University of Tartu
Faculty of Mathematics and Computer Science
Institute of Computer Science
Computer Science Specialty

Andres Luuk

Porting MPI applications to the Friend-to-Friend computing framework

Master Thesis (20 CP)

Supervisors: Ulrich Norbisrath
Eero Vainikko

Author: …………………………………….. "……" May 2008
Supervisors: Ulrich Norbisrath……………..."……" May 2008
/Eero Vainikko…………………"……" May 2008

Professor: Eero Vainikko …………………... "……" May 2008

TARTU 2008

Contents

Acknowledgements

There are some people I would like to thank for their help in making this thesis. First I would like to thank my supervisor Ulrich Norbisrath. He gave many suggestions about writing the work and helped me with the license conflict with P2P-MPI. Then I would like to thank Stéphane Genaud from the P2P-MPI project who gave us the license to modify P2P-MPI under LGPL license instead of GPL. Last, but not least, I would like to thank Keio Kraaner helping me understand and modify the F2F Computing framework.

Chapter 1

Introduction

This thesis is about using the Message Passing Interface (MPI) [45] in a P2P computing framework called F2F Computing [43].

The F2F Computing framework is an easy to use desktop Grid framework that supports distributed computing. The framework is written in Java and runs as a SIP Communicator plug-in [36]. It sends your distributable Java program to selected peers and executes it on them. Of course the framework supports communication between the peers.

However there are some usability problems concerning the F2F Computing framework. For example, we must write a lot of code to do some common operations that we need in different programs. At the beginning of the task we must always find all other allowed peers and explicitly submit our task to them. In the communication, when we need to send some messages to all connected peers, we must iterate over all the peers and send each message separately. There is no support for synchronization or checkpointing. It would be better if we could write the programs so that we must not write the extra functionality every time we make a new program.

One of the solutions is to use some standard of distributed computing in the F2F Computing framework to make programs much shorter and simpler. Then we will not have to submit tasks to all the peers explicitly ourselves and must not spend time writing code for sending and collecting broadcast results from all the peers one by one.

Message Passing Interface (MPI) has become the *de facto* standard in distributed and parallel software development. In this thesis I create an implementation of MPI for the F2F Computing framework so that we can write programs much shorter and easier to understand. An other advantage of MPI is this, if F2F supports MPI then we can port other already existing MPI programs to F2F without having to rewrite the communication logic of the programs. The Distributed Systems Group in the University of Tartu has some legacy MPI programs in its DOUG (Domain Decomposition on Unstructured Grids) [42, 1] system. If we have MPI capability in the F2F Computing framework then we can port the legacy DOUG programs to the framework much easier.

If we can use MPI in F2F, the initialization of tasks will be easier with just one line of code. MPI specifies also a lot of communication functions in its standard. We can use them to reduce our 10-line broadcast functions to just one line of code so that we can spend more time writing the actual program. As you can, imagine there will be much less code for peer communication and much more code of the programs will do some actual work.

To solve this problem one possibility is to write an entirely new MPI implementation for the F2F Computing framework, but another possibility is to take an existing Java MPI implementation and integrate it in to the F2F Computing framework. I chose the second possibility, and for getting MPI support to the F2F Computing framework I take a pure Java MPI implementation P2P-MPI [32], and integrat it into the framework.

In this thesis I will give an introduction to the MPI and the F2F Computing framework, what they are and why they where created. After this I will give an introduction to a pure Java MPI implementation – P2P-MPI and how do I integrate it with the F2F Computing framework. An overview follows on implemented MPI commands and what these commands do exactly. For most of the commands, I am giving also an example. There will be a summary of the usability of the MPI on the F2F Computing framework. I will also provide an example program with detailed explanations and references to some more example programs on how to use the ported MPI on the F2F Computing framework. A reference to the frameworks home page [3] is given where cyou an download the newest version of F2F which includes F2F-MPI.

Chapter 2

MPI

2.1 What is MPI?

The Message Passing Interface (MPI) is a standard for distributed and parallel computing. The goal of MPI is to give a standard witch can be used in different environments for distributed and parallel computing. Now MPI has become the *de facto* standard for distributed and high performance computing.

MPI specifies a set of procedures to be used in message passing between different processes. For these procedures MPI specifies the logic behind them, which is what the commands must do, not how the commands must be implemented. This makes the standard more usable, as the way of sending information can be implementation specific and a specific implementation may need different ways of communication patterns for performance or there may be a need to cache messages.

The goals of MPI are performance, scalability, and portability. MPI is considered to have been successful in meeting these goals. There are a lot of platform specific implementations of MPI out there (FORTRAN [6], C [6], C++ [29]), because the standard is easy to implement in different environments. This is also the reason way the performance of MPI is good. From the many implementations of MPI the people can use the best performing most commonly. Due to the fact that there are a lot of compilers implemented, in different languages, in different operating systems, the programs written in MPI are easily portable.

MPI is used by people who need to make large scale calculations. A lot of supercomputers and Grid systems have platform specific MPI implementations. They use it for communication between parallel processes and for making large scale calculations. [7]

However, due to MPI being an old standard, it is defined for a procedure oriented paradigm and not for the Object-Oriented paradigm.

2.2 History of MPI

In the beginning of the 90-s there were a lot of different programs for parallel and distributed computing, but they were all chaotic and had no standard to follow. There was a need for such a standard in this field. In 1992 80 people from 40 organizations, representing vendors of parallel systems, industrial users, industrial and national research laboratories and universities came together and created the MPI Forum [8]. Its goal was to make a usable standard for distributed and parallel system, getting ideas from existing systems, but not selecting one of them for the standardization. [45]

The process of standardization began in a workshop in Williamsburg, Virginia at the end of April in 1992. After just one year of work in 1993 the first draft of MPI-1 was finished. It was meant to "get the ball rolling" and promote discussion. This effort was successful, because the first draft had the main features and the first official version of MPI-1 was already presented at June 1994. [45]

In 1995 an updated version of MPI was released - MPI-1.1. The changes from Version 1.0 are minor. In Version 1.1 MPI Forum had corrected some errors and made clarifications in the MPI documentation.

After MPI-1.1 was finished the MPI Forum started working on MPI-2. Their goal was to make corrections and clarifications for the MPI-1.1 documentation and future developing the standard with new functionality. 1996 the MPI-1.2 and MPI-2 standards were released.

MPI-1.2 added some small functionality and clarification to MPI-1.1 and was quickly taken to use and it is used even now, because there are a lot of legacy programs. MPI-2 added a lot of new functionality to the MPI standard and it is backwards compatible to MPI-1. However, at that time MPI-1 was already out with a lot of stable implementations. Most people sicked to the version MPI-1.2, instead of trying to use the new, jet unstable and half implemented MPI-2 implementations. Today of course MPI-2 is fully implemented and usable. [8, 7]

After that, the MPI Forum was dormant. The MPI Forum had its mailing list, but was no work there done on the standard. In 2006 the MPI Forum became active again for the purpose of clarifying MPI-2 issues and in the beginning of 2007 the MPI-2.1 was released. Currently the MPI Forum makes effort in making the MPI-2.2 and MPI-3 standards. [8]

2.3 Existing implementations

There exist a lot of different MPI implementations. Most of them are written in C, C++ and assembly language, and are meant for C, C++ and FORTRAN programmers. [7]

The first implementation of MPI was MPICH [41, 29], from Argonne National Laboratory and Mississippi State University. Argonne National Laboratory has developed its MPICH even further, now there is a MPI-2.1 implementation called MPICH 2. Most of the earlier supercomputer companies in 90-s used a commercialized version of MPICH or made their own implementation of MPI-1. IBM was also one of the first implementers of MPI for its supercomputers. [7]

Another early open implementation of MPI was LAM/MPI [47, 6] from Ohio Supercomputing Center. Later LAM/MPI, FT-MPI from the University of Tennessee and LA-MPI from Los Alamos National Laboratory merged and made a new MPI project - Open MPI [40, 31]. Open MPI-s aims to build the best MPI library available. It was created from these three implementations, as the developers of Open MPI thought that they excelled in some areas and would be a good start for an open project. [7]

There are attempts to make an MPI implementation to other languages than C, C++, or FORTRAN.

For example, at least five Python implementations exist: PyMPI [46], mpi4py [28], PyPar [34], MYMPI [48] and the MPI module in ScientificPython [35]. PyPar, MYMPI, and the MPI module in ScientificPython are typical Python modules that can be imported into your program and then be used like any other module. PyMPI is more interesting, as it is a variant Python interpreter with integrated MPI capabilities. You do not use imported modules with PyMPI for running MPI, but PyMPI Python interpreter implements the MPI commands. You can just write the MPI commands inside your program, the PyMPI makes the calls automatically for the compiled code. [7]

Microsoft has its own MPI implementation MS-MPI (Microsoft Messaging Passing Interface) [9]. It is used to communicate between the processing nodes on the cluster network in Windows Compute Cluster Server [37]. MS-MPI implements MPI-2 with over 160 functions, but has been slightly modified, due to the security reasons.

There is even a .NET implementation of MPI: Pure Mpi.NET [33]. It is an

Object-Oriented API for MPI implemented in .NET that takes full advantage of .NET features.

There have been some attempts to make MPI implementations for Java. One of the first was Bryan Carpenter's with mpiJava [39, 30]. It uses local C MPI libraries (through JNI wrappers) to make the MPI calls. It is a hybrid implementation and not very portable unlike most of the Java programs, as this approach uses platform specific C MPI libraries. Another approach is to implement the MPI API fully in Java. This is much more portable and easier to use. One full Java approach to MPI is P2P-MPI [32]. It has implemented a subset of MPI commands and uses the MPJ [38] API. The MPJ API is an MPI API defined for Java, it trys to follow Sun Microsystems coding conventions and be more object-oriented. [7]

2.4 Chapter Summary

Message Passing Interface or short MPI is the *de facto* standard for distributed and high performance computing. The first version of MPI was made in the beginning of to 90-s and now MPI is in version MPI-2.1. The implementations of MPI are mostly for FORTRAN, C, or C++ programming languages. There are a lot of MPI programs out there and it is the *de facto* standard for distributed computing, so we will implement it in the F2F Computing framework for allowing better usability.

Chapter 3

F2F Computing framework

3.1 What is the F2F Computing framework?

The Friend-to-Friend (F2F) Computing framework [43] is a lightweight desktop Grid framework. It utilizes instant messaging for easier Grid management and Peer-to-Peer techniques for faster communication between peers. The framework is based on an instant messenger, this allows easily setting up your own Grids and then distributing your application on it. One of the concepts behind making the Grid in instant messenger is that you know all the people in the Grid, as they are your friends and they can trust you to use their computer.

The framework is developed by the Distributed Systems Group in Tartu University [2]. It is written in Java and is a SIP Communicator [36] plug-in. Like the framework itself, the programs for F2F must be written in Java. The main reason why the language Java is chosen is because Java is multiplatform and easily portable and one of the goals of the F2F Computing framework is to make a gird system that can be run anywhere. SIP Communicator is chosen for the Grid composition, because it is an instant messenger written in Java and it has a plug-in based architecture.

If a Grid is composed then F2F always supports connection between peers. The first possibility of communication comes from the fact that the framework is an instant messenger client plug-in, it can always send messages between peers by instant messages. As these connections are usually slow the framework supports better and faster types of communication between peers. The speed of communication is important in distributed computing. The framework will choose the fastest way of communication possible. If direct TCP between peers is possible then it is used. If it is not possible, then the framework will try to do NAT traversal hole punching for getting a direct and fast link between peers. [44]

3.2 History of F2F Computing framework

The concept's idea behind F2F computing is Ulrich Norbisrath's. He thought it would be nice to have an easy to use desktop Grid system where you would not have to spend a lot of time managing it. You would just collect some friends in your instant messenger client and then you could run your job on their computers and get the result much faster than making it only yourself.

The development of the F2F Computing framework began in spring 2007. The lead programmer behind the project is Keio Kraaner whose main focus is to see that the development will be on track and that the framework will become usable. In the development of F2F the chosen language is Java, because it is multiplatform. The instant message client behind the framework is chosen to be SIP Communicator, as it is written in java and supports plug-ins.

Now a first version F2F is almost ready. And the collecting of ideas for next version has already begun. Of course there will be some more new features. Security will be paid more attention in the next version due to lower priority in the very beginningy. There will be more configuration possibilities like saying how much of your processor time you can give to a process.

3.3 Chapter Summary

The F2F Computing framework is an easy to use Grid framework that utilizes instant messenging for easier Grid management and Peer-to-Peer techniques for communication. F2F is written in Java and it is a SIP Communicator plug-in. The F2F Computing framework is a new framework as its development began only about a year ago. The main part of my work is for further advancing this framework by giving the MPI capability to this framework.

Chapter 4

F2F-MPI

4.1 Why do we need MPI on F2F?

F2F Computing framework is written as an enviroment for writing distributable programs, it supports distributing the program and then allowing simple message sending between peers. Initially the development of F2F Computing framework had the focus was on making a stable and usable framework, and not on making a lot of usability functionality, for making the programming of your programs easier. This is the reason I make this work. To make the programming easier I will extend the F2F Computing framework with MPI functionality, because MPI has become the *de facto* standard in distributed and parallel software development.

To achieve this I have two possibilities:

- Write an MPI implementation from scratch
- Try to adopt an existing implementation to our framework.

I chose the integration of an existing pure Java MPI framework called P2P-MPI. The advantages of it is, that I will have to rewrite the basic communication logic and will get more advanced functions from an already existing system. Another benefit is that there will be test programs for the old MPI system, and it will be easy to import them to our new MPI implementation and see if the system is working properly.

4.2 What is P2P-MPI

P2P-MPI (Peer-to-Peer MPI) [32] is a pure Java MPI implementation written in Java for Java programs in the University of Louis Pasteur.

The main features of P2P-MPI are:

1. No need for platform specific OS binaries - it is a pure Java program.
2. Management of peers is easy – uses super node to locate, add or remove them.
3. Application can have fault-tolerance – system holds track of replicas of running peers, so if one dies then there is another to replace it.

The advantages of a pure Java implementation is that if you can run the Java program then you can run the MPI. This means that the program is highly portable and can be run on any computer without having to specially port it. The disadvantages of a pure Java implementation are that there is no platform specific code and so the performance is said be not that good as with platform specific libraries.

Although the P2P-MPI is peer to peer it now lacks some qualities present in common P2P file sharing applications. It can establish TCP connections between peers, if the peers see each other and the ports are open. However, if the peers are behind firewalls the TCP connection is not possible. Another P2P functionality that it is lacking is UDP hole punching. If the P2P application could do this then we could connect computers form to different networks.

Another drawback is that you must know some configuration information before starting. For example you must know the host of the super node. Otherwise P2P-MPI can not get the information of other peers. If the super node is down or behind a firewall then we simply can not use P2P-MPI.

Another good thing about P2P-MPI is the following, it implements the interface of MPJ [39]. This is an MPI interface that has been modified slightly for Java. So if there are other Java programs written for MPJ, they can be easily ported to P2P-MPI.

As an extra feature P2P-MPI has a peer graph visualization tool. It allows looking up graphically what peers are in the network (Figure 4.1).

4.3 Integrating P2P-MPI

The main focus of this work is practical: Integrating P2P-MPI into the F2F Computing framework.

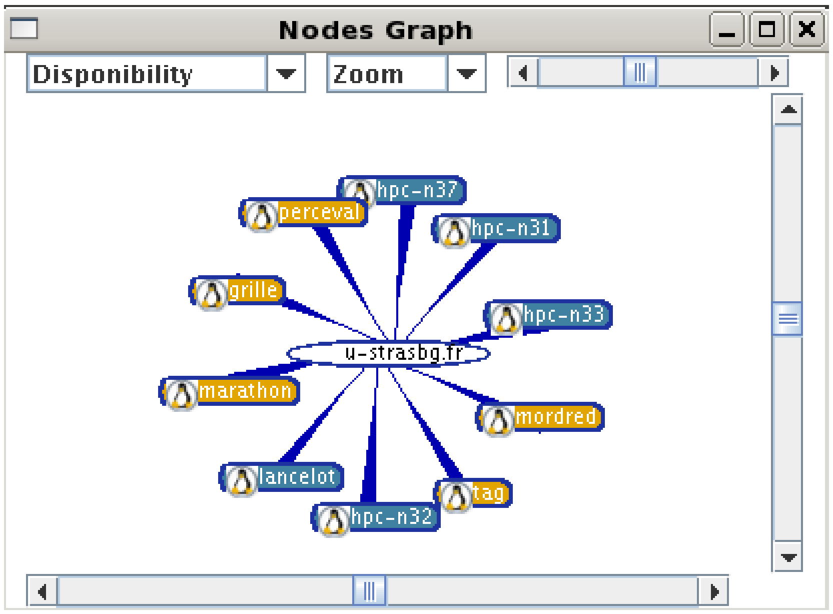

Figure 4.1: P2P-MPI Grid state visualization (taken from [32])

Within the P2P-MPI integration process the most important changes needed where in are the Grid construction and the communication logic between peers. These two issues are the main features of the F2F Computing framework.

The P2P-MPI consists of many services and command line scripts that the MPI program uses for starting up and submitting jobs. For example the File Transfer service for distributing your program and super node for easily finding peers. The submitting of jobs is made in the P2P-MPI with getting the list of peers from the super node and asking a few services if the peers are available and may take on new jobs. Similar functionality is already present in the F2F Computing framework, so the usages of these services from the P2P-MPI will be removed and be rewritten in F2F.

The F2F Computing framework contains a very good and simple Grid construction and job submission system. You just have to select your friends in the SIP Communicator and form a chat group with them. Then you can submit your job to their computers in this chat. After this the job is automatically distributed and executed.

Porting from the P2P-MPI I will remove the super node system for finding peers and the services related to submitting and distributing the jobs and let the F2F Computing framework do this for me.

P2P-MPI uses direct TCP connections for communication between peers, but F2F takes care of the communication between peers. So I replace the base communication logic of the P2P-MPI with the F2F Computing framework.

Another thing that I will rewrite is the initialization of MPI. The initialization of MPI sends a lot of complex messages between peers and uses the P2P-MPI services for asking if the peers are available. I replace the message sending calls and peer synchronization commands with F2F messages, although the main logic behind the initialization remains the same.

The syntax of MPI functions will remain almost the same as in P2P-MPI. The only different is that instead of static fields I use non-static. A static field means that there is only one instance of that object. For example if there are two MPI task running in the same SIP communicator (same JVM) and they would use the same static MPI classes, due to of the static class, they would share the same message buffers too. If we received a message we would not know which of the task should read it in. In P2P-MPI this is not a problem, as in P2P-MPI, each program is executed separately from the command line. However this is a problem in F2F, because all the task will run in the same SIP communicator (same JVM). To solve this I make all the static fields non-static, like `MPI -> MPI()` and `MPI.COMM_WORLD -> MPI().COMM_WORLD()`.

Another difference with P2P-MPI is that the main class of the program must extend class `MPITask` and the first run method is `runTask()` instead main like in P2P-MPI. `MPITask` has access functionality for MPI, like `MPI()` and it extends the usual F2F Computing frameworks main task.

4.4 Legal issues

One major problem we had with the integration of P2P-MPI into the framework was the licensing of the program. Namely, the F2F Computing framework is licensed under LGPL (Lesser General Public License), but the P2P-MPI is licensed under GPL (General Public License). So we can not publish the F2F-MPI together with F2F, as the GPL license is more restrictive then the LGPL.

The best solution would be if P2P-MPI would also be under the LGPL license, because we want a lot of F2F users and do not want to change its license to GPL. So we wrote a letter to the project advisor of the P2P-MPI Stéphane Genaud and talked about our problem. He replied that he was interested of providing the P2P-MPI as widely as possible and gave us a LGPL side license for P2P-MPI, so we could integrate the program to our systems, without a problem. For that wel thank Stéphane Genaud.

4.5 Features

Every software system has some features and F2F-MPI is no different. The main features of F2F-MPI come either from Java, the F2F framework or from MPI. The features of F2F-MPI are:

- **Your F2F-MPI programs are easily portable** – Your programs will be Java programs, so if we can run Java on one computer, then we can run the SIP Communicator with the F2F Computing framework that includes F2F-MPI on that computer.

- **You can port legacy MPI programs to Java and F2F-MPI** – Now F2F-MPI has the MPI commands implemented, in the porting process you just have to port the C or FORTRAN specific code to Java code and there will be already existing Java functions for your MPI commands.

- **Provides better communication between peers** – The F2F Computing framework provides communication between peers. The peers need not to be within the same network. The framework can make as well TCP connection as UDP hole punching, and if that does not work then there is always the instant messenger channel for peer communication. This is much better, as P2P-MPI that only supports TCP connection.

- **Easier Grid construction** – The Grid construction in F2F is simple. You must just collect some friends, which have F2F installed, in SIP Communicator and start a chat. Then you can submit your program in there and the Grid with all these friends in it, is automatically created. You need not know anything about underlying network and need not worry about firewalls or the IP addresses of your friends.

- **Provides a bigger variety of communication functions for F2F users** – In the standard of MPI there are defined a lot of functions for sending and receiving information in a collaboration environment. In the F2F-MPI these functions are implemented and if you decide to use MPI you can use them for writing your programs, instead of implementing similar ones yourself.

- **Possibility for peer fault recovery** – Like P2P-MPI the F2F-MPI supports also synchronized backup peers. This functionality means that you can have peer groups so that inside one group the contents of the peers is the same (synchronized). One of the peers is the master of the group and others peers outside the group communicate only with that peer. If the master of the group dies, then the peer with the lowest sequence number, in that group, is taken as the new master of the group and the process of the calculation must not be cancelled. If the last member of such a group dies, then the calculation can not continue and the process must still be cancelled. Synchronized backup peers will be made if in the initialization of the MPI in `MPI().Init(...)` when the number of peers provided (number of slave peers * `numOfJobsPerPeer`) is greater then the number of peers needed (`maxRank`).

- **Multiple tasks on one peer** – Allows you to say that you need to submit multiple tasks on one peer, so that the F2F-MPI thinks we have more peers. This feature is not made for performance gain. But main interest behind it is the testing of MPI programs in an environment where we have only a few computers. If you have only a few computers and will need to test an MPI application, with a lot of peers, then it is good if you can make it without much effort. For example you may submit within the same MPI task to each peer 5 tasks. Therefore if you have only 2 computers but the logic of ypur calculation needs at least 10, you can let the framework submit 5 tasks per peer to get 2*5=10 running tasks. It does not give a performance gain, but in testing your MPI job you can easily use fewer computers.

4.6 Chapter Summary

It will be good if we have some MPI capability in the F2F Computing framework. For this reason I integrated the pure Java MPI framework P2P-MPI into the F2F Computing framework. For the integration I refactored P2P-MPI to use the functionality of the F2F Computing framework whenever possible. The Features of the F2F-MPI are:

- Your F2F-MPI programs are easily portable
- You can port legacy MPI programs to Java and F2F-MPI
- Provides better communication between peers
- Easier Grid construction

- Provides a bigger variety of communication functions for F2F users
- Possibility for peer fault recovery
- Can run multiple tasks on one peer

Chapter 5

Implementation

5.1 Implemented commands

The F2F-MPI follows the MPI-1.2 standard of MPI. However, only a subset of commands is implemented. Here I will name all implemented commands together with a short command description. The parameters for the communication functions are all very similar, so I will first give list of all the common parameters and their description. If a parameter is different in a function, then that parameter description will be included.

buffer – An array, this is used for buffer. Used for sending and receiving the information in broadcast function. The receiving side of the broadcast must have the same sized buffer as the sender.

offset – Offset in the buffer. From which array element we start using the elements in the buffer.

count – How many buffer elements, starting from the count, the function can use (for sending them or inserting received results into them).

datatype – Of what type the objects are in the buffer (INT, DOUBLE. . .).

sendBuffer – An array, it contains the objects to be sent (same as `buffer`, but only for sending) .

sendOffset – Offset in the buffer, from which array element we start using the elements in the `sendBuffer` for sending (same as `offset`, but only for sending).

sendCount – How many buffer elements, starting from the count, the function can send (same as `count`, but only for sending).

sendType – Of what type the objects are in `sendBuffer` (INT, DOUBLE…) (same as `datatype`, but only for sending).

recvBuffer – An array, contains places for the elements we receive (same as `buffer`, but only for receiving).

recvOffset – Offset in the buffer. From which array element we start using the elements in the `recvBuffer` for receiving (same as `offset`, but only for receiving).

recvCount – How many buffer elements, starting from the count, the function can receive (same as `count`, but only for receiving).

recvType – Of what type the objects are in `recvBuffer` (INT, DOUBLE…) (same as datatype, but only for receiving).

tag – A unique tag on the message. To know if the send or received message is the one we wanted.

op – Says which function to use for reducing the sent data into one result object.

root – Shows if you are the master of the command (The one that sends out the broadcast message to others etc.).

displs – Says where the data for a specific peer is placed in the buffer, specified in the order of peers.

sendCount (array) – Combined with displs. Says how many objects will be sent to a specific peer.

recvCount (array) – Combined with displs. Says how many objects will be received from a specific peer.

Status – A return object. It says if the receiving was successful or not.

Request – A return object. With it, we can check if the receiving is done or not. If it is done then `recvBuffer` contains the result. Otherwise its content is unknown.

5.1.1 MPI().Init(...);

This command initializes the MPI. It must be called at the beginning of the program before other MPI commands are called (except `MPI().Initialized()`). This function may be called only once.

It sends the tasks to all other nodes and starts them. Then it synchronizes the node communication information, so the peers can send broadcast and other messages between them. [17]

```
public void Init([int maxRank[, int numOfJobsPerPeer]]);
```

The parameters this function takes are:

maxRank - An optional parameter, says how many peers we need in calculation (master + slaves). If it is 0, then the value is ignored. If there are more peers than `maxRank` then the remaining peers will be synchronized peers for backup.

numOfJobsPerPeer - An optional parameter, says how many slave processes we submit per peer. If it is 0, then the value is ignored. This value can be used to make virtual peers. If you have just a few computers in your disposal then the program handles one peer as it were `numOfJobsPerPeer` different peers.

Example usage: Let's assume we have a master computer and 5 slave computers.

```
MPI().Init();
```

Start MPI with all given peers (5) so that there is one slave process for each peer with no synchronized backup peers. This means we have one master and 5 slave peers calculating.

```
MPI().Init(4, 2);
```

This starts MPI with all given peers (5) so that there are 3 different slaves running. Each peer has 2 slave processes running. This means we have 5*2=10 slots for slave processes. The first 3 slots will go to main slaves and the remaining slots will be made to synchronized slaves, so that when one peer dies then there is another one to replace it (Figure 5.1).

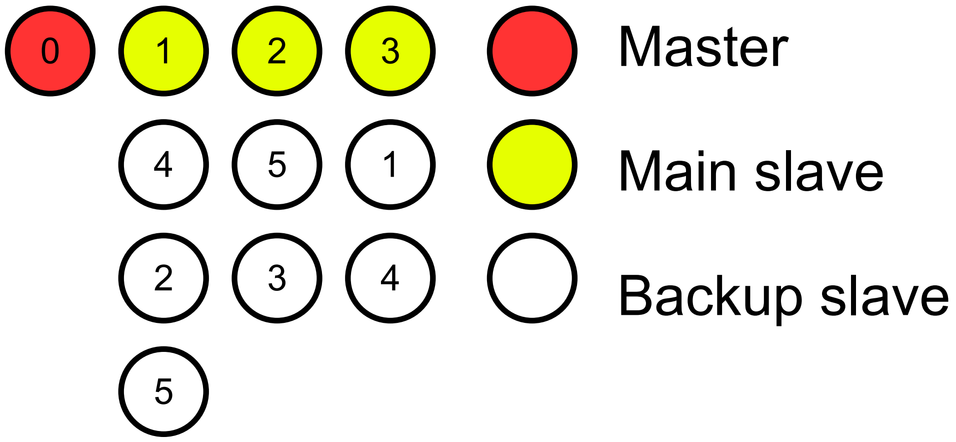

Figure 5.1: Illustrates the use computers in case of MPI().Init(4,2) with 5 slave computers. Each number represents one computer and the color represents its role in the calculation.

5.1.2 MPI().Initialized();

Says if `MPI().Init()` has already been called or not. This is the only command that may be called before `MPI().Init()`. [17]

```
public boolean Initialized ();
```

Returns if `MPI().Init()` has already been called or not.

5.1.3 MPI().Finalize();

Must be called at the end of MPI-s work, after this no more MPI commands may be called. [17]

```
public void Finalize();
```

The functions `MPI().Init()` and `MPI().Finalize()` must be called at the beginning and end of the programs, because the MPI specification needs them there. These functions can be done automatically in Java, but as they are in the MPI specification, they must be called explicitly.

5.1.4 MPI().Wtime();

Name comes from the term wall-clock time and means the current time in seconds in the local computer. The time is not synchronized with other peers because it is meant for time measurement in one computer. [27]

```
public double Wtime();
```

Example usage:

```
...
double start = MPI().Wtime();
...
The program code
...
double end = MPI().Wtime();
log.debug(‘‘Time spent : ‘‘+(start-end));
...
```

Gives how long it takes to run the program between start and end.

5.1.5 MPI().Get_processor_name();

Returns the names of the host computer and the id of the local peer. We use MPI as a Grid systems implementation and run it in Java, so the processors name would make no sense and we use the hostnames of the computer and the local peers id instead. [16]

```
public String Get_processor_name();
```

5.1.6 MPI().COMM_WORLD().Barrier();

Marks a barrier in the program. The program does not continue until all the peers have reached the barrier. If all peers have reached the barrier then they all are released at the same time to continue their work. [13]

```
public void Barrier();
```

This function can be useful to get all the peers at the same place to collect some results at the middle of the calculation.

5.1.7 MPI().COMM_WORLD().Size();

It says how many different peers are running in the job. [26]

```
public int Size();
```

5.1.8 MPI().COMM_WORLD().Rank();

Says which is your rank (order) number in the calculation. The number is between 0 and `MPI().COMM_WORLD().Size()`-1. If the rank number is 0, it is the master process. [26]

```
public int Rank();
```

Example usage of rank and size:

```
...
int size = MPI().COMM_WORLD().Size();
int rank = MPI().COMM_WORLD().Rank();
...
if (rank == 0) {
  for (int i = 0; i < size; i++) {
    ...
  }
  ...
} else {
  ...
}
...
```

In your MPI program you may need to use the functions rank and size. With size you can get how many processes are running and then do something for each process. For example send a message. As in the previous small example the most common use of rank is to determine if the running process is a master or a slave process.

5.1.9 MPI().COMM_WORLD().Send(...);

Sends a message from one peer to another. [24]

```
public int Send(Object sendBuffer, int offset, int count,
  Datatype datatype, int dest, int tag);
```

Unexplained parameter for this function is:

dest – Which peer receives the message.

Example usage:

```
int[] msg = {1,9,2,6,4,1};
MPI().COMM_WORLD().Send(msg, 2, 3, MPI.INT, 3, 9);
```

Send the numbers 2, 6 and 4 from the array to the node numbered 3. The node 3 must be waiting for a message with Recv and must have an array with 3 free slots for results. Node 3 must wait for this message with tag 9.

5.1.10 MPI().COMM_WORLD().Recv(...);

A blocking message receiving command. The program will not return until the requested message is received. [19]

```
public Status Recv(Object recvBuffer, int offset, int count,
  Datatype datatype, int src, int tag);
```

Unexplained parameter for this function is:

src – The sender of the message.

Example usage:

```
int[] buff = {2,9,4,6,5,1};
MPI().COMM_WORLD().Recv(buff, 1, 3, MPI.INT, 1, 9);
```

Receives 3 elements from node 1 and the message is with tag 9. The 3 received elements are copied in the buffer, starting from position 1. If the send is made from previous example and the sent elements are 2, 6 and 4 then the result buffer would be 2, 2, 6, 4, 5 and 1.

5.1.11 MPI().COMM_WORLD().Irecv(...);

A non-blocking message receiving command. After the `Irecv` is called the program must check if the message is already received or not from the retrun object `Status`. [18]

```
public Request Irecv(Object recvBuffer, int offset, int count,
  Datatype datatype, int src, int tag);
```

5.1.12 MPI().COMM_WORLD().Sendrecv(...);

Combines the send and receive operations into one command. This functions sends the `sendBuffer` to the `dest` and receives the object sent by `source` to `recvBuffer`. [25]

```
public Status Sendrecv(Object sendBuffer, int sendOffset,
  int sendCount, Datatype sendType, int dest, int sendTag,
  Object recvBuffer, int recvOffset, int recvCount,
  Datatype recvType, int source, int recvTag);
```

Unexplained parameter for this function is:

dest – Which peer receives the message sent by the function.

source – From which sender the function receives the message.

5.1.13 MPI().COMM_WORLD().Bcast(...);

Broadcast the message in buffer to all nodes. Every node must call this function with the same arguments, so the messages would be received correctly. [14]

```
public void Bcast(Object buffer, int offset, int count,
  Datatype datatype, int root);
```

Example usage:

```
Object[] buff = {data};
MPI().COMM_WORLD().Bcast (buff, 0, 1, MPI.OBJECT, 2);
```

Node 2 sends it buffer to all nodes. So if the runner of this code is peer 2 then its sends its buff content, but if the runner is another peer, like 1 for example, then that peer receives the message sent by 2 and copies it to its buff.

5.1.14 MPI().COMM_WORLD().Reduce(...):

Collects the sendBuffers from all the peers (including the receiver) and combines them with the operation op to one object and returns it to the peer that is said with the `root`. [20]

```
public void Reduce(Object sendBuffer, int sendOffset,
  Object recvBuffer, int recvOffset, int count,
  Datatype datatype, Op op, int root);
```

Example usage:

```
double[] myval = {value};
double[] val = new double[1];
MPI().COMM_WORLD().Reduce(myval, 0, val, 0, 1,
  MPI.DOUBLE, MPI.SUM, 0);
```

Collects the values calculated by each peer (inside an array with size 1) and then sums them together into the peer 0 (master peer).

5.1.15 MPI().COMM_WORLD().Allreduce(...);

Collects the `sendBuffers` from all the peers and combines them with the operation op to one object and returns it to all peers. Works similarly to `Reduce`, but all peers receive the results. [11]

```
public void Allreduce(Object sendBuffer, int sendOffset,
  Object recvBuffer, int recvOffset, int count,
  Datatype datatype, Op op);
```

5.1.16 MPI().COMM_WORLD().Gather(...);

Every peer sends its `sendBuffer` to the peer with rank `root`. The `root` peer collects the sent information to `recvBuffer` in the order of peers ranks. Each peers must send the same amount of data. [15]

```
public void Gather(Object sendBuffer, int sendOffset,
  int sendCount, Datatype sendType, Object recvBuffer,
  int recvOffset, int recvCount, Datatype recvType, int root);
```

5.1.17 MPI().COMM_WORLD().Gatherv(...);

Every peer sends its `sendBuffer` to the peer with rank `root`. The `root` peer collects the sent information to `recvBuffer` in the order of peer ranks. Each peer may send a different amount of data specified by `recvCount` and `displs`. [15]

```
public void Gatherv(Object sendBuffer, int sendOffset,
  int sendCount, Datatype sendType, Object recvBuffer,
  int recvOffset, int[] recvCount, int[] displs,
  Datatype recvType, int root);
```

5.1.18 MPI().COMM_WORLD().Allgather(...);

Every peer sends its information to each other peer. Each peer must send the same amount of data. When this command has finished done then all the peers have the same information. It is like `Gather`, but each peer receives the information. [10]

```
public void Allgather(Object sendBuffer, int sendOffset,
  int sendCount, Datatype sendType, Object recvBuffer,
  int recvOffset, int recvCount, Datatype recvType);
```

5.1.19 MPI().COMM_WORLD().Allgatherv(...);

Every peer sends its information to each other peer. Each peer may send a different amount of data. I this command is finished then all the peers have the same information. It is like `Gatherv`, but each peer receives the information. [10]

```
public void Allgatherv(Object sendBuffer, int sendOffset,
  int sendCount, Datatype sendType, Object recvBuffer,
  int recvOffset, int[] recvCount, int[] displs,
  Datatype recvType);
```

5.1.20 MPI().COMM_WORLD().Alltoall(...);

Every process sends distinct data to each other process. Each peer must send the same amount of data. [12]

```
public void Alltoall(Object sendBuffer, int sendOffset,
  int sendCount, Datatype sendType, Object recvBuffer,
  int recvOffset, int recvCount, Datatype recvType);
```

Example usage: Lets take we have 3 peers and each peers has 3 values to send to each other one.

```
int[]out = {2,5,6};//peer 1
int[]out = {4,7,8};//peer 2
int[]out = {7,6,5};//peer 3
int[] in = new int[3];
MPI().COMM_WORLD().Alltoall(out, 0, 1, MPI.INT, in, 0, 1, MPI.INT);
```

The result in output buffer will be:

Peer 1: 2, 4, 7

Peer 2: 5, 7, 6

Peer 3: 6, 8, 5

5.1.21 MPI().COMM_WORLD().Alltoallv(...);

Every process sends distinct data to each other process. Each peer may send a different amount of data. This function is like `Alltoall`, but the amount sent may differ per peer. [12]

```
public void Alltoallv(Object sendBuffer, int sendOffset,
  int[] sendCount, int[] sdispls, Datatype sendType,
  Object recvBuffer, int recvOffset, int[] recvCount,
  int[] rdispls, Datatype recvType);
```

Unexplained parameter for this function is:

sdispls – Says where the data for a specific peer is placed in the sendBuffer for sending.

rdispls – Says where the data from a specific peer should be placed in the recvBuffer in receiving.

5.1.22 MPI().COMM_WORLD().Scatter(...);

The peer with the number `root` will send its `sendBuffer` to all other peers in equal chunks. So that each peer receives one part of the data. [23]

```
public void Scatter(Object sendBuffer, int sendOffset,
  int sendCount, Datatype sendType, Object recvBuffer,
  int recvOffset, int recvCount, Datatype recvType, int root);
```

Example: We have one master and two peers.

```
int[] cor={4,2,6,7,4,5};
int[] buf=new int[2];
MPI().COMM_WORLD().Scatter(cor, 0, 2, MPI.INT, buf, 0, 2, MPI.INT, 0);
```

When the master executes this command then it will scatter the inside of cor to all the peers so each peer receives 2 elements of data. The bufs in each peer will be:

Master: 4, 2
Peer 1: 6, 7
Peer 2: 4, 5

5.1.23 MPI().COMM_WORLD().Scatterv(...);

The peer with the number `root` will send its `sendBuffer` to all other peers in predefined chunks. So that each peer receives one part of the data. [23]

```
public void Scatterv(Object sendBuffer, int sendOffset,
  int[] sendCount, int[] displs, Datatype sendType,
  Object recvBuffer, int recvOffset, int recvCount,
  Datatype recvType, int root);
```

5.1.24 MPI().COMM_WORLD().Reduce_scatter(...);

This function is a combination of `reduce` and `scatterv`. The function reduces the information sent by all peers with operation `op` and then sends the results again to everyone. As a result, each peer gets the number of results defined in `recvCount`. [21]

```
public void Reduce_scatter(Object sendBuffer, int sendOffset,
  Object recvBuffer, int recvOffset, int[] recvCount,
  Datatype datatype, Op op);
```

5.1.25 MPI().COMM_WORLD().Scan(...);

A reduction of previous peers so that each peer receives the reduced data of previous peers. For example peer 4 receives the reduced data of peers 0, 1, 2, 3 and 4. For the reduction the operation `op` is used. [22]

```
public void Scan(Object sendBuffer, int sendOffset,
  Object recvBuffer, int recvOffset, int count,
  Datatype datatype, Op op);
```

5.2 Disadvantages of my Implementation

In the process of integrating the P2P-MPI into the F2F Computing framework I found some issues that had no elegant way of implementation.

One of them is the termination of programs. For example a peer crashes and there is no way to continue the work. The question is how can we stop that peer and how can we stop all other peers in this task.

If we had a Java standalone program we could use `System.exit()` for stopping the entire program, in case we detect an error or receive a message about it. But this can not be done in F2F. It would also stop the SIP Communicator with all other running tasks as well. Another solution is to kill the threads of the task. But killing threads, is not a good practice in Java [5].

As we can not just close the program the solution was, that if we get a fatal error or an important peer dies, we put up a flag indicating the MPI program should be terminated. Now all the MPI commands will throw the `MPITerminateException`. When this exception is thrown, the user can let it propagate and the program terminates on its own or can catch it and do something that does not involve MPI, before exiting. I such way the program will stop if all the threads in it try to use some MPI command and get the exception.

The other disadvantage in F2F-MPI compared just to the F2F Computing framework is the way to start a program. If you start a node in MPI you will start it with the same class and the same function as you started the master, but F2F supports starting slave peers from different classes than the master was. This feature is lost in F2F-MPI.

However, the biggest disadvantage of MPI is that it is defined for a procedure oriented paradigm and not for the Object-Oriented paradigm that is be more suitable for Java. For example one deficiency in the specification is that the message sending functions have to say what type of message they are sending. But in an Object-Oriented paradigm I can just send the message as an object and can cast the object to the right type after receiving it. In Java every variable is an Object so we would not have to write the type of the Object we send, because it can be asked automatically.

For making the MPI a little more Object-Oriented, there are extra implementations of MPI commands in F2F-MPI, where there is no message type needed. The message typing is selected in the background by casting the given object to the right type. These functions syntaxes are the same as described in 5.1, except there is no Datatype parameter in the syntax.

5.3 Problems in Implementation

The biggest problem that I encountered was with the F2F basic message sending logic. In the local network F2F created a fast TCP connection between the peers and it sent serialized messages on that connection. For serialization and deserialization it used a custom serialization class, as it needs to use job specific class loaders. However, sometimes between some computers the custom serialization does not work properly and the TCP connection in Java broke between these peers. This error occurred very rarely and only in some specific cases with complex data structures.

The solution is that F2F-MPI serializes the messages to byte arrays, before sending them with F2F basic message sending logic. So the serialization process of the message remains entirely custom logic free and we do not get the connection reset.

5.4 Conclusions

Overall using the MPI in F2F is very useful. It gives extended functionality like the Barrier function for synchronization or the broadcast functions for message sending. You can even port other MPI programs to F2F more easily, as the logic behind the program remains the same.

For example, now we can port old MPI programs from the DOUG (Domain Decomposition on Unstructured Grids) [42] system into our framework and use them there. Or we could create a system like DOUG into the F2F Computing framework. Then we could solve bigger time consuming mathematic calculations with our system more easily, without writing a lot of code for it.

MPI has a lot of functions defined in its specification, but in the F2F-MPI we do not have all the MPI functionality implemented. We have only the commonly used commands. The other functions should be implemented when there will be a need for them.

The biggest disadvantage in MPI is that it is defined as a procedure oriented paradigm standard. However, F2F is written in Java which is defined within Object-Oriented paradigm. It would be better if MPI where Object-Oriented and could use Object-Oriented features. It would be good if we had a more Java and F2F like system for supporting Grid computing.

For example one issue that looks bad in F2F-MPI is the syntax of send functions. They all have the type of the message sent in it, but in Java it is not necessary. I can just ask the class what type it was and then later cast it to that.

The `MPI().Init(...)` and `MPI().Finalize()` functions would not be needed in F2F too, but they are here because of the MPI standard. This functionality can be done automatically in the background by the framework and in an Object-Oriented manner.

5.5 Chapter Summary

Using the MPI on F2F is good, because it gives us extended functionality. In the F2F-MPI we have the main MPI commands implemented. For example the Bcast or the Barrier functions. With them it is much easier to write bigger programs in F2F without having to think about how to write more complex information sending functions. There were some difficulties with the porting, but they were overcome.

Chapter 6

Examples

6.1 How to run F2F-MPI

The running of MPI programs is exactly the same as running a usual F2F program. A long guide can be found at Keio Kraaner's work on F2F in chapter "F2F Computing framework in practice" [43].

The main difference in using MPI and usual F2F program is that a F2F Java programs must extend the class `Task`, but a F2F-MPI program must extend the class `MPITask`. `MPITask` has extra functionality for accessing MPI commands and it extends the usual F2F frameworks main class `Task`. The most important function in `MPITask` the programmer must use is `MPI()`. It gives a handler to MPI and other MPI functions can be called from there.

For submitting your program, you must package your program to a jar file. This file must contain your Java files and manifest file. The main class for the task can be specified with in the F2F-MasterTask attribute. For example:

```
F2F-MasterTask: ee.ut.f2f.mpi.examples.games.Moving.java
```

Like in a usual F2F program the framework will execute the `runTask()` function from the main class. Now in F2F-MPI the framework will distribute your program to all your selected friends and submit the task to them.

6.2 Example program

Here I will give an example of F2F-MPI. The goal of this example is to show how to use the F2F-MPI commands.

Because everyone is bored of the endless π examples, I made a pseudo game as my example. It runs on MPI and uses the MPI commands for communication. The structure of the game is simple. The master peer holds the role of the Server and the slave peers are Players. At the beginning of the game the Server gives each player a coordinate. In each turn a player moves around the table and when it reaches a square where another Player resides, then the Server will choose one of them and throw it out of the game. The game ends when there is only one Player remaining or the turn limit is exceeded.

```java
package ee.ut.f2f.mpi.examples.games;
import java.util.Random;
import ee.ut.f2f.core.mpi.MPI;
import ee.ut.f2f.core.mpi.MPITask;
public class Moving extends MPITask {
  private static final int MAX_STEPS = 30;
  private static final int SIZE = 3;
  public void runTask() {
    getMPIDebug().println("Starting MPI commands example game");
    int rank, size;
    MPI().Init();
    size = MPI().COMM_WORLD().Size();
    rank = MPI().COMM_WORLD().Rank();
    Object[] buf = new Object[1];
    Random rnd = new Random();
    int[] dead = new int[size];
    for (int i = 0; i < size; i++) {
      dead[i] = 0;
    }
    int h = 0;
    int alive = size - 1;
    if (rank == 0) {// Master (Server)
      getMPIDebug().println("Number of players : " + (size - 1));
      int[][] cor = new int[size][2];
      for (int i = 0; i < size; i++) {
        cor[i][0] = rnd.nextInt(SIZE);
        cor[i][1] = rnd.nextInt(SIZE);
        getMPIDebug().println("Player " + i + " kordinates: " + cor[i][0] + " " +
            cor[i][1]);
      }
      MPI().COMM_WORLD().Scatter(cor, 0, 1, MPI.OBJECT, buf, 0, 1, MPI.OBJECT, 0)
          ;
      while (alive > 1 && h < MAX_STEPS) {
        h++;
        getMPIDebug().println("Starting Round " + h + " remaining players " +
            alive);
        MPI().COMM_WORLD().Gather(buf, 0, 1, MPI.OBJECT, cor, 0, 1, MPI.OBJECT,
            0);
        for (int i = 1; i < size; i++) {
          if (dead[i] == 0) {
            getMPIDebug().println("Player " + i + " stept to kordinates: " + cor[
                i][0] + " " + cor[i][1]);
          }
        }
        for (int i = 1; i < size; i++) {
          for (int j = i + 1; j < size; j++) {
            if (dead[i] == 0 && dead[j] == 0 && cor[i][0] == cor[j][0] && cor[i][1] == cor[j][1]) {
              getMPIDebug().println("Players " + i + " and " + j + " are on same square, one of them must go");
              dead[(Math.random() > 0.5 ? i : j)] = 1;
            }
          }
        }
        MPI().COMM_WORLD().Bcast(dead, 0, dead.length, MPI.INT, 0);
        alive = 0;
        for (int i = 1; i < size; i++) {
          if (dead[i] == 0) {
            alive += 1;
          }
        }
      }
      getMPIDebug().println("The game is over");
      for (int i = 1; i < size; i++) {
        if (dead[i] == 0) {
          getMPIDebug().println("Player " + i + " survived");
        }
      }
    } else {// Slaves (Players)
      MPI().COMM_WORLD().Scatter(null, 0, 1, MPI.OBJECT, buf, 0, 1, MPI.OBJECT, 0);
      while (alive > 1 && h < MAX_STEPS) {
        h++;
        if (dead[rank] == 0) {
          int[] mycor = (int[]) buf[0];
          getMPIDebug().println("Round " + h + " Player " + rank + " kordinates: " + mycor[0] + " " + mycor[1]);
          for (int i = 0; i < 2; i++) {
            mycor[i] += rnd.nextInt(3) - 1;
            if (mycor[i] < 0)
              mycor[i] = 0;
            if (mycor[i] >= SIZE)
              mycor[i] = SIZE - 1;
          }
        }
        MPI().COMM_WORLD().Gather(buf, 0, 1, MPI.OBJECT, null, 0, 1, MPI.OBJECT, 0);
        int state = dead[rank];
        MPI().COMM_WORLD().Bcast(dead, 0, dead.length, MPI.INT, 0);
        if (state == 0 && dead[rank] == 1) {
          getMPIDebug().println("I lost");
        }
        alive = 0;
        for (int i = 1; i < size; i++) {
          if (dead[i] == 0) {
            alive += 1;
          }
        }
```

```
89           if (alive == 1) {
90             getMPIDebug().println("Round " + h + " Game over");
91             if (dead[rank] == 0) {
92               getMPIDebug().println("I am the Winner");
93             }
94           }
95         }
96       }
97       MPI().Finalize();
98     }
99   }
```

Now I will give a detailed explanation how this program works.

Logically the running of the code splits into two slightly different ways. First I describe the master part and then there comes the slave part of execution.

If you have submitted the program into F2F then the MPI is run as a Master task. First the `runTask()` function is executed. If the program reaches line 11 then inside `MPI.Init()` the MPI will initializes the slave tasks and in each slave peer the `runTask()` function is executed. Next the program prepares for starting the game, it looks how many Players (slaves) we have got and initialize the default values.

Next the code splits to two. First the master part:

In the master node the first thing to do is to initialize the beginning coordinates of all Players. This is done by giving each Player some random coordinates (lines 24-29). Next the master must say to each Player what their coordinates are. For this in line 30 we use the first MPI command - `MPI().COMM_WORLD().Scatter(...)`. If this command is executed by the master node then it will send the coordinates in `cor` to each of the Players. But each Player will get only its own coordinates. So they do not know where the other Players reside.

Next there is a loop that last until only one player is left or the turn limit is reached (line 31). One loop cycle is one turn. In each turn the Players move. At the beginning of the turn the master asks, all the players, for their new coordinates with `MPI().COMM_WORLD().Gather(...)` at line 34. This function does the opposite to Scatter, it collects all the information from slaves and returns them to master. Now the games prints out to the MPIDebug console where the Players are at the moment (line 35-39). Next the master will look if two Players are on the same square, if they are the master selects one of them and puts it dead. After this the master sends the results with broadcast to all the nodes - `MPI().COMM_WORLD().Bcast(...)` at line 48.

If there is only one player remaining (lines 49-54) or the turn limit is exceeded the game will close and the results will be printed (line 56-61). If the game is not over the program will return to line 32 to ask the Players for new coordinates.

The Player part of the program is the following:

At line 63 the Players will ask the master for their beginning coordinates with `MPI().COMM_WORLD().Scatter(...)`. After this they will start a loop that goes on, until the end of the game, like the master. If the Player is not dead then in lines 66-76 it will get itself some new coordinates and at line 77 it will send the new coordinates to the server with `MPI().COMM_WORLD().Gather(...)`. For a slave node the Gather function sends the input buffer to the master.

Next the Player asks the Server for all dead Players with `MPI().COMM_WORLD().Bcast(...)`. If the player detects that itself has entered the dead Players list (lines 80-82) then that Player will not generate new coordinates for itself, but will wait until the game is over. At lines 83-94 the Player will look at the dead Players list and count if the game is over and if it is the only survivor. Now if there is only one Player remaining or the turn limit is over then the Players will end their work.

Finally at the end of program, when master and slaves have ended their work, they both call the `MPI().Finalize()` in line 97. The result of this program will be the number of the peer who won the game.

6.3 More example program references

There are a lot more examples of F2F-MPI in SVN [4]. The module inside SVN, where the examples reside, is java/MPI/. There is an Ant script for packaging the examples into a jar. The Java code is in the src/ directory. The examples are ported from the P2P-MPI examples.

In the examples there is a π calculation example that uses F2F-MPI. The main class is in file `ee.ut.f2f.mpi.examples.pi.Pi.java`. In the same directory there is a pure F2F version of the same example too – `ee.ut.f2f.mpi.examples.pi.F2FPi.java`.

Some other examples are:

`ee.ut.f2f.mpi.examples.Hostname.java` – All peers execute `MPI().Get_processor_name()` and sent their result to the master task, which prints them out.

`ee.ut.f2f.mpi.examples.fileTransfer.RemoteDataTest.java` – Reads a data file from the class path and then sends it to all the nodes, where they check if they have exactly the same version of the file.

`ee.ut.f2f.mpi.examples.mpiCommTest.*` – A lot of MPI test programs. These programs test the different MPI commands.

`ee.ut.f2f.mpi.examples.graph.Floyd.java` – Finds the shortest path with Floyd-Warshall's algorithm.

`ee.ut.f2f.mpi.examples.games.Moving.java` – The Example program that was described in the last section.

6.4 Chapter Summary

In this chapter I introduced some MPI example programs that run on the F2F Computing framework. They can be found at F2F SVN repository at [4]. The running of the samples is the same as running any other program in the F2F Computing framework.

Chapter 7

Summary

The goal of this master's work is to get MPI support into our F2F Computing framework. Additionally I give some example programs on how to use this new MPI layout and write this thesis about it.

The problem currently with the F2F Computing framework is that it has just the basic commands for communication and not very much advanced functionality. It supports easy creation of Grids and point to point communication between single peers, but for example it does not support larger cooperative functions like broadcasting to all calculating computers. This is where MPI comes in.

MPI is the *de facto* standard in distributed and parallel software development. It specifies a lot of different functions that are useful in writing a distributed program. If we have an MPI implementation in the F2F Computing framework then we can take advantages of it.

I had to choose a way to get MPI support to the F2F Computing framework. Instead of writing a new MPI system from scratch I decided to integrate an existing MPI system into the F2F Computing framework – the pure Java MPI – P2P-MPI [32].

The main issue in the integration of P2P-MPI that I changed were the basic logic behind the MPI, like creating the Grid and the simplistic message passing between nodes. This logic already exists in the F2F Computing framework, so I rewrote these parts of P2P-MPI to use our F2F logic instead. If we have basic communication working between peers then the more complex MPI functions can be built.

Additionally to the integration of MPI I give some example programs to show how F2F-MPI works and how it can be used. For this I took the example programs from P2P-MPI and modified them so that they will work also in F2F-MPI. The issues in the MPI programs that I had to change were the MPI calls. I changed them from P2P-MPI to F2F-MPI calls.

The MPI in F2F is not a fully implemented MPI. It follows the MPI-1.2 standard and has the common MPI commands implemented, but some of the commands are still missing.

Now that the F2F Computing framework has the basic MPI support we can write new programs on it using the MPI commands. We can also port other existing MPI programs, like the MPI programs from DOUG system, into F2F more easily, because now we have the corresponding MPI commands.

The goal of giving the F2F Computing framework MPI support has been reached, but there are certain issues that can be improved. For example MPI is an old standard and it is defined for a procedure oriented paradigm and not for the Object-Oriented paradigm. So it would be good if we had a more Object-Oriented support for distributed computing in our framework. Now that we have MPI on the framework, we can even think of creating some system for solving time consuming mathematic calculations in the framework.

All the work on Java programs I made in this work is available from the F2F homepages http://f2f.ulno.net SVN repository.

MPI rakenduste portimine F2F Gridile

Magistritöö

Andres Luuk

Resümee

Töö eesmärgiks oli täiendada F2F raamistikku MPI (*Message Passing Interface* – teadete edastamise liides) toega, mis võimaldab seal lihtsamalt realiseerida paralleelrakendusi.

F2F raamistik on inglise keeles pikalt välja kirjutatult *Friend-to-Freind Computing framework* ja eesti keelde otsetõlkes on see Sõbralt-Sõbrale raamistik. F2F raamistiku arendatakse Tartu Ülikooli Arvutiteaduse instituudi hajusarvutuste grupis. Selle eesmärk on töölaua võrkraalimise (GRID) programmide realiseerimise ja käivitamise võimalikult lihtsaks tegemine sõltumata keskkonnast (Windows, Linux, Sun). F2F raamistik on kirjutatud Java keeles ja töötab *SIP communicatori* pistikprogrammina.

MPI puhul on tegemist liidesega, mis pärineb juba 90-nendate algusest. Antud liides on aja jooksul kujunenud harjusarvutuse *de facto* standardiks ja seda kasutatakse väga laialdaselt. MPI on tavaliselt realiseeritud C, C++ või FORTRANi jaoks ja Java jaoks sellel palju realisatsioone ei ole.

F2F raamistikule MPI toe lisamiseks võtsin ma aluseks P2P-MPI projekti. P2P-MPI on Javas kirjutatud MPI raamistik. Töö põhitulem on P2P-MPI integreerimine F2F raamistikku. Selle saavutmiseks tuli P2P-MPI alusloogika ümber kirjutada. Nimelt tuli see asendada F2F raamistiku poolt pakutava, juba olemasoleva funktsionaalsusega. Lõppeeesmärk oli MPI täielik integratsioon F2F

raamistikku, et MPI programme saaks kasutada nagu tavalisi F2F programme.

MPI tugi F2F raamistikus on vajalik, sest leidub väga palju MPI programme. Ka Tartu ülikoolil leidub vanu MPI pärandrakendusi. Tänu F2F raamistiku MPI toele on neid programme lihtsam kohandada F2F Gridile.

Töö käigus said ka kõik P2P-MPI näiteprogrammid F2F raamistikule kohandatud. Seega on lisaks MPI toele meil nüüd ka hulgaliselt näiteprogramme MPI kasutamise kohta, mis F2F raamistikus töötavad.

Kõik tehtud programmid on kättesaadavad F2F kodulehelt `http://f2f.ulno.net` SVN hoidlast.

Bibliography

[1] Automatic Domain Decomposition on Unstructured Grids DOUG. Available from: http://dougdevel.org/.

[2] Distributed Systems Group site. Available from: http://ds.cs.ut.ee/.

[3] F2F home page : Install. Available from: http://code.google.com/p/spontaneous-desktop-grid/wiki/DevelopmentSetup.

[4] F2F home page : SVN. Available from: http://code.google.com/p/spontaneous-desktop-grid/source/checkout.

[5] Java Thread Primitive Deprecation. Available from: http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html.

[6] LAM/MPI Parallel Computing. Available from: http://www.lam-mpi.org/.

[7] Message Passing Interface - Wikipedia, the free encyclopedia. Available from: http://en.wikipedia.org/wiki/Message_Passing_Interface.

[8] Message Passing Interface (MPI) Forum Home Page. Available from: http://www.mpi-forum.org/index.html.

[9] Microsoft Message Passing Interface. Available from: http://go.microsoft.com/fwlink/?LinkId=55930.

[10] MPI - Standard - ALLGATHER. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node73.html.

[11] MPI - Standard - ALLREDUCE. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node82.html.

[12] MPI - Standard - ALLTOALL. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node75.html.

[13] MPI - Standard - BARRIER. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node66.html.

[14] MPI - Standard - BCAST. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node67.html.

[15] MPI - Standard - GATHER. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node69.html.

[16] MPI - Standard - GET PROCESSOR NAME. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node143.html.

[17] MPI - Standard - INIT/FINALIZE. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node151.html.

[18] MPI - Standard - IRECV. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node46.html.

[19] MPI - Standard - RECV. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node34.html.

[20] MPI - Standard - REDUCE. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node77.html.

[21] MPI - Standard - REDUCE SCATTER . Available from: http://www.mpi-forum.org/docs/mpi-11-html/node83.html.

[22] MPI - Standard - SCAN. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node84.html.

[23] MPI - Standard - SCATTER. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node71.html.

[24] MPI - Standard - SEND. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node31.html.

[25] MPI - Standard - SENDRECV. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node52.html.

[26] MPI - Standard - SIZE/RANK. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node101.html.

[27] MPI - Standard - WTIME. Available from: http://www.mpi-forum.org/docs/mpi-11-html/node150.html.

[28] MPI for Python. Available from: http://mpi4py.scipy.org/.

[29] MPICH-A Portable Implementation of MPI. Available from: http://www-unix.mcs.anl.gov/mpi/mpich1/.

[30] mpiJava Home Page. Available from: http://www.hpjava.org/mpiJava.html.

[31] Open MPI: Open Source High Performance Computing. Available from: http://www.open-mpi.org/.

[32] P2P-MPI Home Page. Available from: http://www.p2pmpi.org/.

[33] Pure Mpi.NET. Available from: http://www.purempi.net/.

[34] Pypar - Parallel Programming in the spirit of Python! Available from: http://datamining.anu.edu.au/~ole/pypar/.

[35] ScientificPython. Available from: http://sourcesup.cru.fr/projects/scientific-py/.

[36] SIP communicator. Available from: http://sip-communicator.org/.

[37] Windows Compute Cluster Server. Available from: http://www.microsoft.com/windowsserver2003/ccs/default.aspx.

[38] M. Baker and B. Carpenter. Thoughts on the structure of an MPJ reference implementation. Available from: http://www.npac.syr.edu/projects/pcrc/HPJava/mpiJava.html.

[39] M. Baker, B. Carpenter, G. Fox, S.H. Ko, and S. Lim. mpiJava: An Object-Oriented Java interface to MPI. *International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP*, 1999.

[40] E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. *Proceedings, 11th European PVM/MPI Users Group Meeting*, pages 97–104, 2004.

[41] W. Gropp and E. Lusk. Users Guide for mpich, a Portable Implementation of MPI. *Argonne National Laboratory*, 1994. Available from: http://www.es.embnet.org/Doc/Computing/mpi/userguide.ps.

[42] M.J. Hagger. Automatic Domain Decomposition on Unstructured Grids DOUG. 1998. Available from: ftp://ftp.maths.bath.ac.uk/pub/preprints/maths9706.ps.gz.

[43] Keio Kraaner. F2F Computing. 2008. Available from: http://code.google.com/p/spontaneous-desktop-grid/.

[44] A. Lind. NAT Traversal in P2P systems in Java. 2008. Available from: http://code.google.com/p/spontaneous-desktop-grid/.

[45] Steven Huss-Lederman David Walker Jack Dongarra Marc Snir, Steve Otto. *MPI: The Complete Reference*. The MIT Press, 1995. Available from: http://www.netlib.org/utk/papers/mpi-book/mpi-book.html.

[46] P. Miller. pyMPI–An introduction to parallel Python using MPI. *Livermore National Laboratories*, 11, 2002. Available from: https://computing.llnl.gov/code/pdf/pyMPI.pdf.

[47] J.M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. *Proceedings, 10th European PVM/MPI Users Group Meeting*, pages 379–387, 2003.

[48] Sarah Healy Timothy H. Kaiser, Leesa Brieger. MYMPI-MPI programming in Python. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 458–464. Available from: http://iec.cugb.edu.cn/WorldComp2006/PDP5055.pdf.

Appendix A

F2F framework

The F2F Computing frameworks homepage can be found at:

http://f2f.ulno.net

There you can find out more information about F2F. There is also a detailed guide how to set up the F2F Computing framework in your own computer:

http://code.google.com/p/spontaneous-desktop-grid/wiki/DevelopmentSetup

The code of MPI is included in the F2F Computing framework (inside F2F, in the directory F2F\src\ee\ut\f2f\core\mpi\...). The code for F2F (including the F2F-MPI) can be found in its SVN repository:

http://code.google.com/p/spontaneous-desktop-grid/source/checkout

Appendix B

MPI Example programs

If the F2F is working in your computer then you can take the MPI examples from the SVN:

http://code.google.com/p/spontaneous-desktop-grid/source/checkout

(http://spontaneous-desktop-grid.googlecode.com/svn/java/MPI/)

After checking out all the examples they can be compiled with ant. If you would like to change the class that the F2F Computing framework suggests for running, you can to it by changing the class name in conf\MANIFEST.MF. In file conf\mainclasses.txt there is a list of example MPI programs that are included in the jar file.

Appendix C

Timeline

At the beginning 5% of time took the choosing of the topic and making myself acquainted with it: December 2007 and January 2008.

About 55 % of the time took the programming. The integration of P2P-MPI, the importing of examples and the debugging that everything works. Mostly from January to March.

The remaining 40% of the time was spent on writing this thesis. From March to May.

Index