
<td>1.23</td>
<td>0.59</td>
<td>0.37</td>
<td>0.29</td>
</tr>
<tr>
<td>cptr 1</td>
<td>0.42</td>
<td>0.22</td>
<td>0.13</td>
<td>0.11</td>
</tr>
<tr>
<td>cptr 2.1</td>
<td>0.49</td>
<td>0.25</td>
<td>0.16</td>
<td>0.17</td>
</tr>
<tr>
<td>cptr 2.2</td>
<td>2.73</td>
<td>1.33</td>
<td>0.76</td>
<td>0.48</td>
</tr>
<tr>
<td>jptr 2</td>
<td>3.44</td>
<td>1.7</td>
<td>0.97</td>
<td>0.60</td>
</tr>
<tr>
<td>cptr 2.3</td>
<td>0.43</td>
<td>0.21</td>
<td>0.12</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Table A.8: Loop execution times with manual OpenMP implementation (size: 200³ cells)

<table>
<thead>
<tr>
<th>Loop</th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>8 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>First cptr</td>
<td>8.16</td>
<td>5.79</td>
<td>3.83</td>
<td>4.20</td>
</tr>
<tr>
<td>jptr 1</td>
<td>9.06</td>
<td>4.52</td>
<td>2.53</td>
<td>1.58</td>
</tr>
<tr>
<td>cptr 1</td>
<td>2.63</td>
<td>1.33</td>
<td>0.83</td>
<td>0.80</td>
</tr>
<tr>
<td>cptr 2.1</td>
<td>4.39</td>
<td>2.23</td>
<td>1.28</td>
<td>1.40</td>
</tr>
<tr>
<td>cptr 2.2</td>
<td>19.46</td>
<td>10.22</td>
<td>5.43</td>
<td>4.21</td>
</tr>
<tr>
<td>jptr 2</td>
<td>29.0</td>
<td>14.77</td>
<td>7.97</td>
<td>4.13</td>
</tr>
<tr>
<td>cptr 2.3</td>
<td>2.65</td>
<td>1.33</td>
<td>0.83</td>
<td>0.80</td>
</tr>
</tbody>
</table>
## A.4 Loop Scaling With OpenMP tasks

<table>
<thead>
<tr>
<th>Loop</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 Thread</td>
</tr>
<tr>
<td>First cptr</td>
<td>0.46</td>
</tr>
<tr>
<td>jptr 1</td>
<td>1.90</td>
</tr>
<tr>
<td>cptr 1</td>
<td>0.21</td>
</tr>
<tr>
<td>cptr 2.1</td>
<td>0.31</td>
</tr>
<tr>
<td>cptr 2.2</td>
<td>1.09</td>
</tr>
<tr>
<td>jptr 2</td>
<td>1.89</td>
</tr>
<tr>
<td>cptr 2.3</td>
<td>0.23</td>
</tr>
</tbody>
</table>

Table A.9: Loop execution times with the OpenMP tasks implementation (size: $100^3$ cells)

<table>
<thead>
<tr>
<th>Loop</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 Thread</td>
</tr>
<tr>
<td>First cptr</td>
<td>3.71</td>
</tr>
<tr>
<td>jptr 1</td>
<td>5.28</td>
</tr>
<tr>
<td>cptr 1</td>
<td>5.34</td>
</tr>
<tr>
<td>cptr 2.1</td>
<td>2.50</td>
</tr>
<tr>
<td>cptr 2.2</td>
<td>8.72</td>
</tr>
<tr>
<td>jptr 2</td>
<td>15.2</td>
</tr>
<tr>
<td>cptr 2.3</td>
<td>1.94</td>
</tr>
</tbody>
</table>

Table A.10: Loop execution times with the OpenMP tasks implementation (size: $200^3$ cells)

## A.5 Mixed-Mode Benchmarks

### A.5.1 2 Threads Per MPI Process

<table>
<thead>
<tr>
<th>MPI Processes</th>
<th>OpenMP Threads</th>
<th>Total PEs</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>2</td>
<td>32</td>
<td>6.76</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>64</td>
<td>4.77</td>
</tr>
<tr>
<td>64</td>
<td>2</td>
<td>128</td>
<td>3.65</td>
</tr>
<tr>
<td>128</td>
<td>2</td>
<td>256</td>
<td>3.86</td>
</tr>
</tbody>
</table>

Table A.11: MG Mixed-mode benchmarks: Execution times with 2 threads per MPI process (size: $100^3$)
<table>
<thead>
<tr>
<th>MPI Processes</th>
<th>OpenMP Threads</th>
<th>Total PEs</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>2</td>
<td>64</td>
<td>28.0</td>
</tr>
<tr>
<td>64</td>
<td>2</td>
<td>128</td>
<td>19.6</td>
</tr>
<tr>
<td>128</td>
<td>2</td>
<td>256</td>
<td>15.0</td>
</tr>
<tr>
<td>256</td>
<td>2</td>
<td>512</td>
<td>14.9</td>
</tr>
</tbody>
</table>

Table A.12: MG Mixed-mode benchmarks: Execution times with 2 threads per MPI process (size: $200^3$)

<table>
<thead>
<tr>
<th>MPI Processes</th>
<th>OpenMP Threads</th>
<th>Total PEs</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>2</td>
<td>128</td>
<td>52.5</td>
</tr>
<tr>
<td>128</td>
<td>2</td>
<td>256</td>
<td>39.7</td>
</tr>
<tr>
<td>256</td>
<td>2</td>
<td>512</td>
<td>34.5</td>
</tr>
<tr>
<td>512</td>
<td>2</td>
<td>1024</td>
<td>36.7</td>
</tr>
</tbody>
</table>

Table A.13: MG Mixed-mode benchmarks: Execution times with 2 threads per MPI process (size: $300^3$)

### A.5.2 4 Threads Per MPI Process

<table>
<thead>
<tr>
<th>MPI Processes</th>
<th>OpenMP Threads</th>
<th>Total PEs</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
<td>32</td>
<td>8.16</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>64</td>
<td>5.26</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>128</td>
<td>3.70</td>
</tr>
<tr>
<td>64</td>
<td>4</td>
<td>256</td>
<td>3.05</td>
</tr>
</tbody>
</table>

Table A.14: MG Mixed-mode benchmarks: Execution times with 4 threads per MPI process (size: $100^3$)

<table>
<thead>
<tr>
<th>MPI Processes</th>
<th>OpenMP Threads</th>
<th>Total PEs</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>4</td>
<td>64</td>
<td>34.5</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>128</td>
<td>21.8</td>
</tr>
<tr>
<td>64</td>
<td>4</td>
<td>256</td>
<td>15.5</td>
</tr>
<tr>
<td>128</td>
<td>4</td>
<td>512</td>
<td>13.1</td>
</tr>
</tbody>
</table>

Table A.15: MG Mixed-mode benchmarks: Execution times with 4 threads per MPI process (size: $200^3$)
### MPI Processes

<table>
<thead>
<tr>
<th>MPI Processes</th>
<th>OpenMP Threads</th>
<th>Total PEs</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>4</td>
<td>128</td>
<td>62.0</td>
</tr>
<tr>
<td>64</td>
<td>4</td>
<td>256</td>
<td>41.5</td>
</tr>
<tr>
<td>128</td>
<td>4</td>
<td>512</td>
<td>32.8</td>
</tr>
<tr>
<td>256</td>
<td>4</td>
<td>1024</td>
<td>29.0</td>
</tr>
</tbody>
</table>

Table A.16: MG Mixed-mode benchmarks: Execution times with 4 threads per MPI process (size: $300^3$)

### 8 Threads Per MPI Process

<table>
<thead>
<tr>
<th>MPI Processes</th>
<th>OpenMP Threads</th>
<th>Total PEs</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>32</td>
<td>11.8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>64</td>
<td>6.91</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>128</td>
<td>4.58</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>256</td>
<td>3.37</td>
</tr>
</tbody>
</table>

Table A.17: MG Mixed-mode benchmarks: Execution times with 8 threads per MPI process (size: $100^3$)

<table>
<thead>
<tr>
<th>MPI Processes</th>
<th>OpenMP Threads</th>
<th>Total PEs</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>64</td>
<td>50.9</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>128</td>
<td>28.5</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>256</td>
<td>18.5</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>512</td>
<td>13.9</td>
</tr>
</tbody>
</table>

Table A.18: MG Mixed-mode benchmarks: Execution times with 8 threads per MPI process (size: $200^3$)

<table>
<thead>
<tr>
<th>MPI Processes</th>
<th>OpenMP Threads</th>
<th>Total PEs</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>8</td>
<td>128</td>
<td>91.2</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>256</td>
<td>54.1</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>512</td>
<td>36.7</td>
</tr>
<tr>
<td>128</td>
<td>8</td>
<td>1024</td>
<td>30.3</td>
</tr>
</tbody>
</table>

Table A.19: MG Mixed-mode benchmarks: Execution times with 8 threads per MPI process (size: $300^3$)
Bibliography


[13] Sam Falle, Sven van Loo, Jack Giddings. Some Applications of AMR to Astrophysical MHD. University of Leeds, Mantis Numerics Ltd. (Slides provided by Professor Sam Falle)

[14] Mantis Numerics, MG Guide. MG documentation. Provided along with the MG code by Professor Sam Falle, Applied Mathematics Department, University of Leeds.
