University of Glasgow | School of Computing Science

Extending the Glasgow Parallel Reduction Machine (GPRM) with MPI for Use in Distributed Memory Architectures

Georgios Goulos

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

A dissertation presented in part fulfilment of the requirements of the Degree of Master of Science at The University of Glasgow

September 6th, 2015

Abstract

This project aims to provide an extension to a version of the Glasgow Parallel Reduction Machine (GPRM) that is used with the Glasgow Model Coupling Framework (GMCF). GPRM is a framework used for parallel task composition and currently it can be used on shared-memory architectures. GMCF is a framework that aims to provide an easier approach to model coupling. The result of this project can be integrated in GPRM, which will allow it to run on distributed-memory architectures using the Message Passing Interface (MPI) for passing messages between different nodes. This will allow the GMCF to utilise more computing power for the computationally intensive operations it handles.

The project focuses on the development of "gateways" between the nodes on the distributed-memory architecture. These gateways are called bridges and allow threads on different nodes communicate with one another by passing messages. This dissertation provides a description of the existing software, design of these bridges and how they were implemented.

A set of experiments was performed in order to test the operation of the bridges and their efficiency. This dissertation explains the rationale behind these experiments and tries to interpret their results.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: ____________________ Signature: ____________________

Acknowledgements

I would like to express my gratitude to the supervisor of this project, Dr. Wim Vanderbauwhede, for his enormous help throughout the course of this project.

To Olga Katsarou, for the immense support that she showed during the last few months, I am thoroughly grateful.

Last but not least, I would like to dedicate this dissertation to my parents for all they have done for me throughout my life.

Contents

1 Introduction **5**

1.1 Statement of Problem 5

1.2 Motivation 6

1.3 Outline 7

2 System Design **8**

2.1 Background Survey 8

2.2 Rationale Behind Bridges 9

2.3 Requirements 9

2.4 From Shared Memory to a Hybrid of Shared and Distributed Memory Systems . . 10

2.5 System Diagram 11

2.6 Packet-passing Procedure 12

2.7 Optimised Topology 13

3 System Implementation **15**

3.1 Changes to the Existing GMCF Code 15

3.1.1 The System Class 15

3.1.2 The Transceiver Class 18

3.2 The Bridge Class 19

3.3 Sending Packets 20

3.3.1 Sending Packets Using New Threads – Non-blocking Approach 20

3.3.2 The Need for Two Sending Methods 21

3.4 Receiving Packets 22

3.5 Compiling, Executing and Testing 24

3.6 Alternative Implementation Approaches Explored 25

4 Evaluation **26**

4.1 Qualitative Evaluation - Integration with GMCF 26

4.2 Sending and Receiving GMCF Packets 27

4.3 Timing the Transmission of Packets 28

4.4 Timing the transmission of single MPI message for varying number of bridges . . . 29

4.5 Memory Leaks 30

5 Conclusion **32**

5.1 Discussion 32

5.2 Further Work 32

A First appendix **34**

A.1 Simple Packet Transfer 34

A.2 Results of Performance Experiments 36

Chapter 1

Introduction

Parallel computing is becoming increasingly popular because the measures for satisfying the need for more computation power have moved from increasing the processor clock speed to using multi-core processors or multiple computing systems together towards a common goal due to limitations of the former method. Therefore, the need for software that allows users to easily write parallel programs has risen.

This project aims to provide an extension to the Glasgow Parallel Reduction machine (GPRM), a framework for parallel task composition. The goal of GPRM is to allow its users write parallel programs for solving computationally-intensive tasks on many-core architectures using libraries developed for single-core systems [13].

The development of this project focused on extending a version of GPRM that was customised for use with the Glasgow Model Coupling Framework (GMCF). The main idea (and long-term goal) behind GMCF is to provide a framework that allows users define a scenario for model coupling (i.e. having multiple models being executed in such a way that the output of the execution of one is used as input for another) in natural language, while the framework takes care of everything else [15]. GMCF uses GPRM as an underlying framework for communication between the threads executing the models.

1.1 Statement of Problem

Currently, GPRM can be used for parallel computations on shared-memory architectures [13]. The goal of this project is to extend GPRM so that it can be used on distributed-memory architectures as well. In order to achieve this, the Message Passing Interface (MPI) shall be used. MPI is a standard for message passing that allows processes in a distributed-memory environment communicate by passing messages to one another.

GMCF uses tiles (the workers of the system) that perform the computations for the models they were assigned and they communicate with other tiles through the transmission of packets. These packets contain data related to the execution of the models, as well as information such as the ID of the tile that sent the message and the size of the packet. Currently, GPRM allows tiles to transmit packets utilising the shared-memory architecture of the system it is running on. Using MPI, tiles should be able to send such packets to any tile across a distributed-memory system.

The result of this project should provide a logical gateway between nodes that allows tiles on a node communicate with others on a different node through this gateway, which we call a **bridge**. However, tiles on the same node should still take advantage of the shared-memory architecture for their communication.

There are many types of packets that can be transmitted between tiles. The type of a packet might indicate that it contains a pointer to an array of data. In a shared-memory architecture, the receiver of such a packet can simply access the memory space on which that data is stored. In a distributed-memory system, however, that data has to be copied to the memory space of the receiver. This is one of the main challenges of this project.

Since the result of this project should be integrated in GMCF, which can be used for computationally-intensive operations such as weather simulations, the communication between tiles should be as fast as possible. Therefore, development of this project should focus on providing a working implementation that shows a good performance.

1.2 Motivation

The idea behind this project is that GMCF and the underlying GPRM framework has the potential to be used in a distributed-memory architecture [15]. GMCF uses a message-passing system for communication, which makes it possible for GMCF to utilise a distributed-memory architecture for model coupling. An opportunity to work on this update of the original code was very appealing to me, due to the challenges of working on an existing parallel framework that has been under development for many years would pose. Developing code which will be executed by multiple systems that communicate with one another is a very exciting (albeit difficult or at least challenging) concept, and the combination of shared and distributed-memory architectures makes the project more challenging and intriguing.

Through this project I am certain that my programming skills and my understanding of parallel computing, shared and distributed-memory architectures will vastly improve. I was introduced to the concept of parallel computing during my undergraduate studies and was quickly enthralled by it and the potential of improvements to performance of systems that its proper use can lead to. However, I did not have the chance to work on such a large project that involves parallelism in the past, which made this task quite intriguing. This project was also a great opportunity for me to become more familiar with C++, a programming language that is widely used in the industry and academia, by studying the existing GPRM code and developing my own.

1.3 Outline

The remainder of this document is organised as follows: Chapter 2 provides the context of the problem and an explanation why MPI was selected to tackle this problem, as well as insight on the requirements that the project should satisfy, information on the architecture of the system and a potential optimisation that MPI implementations might offer based on the logical topology of nodes in the system. Chapter 3 contains a detailed description of the implementation of the system In Chapter 4 tests that were conducted and their results are shown. Chapter 5 summarises the conclusions of this project and provides suggestions for further work.

Chapter 2

System Design

2.1 Background Survey

The main goal of this project is to extend GPRM so that it can be used in a hybrid of shared-memory and distributed-memory architectures. There are many technologies that aim to provide an easy way for users to fully utilise such hybrid architectures.

One notable example is the Partitioned Global Address Space (PGAS) programming model [3]. According to PGAS, the processes (or threads) in SPMD programming can create a global address space. This space is logically partitioned with each process storing a portion of this space locally. The processes are able to take advantage of locality and operate on data that are stored in the partition of the memory space that they have affinity with. A variation of PGAS, Asynchronous Partitioned Global Address Space (APGAS) allows nodes to perform and create tasks asynchronously. Notable languages that implement this programming model are X10, Fortress and Chapel [4].

Another solution to the problem that this project addresses could be implemented using Glasgow Parallel Haskell (GpH), an extension to the Haskell programming language. The Glasgow Haskell Compiler, or GHC, provides concurrency and pure parallelism on a shared-memory architecture [7]. The addition of GpH allows Haskell to be used for parallelism on computer clusters as well [1].

Even though APGAS and GpH would allow GMCF to take advantage of the combination of a distributed-memory and a shared-memory system, that would require a large part of GPRM to be rewritten. GPRM has been under development for many years and its developers have taken many precautions to ensure that proper synchronisation techniques and inter-thread communication are performed. Therefore, an ideal approach to the problem would solve the issue of fully utilising a hybrid of distributed-memory and shared-memory architectures while keeping the existing code. Using an MPI implementation allowed for extending GMCF with the largest part of the new code residing in new files due to an effort to keep changes to the existing framework to a minimum.

2.2 Rationale Behind Bridges

A tile that wants to transmit packets to other tiles on the same node does so by finding the target tile based on its unique identifier and inserting the packet in a FIFO of that tile. A tile could easily invoke an MPI routine in order to achieve this for a tile on a different node. However, giving to each such tile the ability to constantly be able send and receive packets would not be ideal in the case of GPRM.

The reason for that is that MPI currently does not provide adequate support for thread awareness and utilisation in processes. Even though a transition towards this state can be observed (e.g. the introduction of matching routines in MPI-3, such as *MPI_Mprobe()* and *MPI_Mrecv() [9, p. 67]*, which solve the problem of ensuring that a thread will receive the message that was previously probed), the current versions of the standard would make the implementation of this scenario quite difficult. In addition to that, providing a thread-safe environment for MPI can prove to be quite difficult and might cause a large overhead if a large number of threads tries to invoke MPI routines simultaneously [6].

Another reason why bridges were used is that they aim to implement the gateways used for inter-node communication in the approach proposed by Vanderbauwhede for GMCF [15]. This approach is shown in Figure 2.1.

2.3 Requirements

Based on the study of the GMCF code and discussions with my supervisor and developer of GMCF some requirements were defined. These were taken into account during the development of the project and are stated below:

A tile on a node should be able to send/receive messages to/from any tile on any other node. The system should, in principle, allow any tile on a node communicate with any tile on any other node. For communication between tiles on the same node utilisation of the shared-memory architecture and the existing code should take place.

Bridges should be transparent to the user. The user should not interact with the bridges directly. Besides providing input that is related to the number of processes and using commands for compilation and execution of the code that utilises MPI the user should not be aware of the existence of bridges.

Tiles that send packets to other nodes should not block their operations. Tiles should just send the outgoing packet to a bridge and carry on with their operations, while the bridge handles the transmission.

Multiple bridges can listen for incoming messages at a time. Each bridge should try to receive a different incoming message to avoid having two or more retrieving the same incoming message and sending it to the same tile

The use of bridges and MPI should be optional. GMCF should still be able to run the way it did before the integration of bridges if the user decides so.

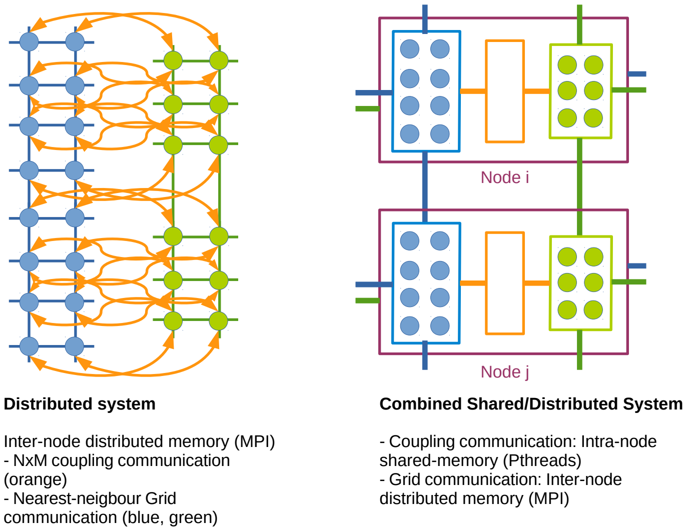

Node i
Node j
Distributed system
Inter-node distributed memory (MPI)
- NxM coupling communication (orange)
- Nearest-neigbour Grid communication (blue, green)
Combined Shared/Distributed System
- Coupling communication: Intra-node shared-memory (Pthreads)
- Grid communication: Inter-node distributed memory (MPI)

Figure 2.1: Left: current approach to model coupling. Right: suggested approach. Figure taken from [15]

The code should be maintainable. The new code should be easy to modify for achieving improvements to performance and utilisation. GMCF is still under development, so it should be easy for developers to update the code that was written as part of this project.

MPI should be initialised and finalised properly. This is very important for every program that utilises MPI. The code should contain exactly one call to an initialisation and one call to a finalisation routine.

The code should be efficient. The use of bridges should lead to a good performance for the overall system. If bridges lead to a large overhead their use would not be recommended.

Bridges and all the modifications to the existing GMCF code should not introduce memory leaks. This would make the new code reliable and robust.

2.4 From Shared Memory to a Hybrid of Shared and Distributed Memory Systems

According to Vanderbauwhede [15], the modern approaches to model coupling have been developed for use on single-core computing systems in a distributed-memory environment. As stated before, GMCF will eventually provide an approach to model coupling that takes advantage of shared-memory and distributed-memory architectures combined.

The approach proposed in [15] considers multiple multi-core computers that utilise each core as a worker. Communication between cores on the same computer is achieved using Pthreads (POSIX threads), whereas cores on different computers communicate by passing messages through MPI routines . The GPRM version that this project was developed for had already implemented the pthread communication. Therefore, this project focused on the use of MPI for grid communication.

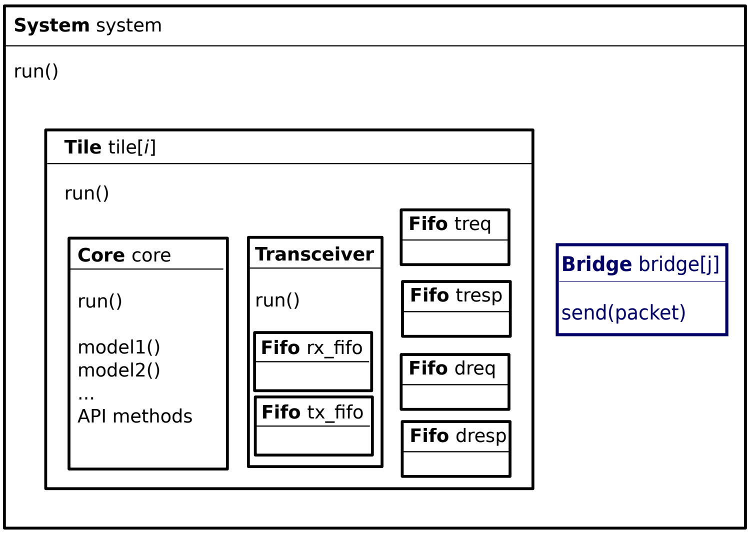


Figure 2.2: Software architecture. Original figure taken from [15]

2.5 System Diagram

Figure 2.2 presents the software architecture of GMCF integrated with the result of this project. During the execution of GMCF a System instance is created for each MPI process. Each such instance creates a number of Tile instances that act as the workers of the system. For each tile a thread is created, which is assigned tasks (or models in the case of GMCF) that it must perform. Typically, each tile should be executed on a different CPU core on the node. Each Tile instance contains 4 FIFOs used for storing GMCF packets according to their type, as well as a Transceiver instance, which provides two FIFOs to the Tile: one for storing packets to be transmitted to other tiles to the system and one for storing packets that are to be received by the current tile.

After integrating the result of this project to GMCF, the System instance shall also create a number of Bridge instances. These bridges will be used for communication purposes between tiles that are executed on different nodes. In order for a tile to send a packet to a different node, it will pass that packet to the System instance, which will then select the most appropriate bridge to make the transfer (this could, for example, depend on whether the bridge is already busy with another packet). Each bridge creates a thread that constantly listens for incoming packets, which means that no API function was required for this operation. The default number of bridges to be created on each node is 3 but this should not indicate that this is the ideal number of bridges for each node. This number was arbitrarily selected. Users can change the number of bridges without having to do any further modifications to the system.

The GMCF code comprises many classes and routines. However, achieving the goal of this project required the use of only a few of those. In fact, the main classes used are depicted in Figure 2.2. In addition to those, some user-defined types, macros, variables, functions and constants from the original GMCF code were used. The files that included these elements in GMCF were modified for simplifying the development process and are included in the submitted code. Documentation of the code states which parts were written by the GMCF developers.

The Bridge class was tested using the aforementioned files and classes but stripped down to the absolutely necessary elements for testing the implementation. It is also worth mentioning that

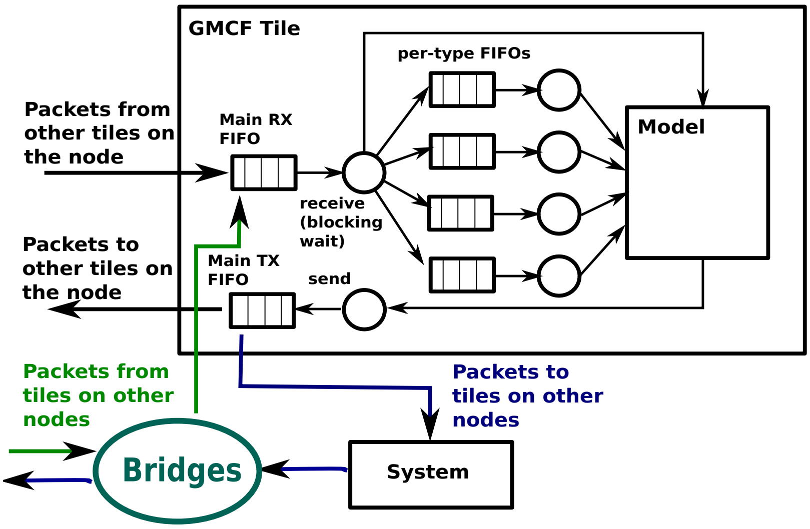

GMCF Tile
per-type FIFOs
Packets from other tiles on the node
Main RX FIFO
Model
receive (blocking wait)
Packets to other tiles on the node
Main TX FIFO
send
Packets from tiles on other nodes
Packets to tiles on other nodes
Bridges
System

Figure 2.3: Run-time architecture of the system. The original figure was taken from [15]

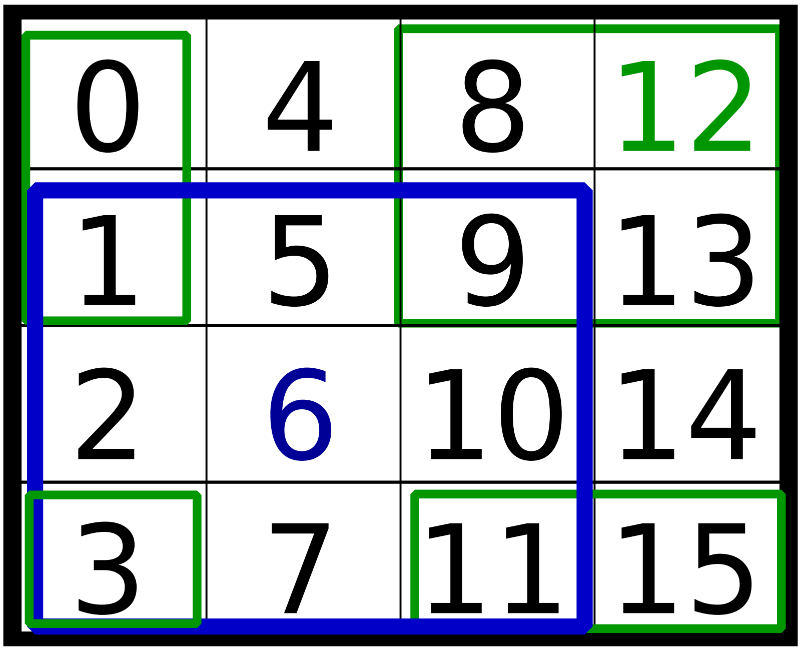
many approaches to achieving certain goals were tested and therefore some of the elements that remain in the final versions of the files might not actually be used by the implementation.

2.6 Packet-passing Procedure

Bridges allow tiles in different MPI processes to communicate with one another through MPI routines. Tiles store outgoing GMCF packets in the TX FIFO provided by the Transceiver instance. When the transceiver detects packets in that FIFO, it checks whether the tile to receive the packet is on the same node. If that is true, then it can simply access the RX FIFO of that tile and store the packet there, since they both reside on a shared-memory environment. Otherwise, the packet is sent to the System instance, which is responsible for deciding which bridge shall be used for the send operation. That bridge receives the packet and transmits it to the corresponding node. The bridge on the receiving side receives the message, detects the target tile by checking a field of the packet header and stores the GMCF packet in its RX FIFO[1]. Figure 2.3 shows the runtime architecture of the system. The bridges should be only used when a tile wants to transmit GMCF packets to a tile on a different node. If the tiles are on the same node there is no reason to involve a bridge in the operation since that would only cause unnecessary overhead.

There are many types of packets the GMCF uses. The packets that a tile receives are stored in FIFOs according to their type. This procedure and how packets are stored in the TX FIFO are outside the scope of this project.

[1] GMCF packets comprise two parts: a header and a payload. Both of those are vectors that contain elements of type Word, or unsigned 64-bit integers. One of the elements in the header contains various information regarding the packet, such as the ID of the tile that sent the packet and the ID of the receiver



0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Figure 2.4: An example of logical topology of the processes on the system. The black numbers in the blue square represent the ranks of the neighbours of process 6. The black numbers in the green rectangles represent the ranks of the neighbours of process 12.

2.7 Optimised Topology

MPI defines routines for creating a virtual topology of the processes[2] in a process group [3]. A virtual topology can provide a conventional way of naming and ordering processes. For example, distributed systems that implement a logical ring or a torus architecture could greatly benefit from such a topology.

When a MPI initialisation routine is invoked, each process is assigned a unique number from 0 to $number\_of\_processes - 1$, which can be used to identify each process. This number is known as the rank of a process. During the creation of a cartesian virtual topology, which takes place after the MPI initialisation in a program, the user can decide on the reordering of the numbering of the process ”possibly so as to choose a good embedding of the virtual topology onto the physical machine” [9, p. 292]. Otherwise, the processes will keep the original rank they were assigned during initialisation of MPI.

This project uses an MPI routine to create a logical torus architecture with each node representing a process. MPI defines a routine that a process can use in order to identify its rank and detect its surrounding neighbours, if a logical topology was created. This could help the processes with performing stencil or other operations that involve passing messages to neighbours.

Figure 2.4 shows the neighbours of a process in the system if the use of this topology is requested. The process of rank 6 will detect and consider as neighbours the ranks in the blue square (namely, processes 1, 2, 3, 5, 7, 9, 10, 11). Rank 12 will have the same number of neighbours because the

[2] In this dissertation, the terms (MPI) process and node are used interchangeably.

[3] A process group is an ordered set of MPI processes.

system considers the topology toroidal in the sense that a process on the edge of a dimension of the table is considered adjacent to the processes on the other edge on that dimension. Therefore, its neighbours are the processes of rank 0, 1, 3, 8, 9, 11, 13, 15. It is worth noting that the ranks of processes are assigned column-first on the topology by the aforementioned MPI routine.

The MPI standard does not guarantee a logical topology of processes in a group to resemble the physical topology of the hardware on which the software is executed since the standard does not cover this sort of operations. It is up to the developers of an MPI implementation to decide whether an option for this optimisation will be provided to users [4].

[4] It is worth mentioning that I tried searching for implementations that actually provide this optimisation but I was not able to find any proof. However, *MPI_Cart_create()*, the routine used for creating the logical topology, can still be used as a way of creating a logical torus and detecting the neighbours of a node easier. This could prove to be very helpful for stencil operations that require a GMCF node sending packets to logical neighbours.

Chapter 3

System Implementation

This chapter provides a description of the implementation and the operations that take place during the execution of the submitted program. However, it does not contain a description for every variable or function written and used during execution. There is detailed documentation of all the new code in the submitted C++ files.

3.1 Changes to the Existing GMCF Code

This section provides a description of the modifications performed in files of the original GMCF code that were used and modified while developing this project. One of the requirements of developing this project was that its use in GMCF should be optional. In order to satisfy this requirement, the macro BRIDGE was used. If the BRIDGE macro is defined during the compilation stage, the compiler will take into account the Bridge class and the modifications performed in the original code using preprocessor directives.

3.1.1 The System Class

In GMCF an instance of the System class is created. This instance is responsible for handling the creation of the Tile instances (workers of the framework) among other operations that are outside the scope of this project. After the integration with this project, GMCF is expected to create a System instance on every node running the framework. Therefore, it was deemed a good idea to allow this instance handle the initialisation and finalisation of MPI.

The original System class has two constructors: one that receives a TaskDescList argument and one that receives a string. These arguments are not required for testing the use of bridges and therefore they were not used in the System class that was created based on the existing GMCF code. However, the new constructor can easily be modified to receive these arguments, as well as perform the proper initialisations for member variables that were ignored.

The new constructor receives two integers that indicate the dimensions of the logical grid topology of the MPI processes. These values will be used the creation of a process table that represents the grid topology takes place. The constructor then proceeds to invoke the *MPI_Init_thread()* routine.

In order to use MPI routines for inter-process communication, either *MPI_Init()* or *MPI_Init_thread()* must initially be called by the program [9, p. 355]. These routines make initialisations to the message-passing layer that are required by most MPI routines[12]. This project uses the *MPI_Init_thread()* routine, which allows users to define the desired level of thread support. The levels of thread support defined by MPI are the following:

MPI_THREAD_SINGLE Used when only a single thread is expected to be executed

MPI_THREAD_FUNNELED Used when multiple threads are expected to be created and executed in a process but only the thread that called *MPI_Init()* or *MPI_Init_thread()* is expected to make MPI calls

MPI_THREAD_SERIALIZED Used when multiple threads are expected to be created and executed in a process and all or some of them are expected to make MPI calls. The MPI calls should be executed in a serialised manner (i.e. one such call at a time)

MPI_THREAD_MULTIPLE Used when multiple threads are expected to be created and executed in a process and all or some of them are expected to make MPI calls. MPI calls by multiple threads can take place at the same time

The default level provided by *MPI_Init()* is *MPI_THREAD_SINGLE*. For this project, however, this was considered unsuitable since the plan was to create multiple bridges for each MPI process and having each of them create threads that will be able to use MPI routines in order to send and receive messages on the same node at the same time. The *MPI_THREAD_MULTIPLE* level provides the ideal level of thread support for the GPRM framework.

Besides allowing users to define the requested level of thread support, *MPI_Init_thread()* provides as an output parameter the provided level. It is possible for a user to request a thread support level which the MPI implementation might not be able to provide. In this scenario the MPI implementation will try to use a lower thread support level. Therefore, checking the actual thread support level that will be used will ensure that the desired level of thread support is available. For example, in order for OpenMPI to be able to provide the *MPI_THREAD_MULTIPLE* level it must first be configured to do so [11]. If that level is requested but it is not available, then OpenMPI will try to provide the next highest level, which is *MPI_THREAD_SERIALIZED* [1].

After initialisation of MPI the program decides on the communicator which will be used for the communication between the MPI processes. Communicators in MPI provide a way for processes in a process group to communicate with one another (point-to-point communicator) or all together (collective communication) [2] [9, p. 227]. MPI implementations provide a predefined communicator named *MPI_COMM_WORLD* which covers all of the MPI processes that were created during

[1] Even though the OpenMPI implementation provides the *MPI_THREAD_MULTIPLE* level, according to the documentation it has only been ”lightly tested” [11] and, therefore, its use is not recommended. If users decide to use *MPI_THREAD_SERIALIZED*, then modification to the existing code should take place in order to ensure that no more than one MPI routine will be executed at the same time. However, this is not recommended since it would lead to a severe performance decrease for the bridges developed in this project.

[2] Actually, there are two types of communicators in MPI: intra-communicators and inter-communicators. The former is used for communication between processes in a process group, while the latter is used when communication between processes in different process groups is required. Since the latter is not used at all in this dissertation, every mention of a communicator refers to an intra-communicator, unless stated otherwise

the execution of a program. However, new communicators, which might cover all or a subset of the MPI processes, can be created and used. The communicator used by the bridges depends on whether the *MPI_TOPOLOGY_OPT* macro was defined during compilation or not. This macro is used for creating a toroidal logical topology, as explained in 2.7. If this macro is not defined then *MPI_COMM_WORLD* is used instead. In both cases, all MPI processes are covered by a single communicator and therefore each process can communicate with any other process.

Then, a process table is created. This table represents the logical topology of the processes running on the system. The System class also provides a member function for printing this table. It can be used for making it easier for the user to detect logically adjacent nodes.

The constructor, then, proceeds to create the Tile and Bridge instances. This is achieved just like in the constructor of the original GMCF code, using a loop that assigns a number to a *node_id* variable and then maps *node_id* to a pointer to a newly created Tile instance. This makes the *node_id* of each tile unique. The original System constructor uses integer numbers from 1 up to (including) the value of the *NSERVICES* constant, which is defined in the file *System.h*. However, it was decided that when the use of bridges is requested, each process should use a different range of numbers for the *node_id* values of the tiles. For example, instead of each process assigning to the Tile instances values from 1 to 64, the process whose rank is 1 should use the values 1-64, rank 2 should use 65-128, etc. This way, each tile will have a unique *node_id* and it will be easier for a GMCF packet to be transmitted to the right receiver [3] , since a simple division of *node_id* with *NSERVICES* will let a bridge detect which MPI process contains the corresponding tile. The constructor then proceeds to create *NBRIDGES* bridges and store pointers to them in a vector named *bridge_list*. *NBRIDGES* can be defined by the user during compilation. Otherwise it is assigned the value 3 in *Types.h*.

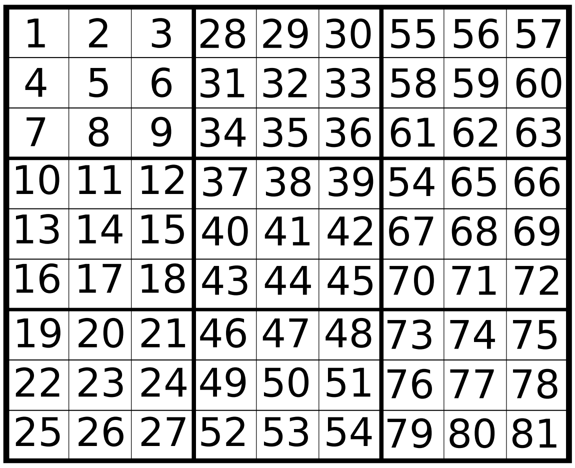
Finally, the constructor initialises two spinlocks. *bridge_selector_lock* is used in member functions of the System class for properly incrementing the value of a variable that indicates the index of the next bridge in *bridge_list* to be used. The use of this lock was deemed necessary because there might be multiple tile threads that will try to update the variable in question at a time. *killed_threads_lock* is used to ensure that the counter indicating the number of terminated receiving threads will be updated properly. This lock is used in the only function that updates the value of that counter and the function is called by receiving threads, which is the reason why a lock was used.

The first action that the bridge-related part of the destructor does is set the boolean variable *active* to *false*. The receiving threads created by Bridge instances check the value of this variable periodically to detect when the destructor of System has been invoked. Receiving threads are constantly using MPI routines for detecting and receiving incoming MPI messages. Since there should be no use of such routines after a call to *MPI_Finalize()* the receiving threads should first be terminated [4].

After setting the value of *active* to *false*, the destructor waits for notifications from all receiving threads that they are being terminated. Since each bridge creates a single receiving thread, the destructor waits until a number of threads equal to the number of bridges in the *bridge_list* vector

[3] Another idea was to have each MPI process use the range *1-NSERVICES* for assigning values to *node_id*. In order to define on which process the sender and receiver tiles reside, the ranks of their processes were added as fields in the header of the GMCF packet. That way a bridge could easily identify the target process of the GMCF packet. However, even though this idea was implemented, it was later discarded because it required further updates to the original code (for example, modifications to all the blocks of code that created new GMCF packets were necessary since the new fields of the header had to be assigned the proper values)

[4] It should be noted that such a constraint has not been considered for sending threads because a call to the System destructor should only be made after all operations of the framework have been completed, i.e. no more packets are being sent between tiles and all sent packets have been retrieved and used



1	2	3	28	29	30	55	56	57
4	5	6	31	32	33	58	59	60
7	8	9	34	35	36	61	62	63
10	11	12	37	38	39	54	65	66
13	14	15	40	41	42	67	68	69
16	17	18	43	44	45	70	71	72
19	20	21	46	47	48	73	74	75
22	23	24	49	50	51	76	77	78
25	26	27	52	53	54	79	80	81

Figure 3.1: Logical topology of processes and tiles. Each 3x3 square represents a process and the numbers in the square represent the IDs of the nodes in the process.

have notified the System that they are being terminated (by calling the member function *increment_bridge_pos()* which increments the member variable by 1).

Then, the *killed_threads_lock* and *bridge_selector_locks* are destroyed and the bridges are deallocated since they are not needed anymore. Finally, *MPI_Finalize()* is invoked, which ensures the proper termination of the MPI environment [9, p. 357].

During the development of this project, new member functions were added to the System class. Some of them are just used for printing messages regarding the the arrangement of the process ranks on the topology created or the execution of the program. However, there are some other member functions that are have a more practical use. For example, *System::kill_thread()* is used by receiving threads to notify the System instance about their termination. Some of these functions are mentioned and described later in this chapter.

3.1.2 The Transceiver Class

In the original GMCF code the constructor of the *Tile* class creates an instance of the *Transceiver* class, which holds 2 FIFOs that the tile can use for storing packets. TX FIFO stores packets to be transmitted to other tiles, whereas RX FIFO contains the packets received from other tiles. The *Tile* class contains a member function *run()*. During the execution of that function, a tile invokes the member function *run()* of its transceiver. *Transceiver::run()* calls *Transceiver:: transmit_packets()* which is responsible for transmitting GMCF packets from the TX FIFO of the transceiver to the RX FIFO of the target tile. The transceiver first pops the packet from the top of the TX FIFO and checks the *To* field of the packet header. This field indicates the *node_id* of the destination tile. If the value of that field is 0 the packet is sent to the gateway tile, a special tile whose purpose is outside the scope of this project. In any other case, a simple computation involving modulo arithmetic would take place to ensure that the *node_id* of the tile to receive the packet is not larger than *NSERVICES*. For example, if the value of the To Field was 73 and the value of *NSERVICES* was 64 the packet would be sent to the RX FIFO of the tile with *node_id* equal to $((73 - 1)\%64) + 1 = 9$, where $\%$ denotes a modulo operation. Then, the popped packet is pushed in the back of the RX FIFO of the tile with that *node_id*.

This implementation works successfully in shared-memory architectures in which different instances of the *Transceiver* class can read data (like float arrays) allocated on the free store. However, this would not be successful in a distributed-memory architecture. In this scenario, the required data should be copied on the memory of the receiving (target) process. This is the reason why bridges were created.

In the new design, the transceiver performs a similar computation. Instead of using a modulo operation with *NSERVICES* as a divisor, it uses the product of the number of MPI processes and *NSERVICES*. That way, the transceiver ensures that the result will not be a *node_id* that does not exist (the largest *node_id* will be the last created tile in the process with the highest rank and its value would be $process\_rank \times NSERVICES$, since each process has *NSERVICES* tiles). The result of the operation will be the *node_id* of the tile that will receive the popped GMCF packet. The transceiver, then, compares the rank of the process that the target tile is in to the rank of the process it is running on. The receiver's rank is calculated by simply dividing *node_id* -1 by *NSERVICES*. If the two process ranks are the same, the transceiver simply pushes the packet in the RX FIFO of the corresponding tile, just like in the original GMCF code. However, if the receiver's rank is different, then a bridge will have to be used. The way that the packet will be transmitted depends on whether the *THREADED_SEND* macro was used. If it was defined during compilation, the transceiver will call the *System::send_th()* method which will create a new thread that will receive the packet and send it to the destination. If *THREADED_SEND* was not defined, *System::send()* would be called instead, which would invoke *Bridge::send()* and that member function will handle the transmission of the packet to the destination. The threaded way will not block the execution but is slower because creating a new thread for each outgoing packet can be quite expensive. An alternative way using a thread pool is discussed in section 5.2.

The transceiver should not be concerned with how the packet will be or if it were transmitted, since the *System* instance and a bridge will handle the transmission of the packet to the right process and tile.

3.2 The Bridge Class

The constructor of the Bridge class receives a pointer to the System instance that initiated the creation of a new Bridge object and the rank of the current MPI node. The pointer to the System instance is required for many tasks, such as the detection of the tile to receive an incoming GMCF packet or storing the time elapsed since starting a performance test. The rank is required mainly for printing messages on the screen when the *VERBOSE* macro is used. The constructor gets the list of the neighbours of the MPI node in the logical topology and initialises the *recv_lock* lock. This lock is used for ensuring that a receiving thread will retrieve the message it actually probed, instead of another incoming message and it is only used for MPI implementations that are based on versions of the standard that were published before MPI-3.0 (e.g. MPI-2.1). More information on this is provided in section 3.4. The constructor also creates a thread which will listen for incoming MPI messages from other nodes and forward them to the appropriate tile. Therefore, for each Bridge instance that was created on a node there will be one thread listening for incoming messages. The existence of multiple receiving threads on a system means that each of them can handle a different incoming message at the same time, so long as the hardware allows this.

A detailed description of how bridges help tiles pass GMCF packets between them is provided in the following sections.

3.3 Sending Packets

The current implementation of the Bridge class provides two methods for sending GMCF packets: *Bridge::send_th()* and *Bridge::send()*. The former creates a new thread which will handle the transmission of the packet, whereas invocation of the latter means that the thread that invoked the method will block until the transmission of the GMCF packet is completed.

Each time a GMCF packet has to be sent to another node, the tile that has the GMCF packet stored in its TX FIFO invokes a method of the System instance that will then pass the packet to the bridge. The reason that the communication between a tile and a bridge is implemented this way is that the system is responsible for selecting the bridge that will transmit the GMCF packet. In order to avoid using the same bridge for two subsequent packets, the System instance keeps track of the last used bridge. Therefore, if a tile requests a packet to be sent, the System instance sends that packet to the next bridge in the vector that it stores.

Before sending the packet to a bridge, the System instance updates the *bridge_pos*, an integer variable which points to the position of the next bridge to be used in the vector. Since multiple tiles might request a sending operation to initiate, a spinlock was used to ensure the proper update of *bridge_pos*. This is a simple method which ensures that the next bridge to be used should be the one that was least recently used. A different approach to selecting a bridge is suggested in 5.2.

3.3.1 Sending Packets Using New Threads – Non-blocking Approach

This subsection provides a description of *Bridge::send_th()*. The operations performed by invoking this system function are nearly identical to those performed by *Bridge::send()*. Therefore, there is no point in providing a description for that function as well. The differences will be mentioned below.

System::send_th() will pass the packet it received as input to the *Bridge::send_th()* method of the selected bridge (similarly, *System::send()* will invoke *Bridge::send()* and pass the packet to it). Along with that packet, *Bridge::send_th()* can also receive as input an integer indicating the tag to be used during the sending operation. Each message sent using MPI routines carries a tag which can be used for distinguishing between different types of messages and their purpose [9, p. 15]. In this project, when a tile needs to send a GPRM packet to a tile on another instance, it uses the *tag_default* tag. However, other tags have been defined as well. For example, the tag *tag_dresp_data* should be used when a bridge passes a message that contains a float array for a previously sent DRESP packet (more details on the way DRESP packets are handled in the next subsection). Users can define and use new types of tags for operations other that a simple send-receive (e.g. stencil operations). However, this might mean that some modifications to the current code should take place since the receiving threads only listen for *tag_default* or *tag_dresp_data* messages [5].

Bridge::send_th() will create a new thread which will send the MPI message to the corresponding process. The decision to create a new thread that will handle the sending operation was based on

[5] This could be achieved by using different types of receiving threads for each operation

the requirement that tiles should not be blocking while waiting for a packet to be transmitted to a tile on a different node.

The sending thread receives as input a pointer to a structure that contains pointers to the Bridge instance that created the thread and the packet to be transmitted. When a POSIX thread, or pthread, is created it can only receive one argument of type *void**. Therefore, only the absolutely necessary information for the sending operation is passed to the thread (these operations are not required for *Bridge::send()* since it is simply a method of the class Bridge). Memory space for this structure is allocated on the free store, so the thread must ensure that it will be deleted before it terminates.

The reason why it is necessary to know when this operation has been completed is that the sending thread must deallocate the memory of the transmitted data. Packets of type DRESP contain pointers to float arrays that are stored on the free store. Additionally, the packet to be sent is also stored on the free store so that the thread can retrieve it. In order to avoid memory leaks, the thread should be responsible for deallocating the memory used for the aforementioned data after the operation has been completed successfully (*Bridge::send()* only has to deallocate the memory space used for the float array).

The thread retrieves the header of the packet to be sent and finds the field of the header that indicates the node ID of the tile to receive the packet. Then, it performs a simple modulo computation to ensure that the ID will not be larger than the largest ID that any tile on any rank may have. This means that this ID should not be larger than $number\_of\_nodes \times number\_of\_tiles\_per\_node$, because a tile with that ID does not exist and the thread will try to send a message to a node that does not exist, which would lead to an error.

Knowing the actual ID of the tile allows the thread to calculate the rank of the node on which the receiver resides using an integer division. The thread then initiates the sending operation using the *MPI_Isend()* routine [9, p. 49] and constantly checks whether the operation has been completed using *MPI_Test()* [9, p. 52]. If the sent GMCF packet was of type DRESP then the float array that the packet had a pointer to must also be sent. This is performed using the same approach (i.e. *MPI_Isend()* and *MPI_Test()*) but the tag of the message to be sent will be different. More specifically, the tag will be a combination of the sending tile's *node_id*, the receiver tile's *node_id* and the tag type *tag_dresp_data*. The reason why such tags are used for sending float arrays is thoroughly explained in subsection 3.4. Once the whole array has been sent (which might mean that either the target node started receiving the MPI message or that the array has just been copied to a buffer on that node) the sending thread deletes the array to ensure that there will be no memory leaks, since it is no longer needed. Finally, the thread deletes the structure passed to it during its creation before terminating.

3.3.2 The Need for Two Sending Methods

The reason that Bridge provides two methods for sending GMCF packets is that each of those methods has advantages and disadvantages. *Bridge::send()* will wait until the *MPI_Test()* routine will ensure that the MPI message has been successfully sent to the destination node. This means that a thread that calls this method will block until this operation is completed. This will have a negative effect for each tile that wants to send multiple packets and continue its operations as quickly as possible. On the other hand, *Bridge::send_th()* does not cause this problem since it creates a new thread that will wait until the sending operations are successful. However, *Bridge::send()* can be quite fast compared to *Bridge::send_th()* because creating a thread can be an ”expensive” operation and doing so multiple times can cause quite an overhead for the system. A different approach which involves a thread pool is discussed in section 5.2.

3.4 Receiving Packets

Just like the sending threads that are created using *Bridge::send_th()*, the receiving threads are pthreads and can only accept one argument of type void*. However, in this case no packet is required and therefore only a pointer to the bridge that created the thread is passed. Through this pointer, the thread will be able to detect the receivers of the GMCF packets that it has received.

The largest part of the code executed by a receiving thread lies in a loop that checks the value of the variable *active* of the System instance that created the bridge. This variable is set to *false* when the destructor of that instance is invoked and is frequently checked by all receiving threads on a MPI node in order to detect when their execution is no longer needed, since a call to the System destructor would signify that the system is being terminated and no more packets are sent or expected to be received. It should be noted that the thread only checks the *active* variable of the System instance at the beginning of the loop. Therefore, if a MPI message is detected by the thread it will perform all the operations required for retrieving the message (as described later in this section) and will not check the value of that variable again until all these operations have been completed.

Inside the aforementioned loop, the thread will check for incoming MPI messages and if such one is detected it will try to retrieve it. The way that this is achieved depends on the MPI standard that the MPI implementation is based upon. The MPI-3.0 standard introduces the matching routines *MPI_Improbe()* and *MPI_Imrecv()* [9, p. 67], which can be used together for retrieving incoming MPI messages. *MPI_Improbe()* listens for messages and returns a *flag* indicating whether a message was detected or not. The routine also returns two more parameters: a *message* handle and a *status* object. *status* contains information on the detected message. Using this *status*, the receiving thread detects the tag of the MPI message, as well as the size of the packet. The latter is especially useful, since the thread needs to know the size memory to be allocated for the incoming message. The *message* handle works as a unique identifier of the probed MPI message and can be used by *MPI_Imrecv()* to retrieve it. If a call to *MPI_Improbe()* returns such a handle, the message that it refers to is removed from the incoming message buffer, so that no other invocation of that routine can return a handle to it. This makes matching routines ideal for use with multiple threads, as is the case with bridges in this implementation. After a flag that indicates the presence of an incoming message is returned, the receiving thread calls *MPI_Imrecv()* using the *message* handle as an input parameter.

As previously mentioned, *MPI_Improbe()* and *MPI_Imrecv()* were introduced in MPI-3.0. The use of routines defined in older versions was explored in order to provide support for MPI implementations of such versions. Finally, it was decided that *MPI_Iprobe()* [9, p. 64] and *MPI_Irecv()* [9, p. 51] should be used. *MPI_Iprobe()* works similarly to *MPI_Improbe()*, with the exception that it does not return a message handle. This means that a subsequent call to a receiving routine will not ensure that the message detected by *MPI_Iprobe()* will be the one retrieved. Such a situation would occur if another thread was to make a call to a receiving routine before this one. Since a MPI process is expected to create multiple bridges (and hence multiple receiving threads) a synchronisation technique is required for ensuring that after a call to *MPI_Iprobe()* was successful in detecting an incoming message, the same thread will call *MPI_Irecv()* before any other thread. A spinlock is used for this purpose. A thread locks the spinlock before invoking *MPI_Iprobe()*. If no incoming message was detected, it unlocks the spinlock and tries to acquire it again. If, however, a message was detected, it performs some necessary operations for retrieving the message, as explained above. After a call to *MPI_Irecv()* the thread has started receiving the message and it unlocks the spinlock, so that other threads may use it.

The receiving threads use preprocessor directives to detect the MPI version. The MPI versions after 1.2 define an integer named *MPI_VERSION*, which is assigned the version number of the MPI standard implemented. If this integer is less than 3 then the non-matching MPI routines are used (*MPI_Iprobe()* and *MPI_Irecv()*)[6].

After the whole message has been received, using *MPI_Test()* (a routine that, given a handle to a message that is being received, returns a flag indicating if it has been retrieved successfully) to ensure that, the type of the packet contained in the message can be retrieved. If the GPRM packet is of type DRESP, the receiving thread will wait for another MPI message which will contain a float array. The DRESP packet should have a pointer to this array. It is of crucial importance that it is this thread and no other that receives that MPI message since it has already received the contents of the GMCF packet and, therefore, is the only such thread that can "link" the packet to the expected float array. In order to ensure that no other thread will receive the expected data a unique tag is created for labelling the MPI message to be sent. More specifically, the tag is a combination of the *node_id* of the tile that sent the GMCF packet, the *node_id* of the tile to receive the packet, and the type of tag, which is *tag_dresp_data*. This tag can be constructed by the thread that sent the packet and the thread that should receive it (since they both have the information required for its construction) but no other receiving thread can create it since it does not know who the sending and receiving tiles are.

The tag of a MPI message is an number of type *int*, which means that it is 32 bits long[7]. 4 of those bits are used for the tag type, 14 bits for the *node_id* of the receiver and the rest 14 bits for the *node_id* of the sender. This allows 16 (2^4) types of tags for MPI messages and 16384(2^{14}) IDs for tiles to be used. The MPI routines that the thread uses for receiving the data depend on the MPI standard implemented and were described earlier. It should be noted that this implementation assumes that **only one DRESP packet can be sent from one tile to another at a time** and that **every GMCF packet of type DRESP has a pointer to a float array**.

Once the floating-point data has been received and stored in an array a pointer to that array is stored in the back of the DRESP packet, overriding the existing pointer that was used on the sender node, and the packet is stored in the RX FIFO of the receiving tile. Packets that are not of type DRESP are simply stored in that FIFO as soon as they are received, without waiting for any additional data.

After completion of the operations described above, the thread starts listening again for other incoming messages.

The description provided above refers to the behaviour of the receiving thread during use of the bridges with GMCF. This behaviour is modified when the EVALUATE macro is defined. More

[6] Even though MPI-1.3 defines *MPI_VERSION*, versions 1.0, 1.1 and 1.2 don't. However, when the check $MPI\_VERSION < 3$ is performed during compilation, if *MPI_VERSION* was not defined it temporarily receives the value 0. Therefore, this case takes into account the aforementioned MPI versions

[7] This implementation assumes that the compiler allocates 4 bytes (32 bits) for each element of type int

information on that is provided in Chapter 4.

3.5 Compiling, Executing and Testing

As previously stated, the project was not developed using GMCF, but in a separate environment. This was deemed appropriate because the main focus of the project (i.e. the bridges) uses only a small part of GMCF. Therefore, using the whole framework and having to build everything every time some changes were made would not provide any benefits. In order to test the code, the files *main.cc* and *run.sh* were created.

The file *main.cc* contains the main function of the program. This function is used for creating the System instances and performing various tests to ensure the proper communication between nodes, as well as evaluate performance of the implementation.

The main function can execute indefinitely or terminate after a certain period of time defined by the user. Testers can choose which option they prefer and set the appropriate loop. However, users should make sure that no sending threads are active when the System destructor is called. This is of vital importance because, a call to some MPI routines such as *MPI_Isend()* after *MPI_Finalize()* will result in an error.

Certain functions have been developed for testing various aspects of the implementation. These functions receive as input a reference to the System instance, since that is essential for reaching the tiles, loading packets to FIFOs and initiating communication procedures through bridges. More information on these functions can be found in chapter 4.

The program can be compiled and executed using the *run.sh* shell script. In this script, users can define the dimensions of the grid of processes to be created, as well as the macros to be used during compilation of the code. The script will compile the code and then execute the program.

The macros that can be used with the testing program are the following:

BRIDGE Essential for executing this program successfully, required if bridges are to be used in GMCF

NBRIDGES Used for defining the number of bridges per MPI node. If not defined, it will be assigned the value 3. This assignment occurs in the file *Types.h*

THREADED_SEND Used when users want a new thread to be created for each send operation

EVALUATE Used for performing evaluation testing. **Should not be used during actual deployment**

MPI_TOPOLOGY_OPT Used when the (potentially optimised) cartesian topology should be used

VERBOSE Prints messages indicating the operations taking place

3.6 Alternative Implementation Approaches Explored

During the early weeks of development of this project, communication methods other than point-to-point were explored for the purpose of passing messages between tiles. One of those methods was the use of one-sided communication, which was introduced in MPI-2 [8]. This allows Remote Memory Access (RMA) operations to be performed, which allow one process to access a specified part of memory of a different process [5]. During one-sided communication, a process does not need another process to respond to a request for sending or retrieving data. Instead, a source process, which is the process that initiates the RMA operation, can access a memory segment of a target process and read, write or update the data. In order for that to be achieved, the target process must have defined a memory segment which will be exposed to the rest of the processes so that they can manipulate its contents. The exposed memory segment is referred to as a window and in the case of GPRM multiple such windows would be created for each System, so that many Tile instances would be able to store data and wait for other MPI processes to read them. The main RMA MPI routines that would be used are *MPI_Put()* (remote write of data) and *MPI_Get()* (remote read of data) [9, p. 401].

RMA seemed ideal for use with GMCF because it provides a very easy and simple way for the GMCF packets and float arrays to be retrieved by other processes. However, the use of RMA would mean that a process could not know when it would be safe to update the contents of a buffer that is exposed to other processes. There could be some tools provided by MPI which solve the problem of notifying processes that an operation has finished and deallocating memory on the buffer would not cause any problems, but due to time limitations I was not able to study exhaustively what the MPI standard defines for RMA operations. Such a notification system is described in [2], but its implementation and testing would not permit enough time to be allocated on developing the rest of the system.

The current implementation sends 2 MPI messages when a DRESP packet is required to be transmitted between different nodes. One of these messages contains the actual packet, whereas the other is used to transfer the float array that the DRESP packets have a pointer to. Originally, the implementation would send these two together in the same MPI message. In order to achieve that, the floating-point numbers in the array were first reinterpreted to unsigned 64-bit integers (just like the elements in a GMCF packet) using the *reinterpret_cast* expression and then packed along with the contents of the packet before being sent to the target node, which would unpack the data, reinterpret the appropriate elements back to floating-point numbers and store them in an array.

This approach was used for most of the development stages of this project but was discarded during the evaluation stage, which showed that retrieving the mpi message, separating the packet elements, reinterpreting the floats and storing them in an array was a very costly operation. More specifically, tests showed that it would take approximately 3.5 milliseconds to send a DRESP packet with a pointer to an array with 4000 bytes worth of floating-point numbers and receive an acknowledgement packet back using threaded sends.

Chapter 4

Evaluation

This chapter provides a description of tests that were performed for ensuring that the bridges work as intended and for measuring their efficiency. The tests described below were performed using an MPICH implementation based on the MPI-3.0 standard, therefore, the MPI routines *MPI_Improbe()* and *MPI_Imrecv()* were used in the receiving threads. The results of these tests might vary for MPI implementations of earlier versions, especially since additional synchronisation methods should be performed.

The evaluation tests that are described in sections 4.3 and 4.4 were conducted on the Togian machine, which is maintained by the University of Glasgow. The server approximate specs are as follows:

- DELL 815 power edge server.
- AMD opteron 6300 processors. 4 x 16 cores.
- RAM 512 GB.
- Hard drive 6TB.

For these tests the *EVALUATE* macro was used. It is important to note that when the *EVALUATE* macro is used, the receiving thread of a bridge adopts a different behaviour: the thread checks the type of the received GMCF packet. If it is a DRESP packet it sends a packet of type DACK back to the sender to signify that the DRESP packet was received successfully. If the packet is of type DACK it assumes that it is the node that initiated the test and tries to measure the time elapsed since the start of the test by calculating the difference between two variables, one set before initiating the transmission of the DRESP packet and one after the DACK packet was received. No packet is ever stored on the RX FIFO of any tile. Therefore, the *EVALUATE* macro should only be used for testing purposes.

4.1 Qualitative Evaluation - Integration with GMCF

In order for the bridges to be used successfully and for the purpose they were developed, they should be able to be integrated with GMCF. Unfortunately, after the development of the bridges there was not enough time to perform this integration. However, this should be a very easy task, because this was taken into consideration since the beginning of the development of the code and that was the reason why this project was developed using files from the original GMCF code. As stated before, the required files were stripped down to the absolutely necessary code for developing the bridges. Therefore, the integration with GMCF should be easy.

Since one of the requirements defined before the start of the development of this project was that the use of bridges should be optional, an easy way to enable/disable bridges had to be used. The use of a macro with an intuitive name (i.e. *BRIDGE*) was considered a good solution to this task and was therefore used. The code that involves the utilisation of bridges was put between *#ifdef*...*#endif* preprocessor directives to ensure that it will be compiled only if bridges are desired for the execution. However, since the creation and proper execution of bridges requires the use of MPI, the code should be compiled and executed using appropriate commands (e.g. **mpic++** for compilation and **mpiexec** for execution). Additionally, users should define the number of rows and columns in the table representing the logical topology before executing the program. These numbers are required by the constructor of the System class, as stated in subsection 3.1.1.

The submitted project can be compiled and executed using the *run.sh* script. In this script, the number of columns and rows in the logical topology is defined and then the product of those numbers is used as the number of processes to be created by MPI. The GMCF framework uses building scripts as well, so such parameters (as well as flags and macro definitions) can be defined in a similar building script after the integration of this project with GMCF.

4.2 Sending and Receiving GMCF Packets

Each packet sent was a DRESP packet. This type of packet was used because the size of the MPI packet was mainly affected by the size of the float array that the GMCF packet contains a pointer to (All GMCF packets had a size of 5 Word elements, or 5x(64/8)=40 bytes, including the header. This size was considered insignificant). Since many people could have access to the Togian machine there is a chance that during the testing of the project the machine was also used for other purposes.

The main goal of this project is to ensure that the bridges are able to perform the required sending and receiving operations between tiles on different nodes. In order to test this, the functions *send_packet_dresp()* and *send_packet_no_dresp()* were developed[1]. Each of these functions creates two packets and adds them on the TX FIFO of a tile before invoking the *Transceiver::transmit_packets()* member function of the tile's transceiver, in order for the packets to be sent to their destinations. The difference between these two methods is that *send_packet_dresp()* creates DRESP packets, whereas the latter is used for any other type of packet that does not contain a pointer to a float array. Passing the float array successfully and efficiently was one of the major concerns during the development of this project and therefore it had to be tested separately.

The use of these functions showed that bridges can successfully transmit/receive packets to/from bridges on other nodes and store received packets in the RX FIFO of the target tiles. These functions were also modified during development in order to test multiple cases (e.g. multiple tiles on a node sending multiple packets).

[1] These functions can be found in the file *main.cc*

The results of these tests can be observed better by using the *VERBOSE* macro, which will print many helpful messages regarding the message passing on the screen. Some of the messages, for example, will let users know which bridge received an outgoing packet from a local tile, or when a bridge has received an incoming packet from a remote tile, as well as the destination of the packet.

An example output of such messages is shown in A.1.

4.3 Timing the Transmission of Packets

One of the tests performed on bridges aimed to measure the time required for sending a message between two different nodes. For this test, the function *test_time_dresp()* was developed. This function receives the number of packets to be sent and the size of data for each packet. The size of data is given in bytes and represents the size of the float array that will be sent along with a DRESP packet. The program sends one packet at a time and measures the time elapsed since the start of the operations. These operations include the following stages:

process A The *Transceiver::transmit_packets()* member function is called. When this calls occurs there is one packet that should be transmitted to a node on a different process.

process A The packet is sent to a bridge (through the use of *send()* or *send_th()* methods of System and Bridge instances), which then sends the GMCF packet to process B. The type of the GMCF packets that are sent is DRESP, so two MPI messages are sent for each packet as explained in section 3.4.

process B The MPI messages containing the GMCF packet and the floating-point data are received

process B A simple GMCF packet of type DACK is created and sent to process A

process A The ack packet is received and the time it took for this whole operation to be performed is added to the *end_time* variable of the *System* instance

Once all the packets have been sent, the testing function calculates the average time by dividing the aforementioned System variable by the number of packets sent and prints it on the screen. If the *VERBOSE* macro is used the program will also print the time required until either creating a new sending thread or invoking the *Bridge::send()* method (depending on whether the *THREADED_SEND* macro was used) and the total time required by each individual packet. In order to time these tests the MPI routine *MPI_Wtime()* was used [9, p. 356]. This function returns the time elapsed since an arbitrary point in the past.

To test the performance of the bridges, *test_time_dresp()* was used for various data sizes, using both a threaded and a non-threaded way of sending packets. 10000 packets were sent for each data size and sending method. The results of these tests are shown in Figure 4.1.

It is obvious from the figure that using the non-threaded method requires significantly less time than the threaded transmissions at all times. However, the time required for a test is not proportional to the size of the data sent for any of the two methods used. Non-threaded transmissions required 81 microseconds on average for sending a DRESP packet with a float array of 1000 bytes, whereas threaded transmissions required 278 microseconds. The non-threaded method required

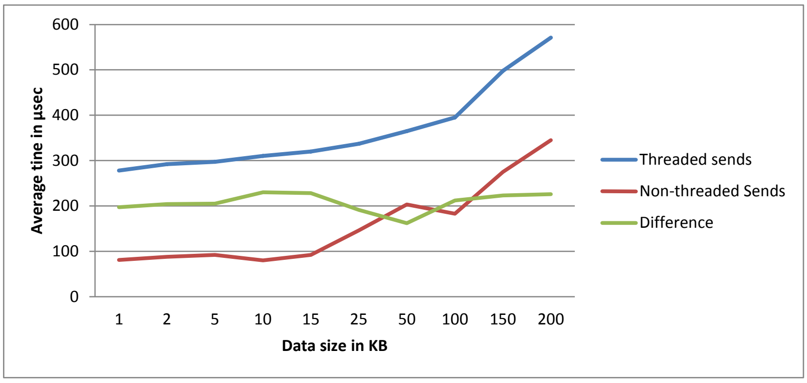


Figure 4.1: The results of sending multiple DRESP packets, one at a time

approximately 86 microsecond on average for transmitting packets of sizes from 1000 to 15000 bytes, while the threaded method required approximately 299 microseconds on average for data of the same sizes. The time required by both method increases as the size of data increases as well, which was a predicable behaviour.

Figure 4.1 also displays the difference in time required between the two sending methods that were used. It is easy to observe that the difference at each time was around 208 microseconds. This could be attributed to the creation of new threads that were used for sending the packets. Since two threads were created for this purpose (one responsible for sending the DRESP packet from A to B and one for sending the DACK from B to A) one can assume that the creation of a thread might require approximately 100 microseconds. This means that creating a thread for each sending operation might be too expensive if it is expected to make hundreds or thousands of such transmissions, so non-threaded is the best of those two methods. However, the non-threaded method has the tile that invoked it block until the message has been sent successfully, which is not a desired behaviour. An alternative way, which combines the benefits of both methods, is suggested in 5.2.

It is worth noting that the non-threaded transmissions did better with the data of size 10000 bytes than they did with 5000 or 15000 bytes. A similar behaviour was detected when 100000 bytes of data were sent. I was not able to attribute these results to a certain cause and I can only assume that it has to do with the use of the Togian machine by other users when those tests were performed.

A table with the precise results of these tests is provided in A.2.

4.4 Timing the transmission of single MPI message for varying number of bridges

In order to test the overhead that the existence of multiple bridges running on one node might cause to the system, similar tests were performed for varying numbers of bridges, while keeping the size of the data size to 200000 bytes. The results of those tests are depicted in Figure 4.2.

The figure shows that when two bridges are used the average time required for a packet transmission increases by approximately 400 microseconds for threaded and 350 microseconds for non-threaded sends. An increase can be observed if more bridges are added to each node on the system as well. This behaviour should be expected because each bridge creates a receiving thread that competes

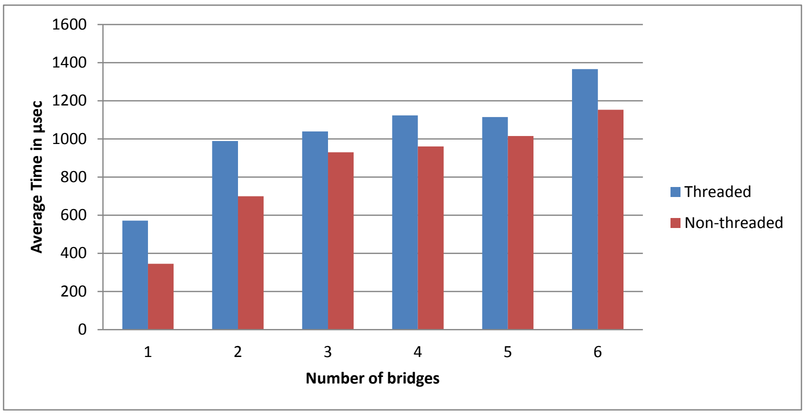


Figure 4.2: The results of sending multiple DRESP packets, one at a time, with a varying number of bridges

with other such threads for retrieving incoming MPI messages. In the scenario tested, however, only one bridge is required since there is no chance of a second message being stored on the receiving buffer and waiting to be retrieved. The results in Figure 4.2 simply indicate that if only one packet is to be transmitted between two nodes at a time one bridge on each node would provide the ideal performance.

4.5 Memory Leaks

Proper memory management is essential for this project, just like for most programs written in C++. This will ensure that the result of this project is robust and reliable. In order to ensure that no memory leaks would be introduced by this project, the use of a tool for memory leak detection was considered. Valgrind is a framework that can be used for this task [10]. However, it has been noted that the use of Valgrind with MPI may lead to many false errors [14]. These errors can be reduced by writing wrappers for MPI routines, as explained in [14], but non-blocking routines, such as the ones used in this implementation, will make this task very challenging. Therefore, the detection of memory leaks was based on studying the code.

In GMCF, bridges will receive existing packets, so it will not be responsible for allocating memory for such packets. However, upon receipt of a DRESP packet and the floating-point data accompanying that packet, a bridge must allocate memory space for the float array. This is handled properly in the current code. After this packet is stored in the RX FIFO of the destination tile it will be ”consumed” by a model executed on the tile and after its use the allocated memory will be freed by that tile. Therefore, there is no need for the bridges to handle this task. This means that this implementation should be responsible for freeing the memory space used in the following cases:

- Memory on which the bridges are stored during their creation (System constructor)
- Memory on which the structures that are passed to sending threads are stored (*Bridge::send_th()*)
- Memory on which the float array of a DRESP packet is stored after it has been sent to a different node, since it’s no longer useful to the sender

- Memory allocated during the initialisation of MPI through the call to *MPI_Init_thread()* in the System constructor.

This implementation handles all of those tasks when it is deemed appropriate.

Chapter 5

Conclusion

5.1 Discussion

The purpose of the project documented in this dissertation aims to provide an extension to the GPRM framework that is used by GMCF, a model coupling framework. Currently, GPRM is able to utilise the shared-memory architecture of a system. The project in question intends to provide a way for nodes on a distributed-memory architecture communicate with one another when this is required, and doing this while being as transparent to the user as possible. The result of this project satisfies the aforementioned requirements, as well as some other that were set during the early stages of development. The project allows the creation of logical gateways (or bridges, as they are called in the context of this project) that provide a way for threads on different nodes to communicate with one another by passing messages. This is achieved using MPI routines designed for point-to-point communication.

This project was not integrated with the GMCF code, but it was developed using files of that framework. Therefore, the integration should be very easy. Additionally, the code was written in such a way that allows users to select whether the result of this project (i.e. the bridges) should be utilised during the execution of the program or not.

Tests on the result of this project showed that the communication between two tiles on different nodes can be achieved successfully. Experiments focusing on the performance of bridges are not ideal but show promise. However, GMCF is used for computationally intensive operations and bridges would be frequently used for passing messages between tiles on different nodes. Therefore, more tests should be performed in order to ensure their desired operation. Additionally, suggestions for improvements in performance are suggested in the following section.

5.2 Further Work

To ensure the proper execution of the project, synchronisation techniques were required. The method that was used during the development of this project was utilisation of spinlocks. Threads that try to acquire a spinlock are stuck in a loop until no other thread has a hold of the lock. Having multiple threads in a loop means that they will keep using system resources just by waiting until the spinlock is released. However, other types of synchronisation mechanism could be used instead. One such example would be mutexes. If a thread tries to acquire a mutex that is already held by another thread, it will go to sleep. When the mutex is released, one of the threads that requested the mutex but are asleep will wake up, acquire the mutex and execute the required operations. Then the thread releases the mutex which will be available for another thread. In order to decide which of these two would be the best for use with GMCF thorough testing on the hardware that GMCF would run on should be conducted.

Currently, there are two options for sending packets between nodes: creating a new thread that will handle the transmission or invoke a member function of a Bridge instance that will perform the same operation. The former approach was shown to be slower in section 4.3 but it offers a significant advantage: the tile that initiated the send operation does not block until the message is sent. In order to combine the advantages of the aforementioned methods a thread pool could be created. This thread pool would contain a number of sending threads that would constantly listen for GMCF packets that needed to be sent to other nodes on the system. These packets could be stored in a thread-safe structure on the System instance running on the node, since all bridges have access to it. Once a thread detected an outgoing packet it would retrieve it, send it and then listen for new packets. This means that they would run indefinitely, or until the termination of the System was invoked (e.g. when the System destructor was called, a method that was used for terminating receiving threads). This would avoid creating a new thread for each requested send operation and it would allow tiles to continue their execution without blocking until the whole packet was transmitted.

As explained in chapter 3, other ways of selecting which Bridge instance will send a GMCF packet can be explored. One approach would be to use flags for each bridge that indicate its status. For example, each bridge could have a boolean variable called *sending* that is false when a thread of that bridge performs a send operation and false otherwise. Initially, *sending* would be set to false. Then, when a bridge was instructed to send a new GMCF it would change to true. Finally, when the sending thread has been finished with the send operation the value of *sending* should be set to false before the thread terminates. The System instance could check the *sending* member variable of each bridge and decide which would be the most suitable one for the next send operation. If all the bridges had such threads executing, then the System instance could either wait until one of them was available again or use a random bridge anyway (meaning that that bridge would have two active threads performing a send operation), which means that synchronisation techniques should be utilised in case the threads will try to modify the state of the bridge (there is currently no such issue but a future update could add this danger).

Appendix A

First appendix

A.1 Simple Packet Transfer

Below are the messages printed as the result of sending packets between nodes. For this test the following parameters were used

- Compilation using the macros *BRIDGE NBRIDGES=4 VERBOSE*
- *NSERVISES* (indicating the number of tiles on each node) was assigned the value 9
- invocation of the functions *send_packet_dresp()* and *send_packet_no_dresp()*

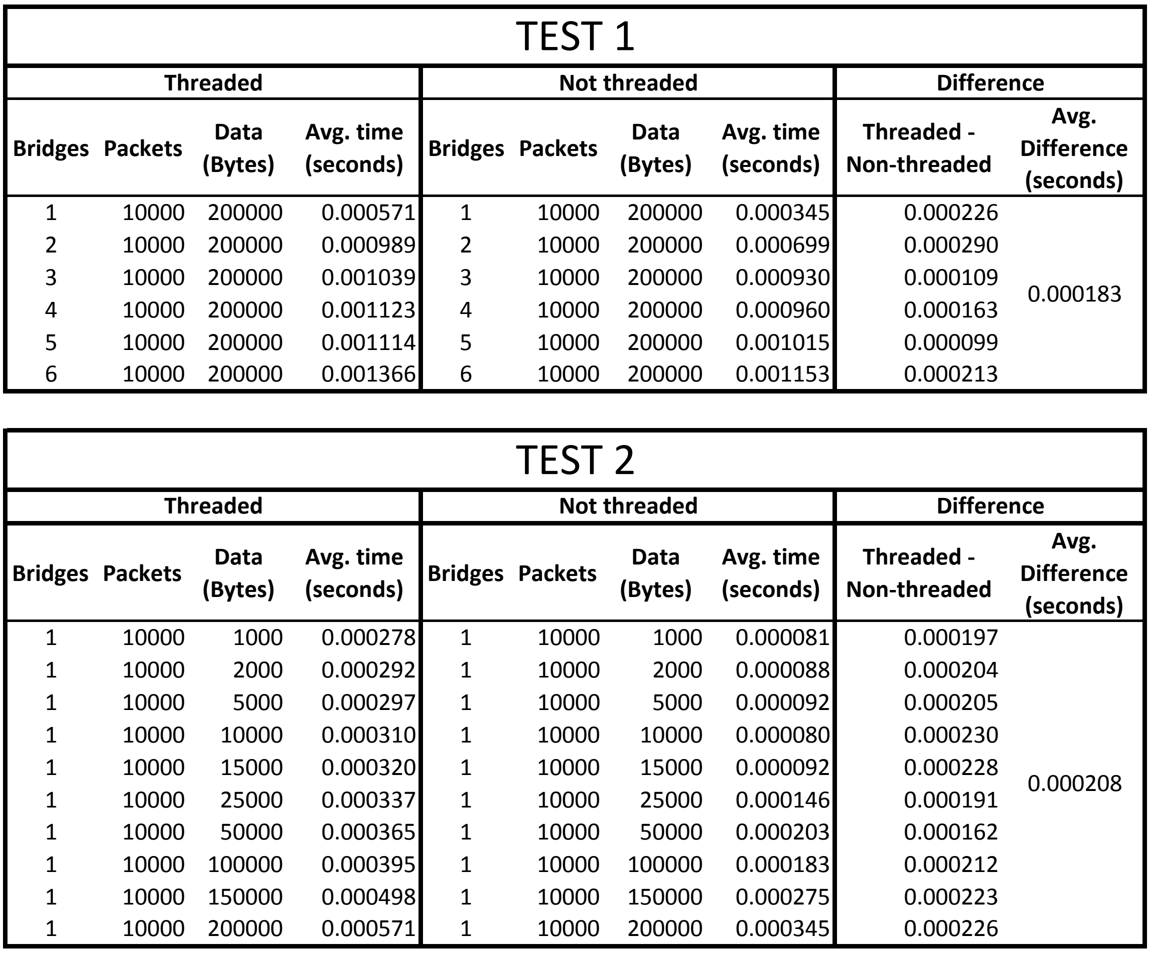
The messages that were related to the initialisation of the system are not provided.

OUTPUT

```
2 TRX:transmit_packets(): FIFO length: 2
WARNING: ad-hoc dest computation (MPI): 29 (original: 29)
Rank 0 (transceiver): sending packet to a different MPI node
Rank 0: Bridge 0(0-3) was selected to send a message
Rank 0: Sending float array...
Rank 0: Float array sent!
Rank 0 (Sent): Sent a packet to 3
WARNING: ad-hoc dest computation (MPI): 18 (original: 126)
Rank 0 (transceiver): sending packet to a different MPI node
Rank 0: Bridge 1(0-3) was selected to send a message
Rank 0: Sending float array...
Rank 0: Float array sent!
Rank 0 (Sent): Sent a packet to 1
6 TRX:transmit_packets(): FIFO length: 2
WARNING: ad-hoc dest computation (MPI): 29 (original: 29)
Rank 0 (transceiver): sending packet to a different MPI node
Rank 0: Bridge 2(0-3) was selected to send a message
Rank 0 (Sent): Sent a packet to 3
```

```
WARNING: ad-hoc dest computation (MPI): 29 (original: 29)
Rank 0 (transceiver): sending packet to a different MPI node
Rank 0: Bridge 3(0-3) was selected to send a message
Rank 0 (Sent): Sent a packet to 3
Rank 1: Received msg from rank 0
Rank 3: Received msg from rank 0
Rank 3: Waiting for a float array...
Rank 3: Float array received!
Rank 3 (Recv): Sent packet to dest 29
Rank 3: Received msg from rank 0
Rank 3 (Recv): Sent packet to dest 29
Rank 3: Received msg from rank 0
Rank 3 (Recv): Sent packet to dest 29
Rank 1: Waiting for a float array...
Rank 1: Float array received!
Rank 1 (Recv): Sent packet to dest 18
```

A.2 Results of Performance Experiments



TEST 1

Threaded				Not threaded				Difference	
Bridges	Packets	Data (Bytes)	Avg. time (seconds)	Bridges	Packets	Data (Bytes)	Avg. time (seconds)	Threaded - Non-threaded	Avg. Difference (seconds)
1	10000	200000	0.000571	1	10000	200000	0.000345	0.000226	0.000183
2	10000	200000	0.000989	2	10000	200000	0.000699	0.000290	
3	10000	200000	0.001039	3	10000	200000	0.000930	0.000109	
4	10000	200000	0.001123	4	10000	200000	0.000960	0.000163	
5	10000	200000	0.001114	5	10000	200000	0.001015	0.000099	
6	10000	200000	0.001366	6	10000	200000	0.001153	0.000213	

TEST 2

Threaded				Not threaded				Difference	
Bridges	Packets	Data (Bytes)	Avg. time (seconds)	Bridges	Packets	Data (Bytes)	Avg. time (seconds)	Threaded - Non-threaded	Avg. Difference (seconds)
1	10000	1000	0.000278	1	10000	1000	0.000081	0.000197	0.000208
1	10000	2000	0.000292	1	10000	2000	0.000088	0.000204	
1	10000	5000	0.000297	1	10000	5000	0.000092	0.000205	
1	10000	10000	0.000310	1	10000	10000	0.000080	0.000230	
1	10000	15000	0.000320	1	10000	15000	0.000092	0.000228	
1	10000	25000	0.000337	1	10000	25000	0.000146	0.000191	
1	10000	50000	0.000365	1	10000	50000	0.000203	0.000162	
1	10000	100000	0.000395	1	10000	100000	0.000183	0.000212	
1	10000	150000	0.000498	1	10000	150000	0.000275	0.000223	
1	10000	200000	0.000571	1	10000	200000	0.000345	0.000226	

Figure A.1: Results of the experiments discussed in sections 4.3 and 4.4

Bibliography

[1] M. Aswad, P.W. Trinder, and H.W. Loidl. Architecture aware parallel programming in glasgow parallel haskell (gph), 2012.

[2] R. Belli and T. Hoefler. Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization. IEEE, May 2015. Accepted at 30th IEEE International Parallel & Distributed Processing Symposium (IPDPS'15).

[3] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: Co-array fortran and unified parallel c. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 36–47, New York, NY, USA, 2005. ACM.

[4] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned Global Address Space Languages. *ACM Computing Surveys*, page 29, January 2016.

[5] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. Enabling highly-scalable remote memory access programming with mpi-3 one sided. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 53:1–53:12, New York, NY, USA, 2013. ACM.

[6] William Gropp and Rajeev Thakur. Issues in developing a thread-safe mpi implementation. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, EuroPVM/MPI'06, pages 12–21, Berlin, Heidelberg, 2006. Springer-Verlag.

[7] Simon Marlow. Parallel and concurrent programming in haskell. In *Proceedings of the 4th Summer School Conference on Central European Functional Programming School*, CEFP'11, pages 339–401, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.0, November 2003. available at: `http://www.mpi-forum.org`.

[9] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.0, September 2012. available at: `http://www.mpi-forum.org`.

[10] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[11] Open-mpi.org. Mpi_init_thread(3) man page (version 1.8.8), 2015. available at: `https://www.open-mpi.org/doc/v1.8/man3/MPI_Init_thread.3.php`.

[12] Jeff Squyres. *MPI: What Really Happens During MPI_INIT?* 2004. available at: `http://cw.squyres.com/columns/2004-02-CW-MPI-Mechanic.pdf`.

[13] Ashkan Tousimojarad and Wim Vanderbauwhede. The glasgow parallel reduction machine: programming shared-memory many-core systems using parallel task composition. *Electronic Proceedings in Theoretical Computer Science*, 137:79–94, December 2013.

[14] Valgrind.org. Valgrind - memcheck: a memory error detector, 2015. available at: `http://valgrind.org/docs/manual/mc-manual.html`.

[15] Wim Vanderbauwhede. Model coupling between the weather research and forecasting model and the DPRI large eddy simulator for urban flows on gpu-accelerated multicore systems. *CoRR*, abs/1504.02264, 2015.