

Fault Tolerant Parallel Programming

Individual Project Report

Karl Orbell (kao98@doc.ic.ac.uk) (MEng4)

Project Supervisor: Dr W. Knottenbelt (wjk@doc.ic.ac.uk)
Second Marker: Dr P. Kelly (phjk@doc.ic.ac.uk)

June 2002

Department of Computing
Imperial College of Science Technology & Medicine
University of London

Abstract

Many applications in Computer Science take a long time to execute, due to the large quantity of computation they require. Parallel programming techniques are often used to speed up the execution of these applications by using multiple processors simultaneously to gain an increase in the speed of execution. However, with increased numbers of machines comes an inherent increased risk of application failure. Many applications are so long running on such a large number of nodes that they are unlikely to ever complete in one go without any node failure. Because of this, some level of fault tolerance is required in these applications, so that in the event of a failure, a minimal amount of work is lost.

In serially executing code, fault tolerance is generally implemented using checkpointing and rollback systems. Very few parallel implementations of checkpointing exist, and the ones that do are not entirely satisfactory for all circumstances.

Most parallel programming is done using message passing communication libraries such as MPI and PVM. At present these libraries have no real fault tolerance built-in. This project aims to develop a fault-tolerant parallel programming library, that can be used to minimise the loss incurred by node failure in parallel applications.

Acknowledgements

Throughout the development of this project I have been given invaluable help and support by many people, here I would like to thank them.

My supervisor, Will Knottenbelt, has gone above and beyond the call of duty, always ready to listen to my rants and advise me on the best course of action. Thank you very much for your frank, clear, prompt and helpful comments, and most of all for calming me down all those times when I lost my head.

Paul Kelly, my second marker, has also been very helpful to me. Always making sure that I knew exactly what I was trying to achieve and what I needed to do to achieve it. Thank you.

Thanks goes to William Dieter (author of `libtckpt` [20]) and Victor Zandy (author of `ckpt` [19]) for extensive help via email over many check-pointing issues I encountered. It is reassuring to see that people can still take time to help students in far away places.

Special thanks goes to Charlee Brown, my love, for giving me purpose and determination to succeed (not to mention proof reading this entire report!); Andy Shuttlewood, for life saving help and insight at all times of day and night; Andy Ferrier for various \LaTeX hints and tips; and Dhruv Pandya for constantly keeping me on my toes and not letting me give up, ever.

Lastly to all my friends and family, who have put up with me and my stress for far too long. Thank you all.

Contents

1	Introduction	9
1.1	The Problem	9
1.2	Aims and Objectives	10
1.3	Assumed Knowledge	13
1.4	Report Outline	13
1.5	Project Website & Code Usage	14
2	Background Issues & Related Work	17
2.1	Parallel Programming	17
2.2	MPI and PVM	18
2.3	Parallel System Failure	20
2.4	Checkpointing	21
2.5	Process Migration	24
2.6	Heterogeneity	26
3	The Software - FTMPI	29
3.1	Requirements	29
3.2	Design	32
3.2.1	MPI System Limitations	32
3.2.2	System Startup	34
3.2.3	Message Passing System	37
3.2.4	Heartbeat System	39

3.2.5	Checkpointing & Restoration System	40
3.3	Implementation	43
3.3.1	General Structure	43
3.3.2	Communicator System	46
3.3.3	Message Passing System	47
3.3.4	Heartbeats	49
3.3.5	Collective Operations	50
3.3.6	Checkpointing System	51
3.4	Testing	58
3.4.1	Test Programs	58
3.4.2	Tests	61
4	Evaluation	75
4.1	Overview	75
4.2	System Performance	75
4.2.1	Raw MPI Performance	76
4.2.2	Checkpointing Performance	78
5	Conclusions & Future Work	91
5.1	General Appraisal	91
5.2	Portability	94
5.3	Future Work	95
5.3.1	MPI Implementation Extensions	95
5.3.2	Checkpointing Improvements	96
5.4	Problems, Bugs and Omissions	97

Chapter 1

Introduction

1.1 The Problem

For years scientists have been using computers to perform extremely complex and time consuming work. Their work entails the use of very intensive algorithms, that require immense computing power to complete. This computing power has been increasing for many years, with the field of computing continually searching for new ways to increase the power, in order to speed up the work of the scientists. Even though individual processors have been getting faster and faster over the years, the main developments in the field of high-throughput computing have been in the area of parallelization. Superficially, parallelization is making a process take place faster by dividing it in to separate tasks and executing them simultaneously, effectively making use of the old adage “Two heads are better than one”. Extending this to one thousand heads, or even a million, is often substantially more useful than two even. However, there is a limit to the extent to which a program can be split up, with most programs making use of their own results as they proceed, so things must progress sequentially. But for many tasks (especially in the field of data analysis) there are large tasks that can be split up and processed simultaneously. Hence the birth of parallel computing.

In the past, the only way to achieve a decent level of performance from parallel computing, was through the use of multi-processor machines. This led to super computers using Massively Parallel Processing (MPP) techniques, to achieve high performance and throughput. These machines were, and still are, amazingly complex and expensive. They are very useful computing resources, but are not very economical. The price of processing power on these machines is very high, as the cost of single machines grows expo-

nentially with the power of the machine. This being the case, these machines are made to work hard, and can not often be used for long periods of time due to the high cost. However in the late 1990's the speed of the desktop computer improved significantly, and with the growth of the internet many of these machines (especially in businesses) spend a lot of their time very well connected but standing idle. In businesses, most machines remain idle for most of the time, certainly at night when everyone goes home. So some people suggested that it might be a good idea to harness this huge, relatively cheap resource of processing power for use with parallel computing, using the available idle time of machines to process data for people. Indeed desktop machines are now so cheap and powerful that it is often cheaper to operate a large network of desktop machines than a single MPP machine of a similar processing capability. This is Grid Computing [1], using large networks of workstations (NOW's) to perform parallel processing tasks.

A few problems are introduced in this model of parallel computing compared to the traditional "one machine" approach. With lots of independent machines operating together as one, the chance of failure of an individual node increases dramatically, particularly if the nodes are very physically dispersed and connected using the internet (a notably unstable medium). As the number of nodes increases, the odds of one failing massively increases, also, in these days with the possibility of many thousands of machines working together - the odds of a long running process not failing on at least one node is almost zero. When only using idle cycles of machines and giving the machine back to users when they want it, this becomes a definite problem. As a machine stopping execution for whatever reason, when unscheduled, would have the same effect as a node failure. In the past on single machines, the only type of node failure you would get was a hardware failure - something that would probably cause the entire machine to fail at any rate, so this area was largely unaddressed. It is now essential to be able to cope with the failure of one or more nodes at any point.

The problem this project seeks to address is how do we cope with these node failures on large parallel systems?

1.2 Aims and Objectives

The aim of this project is to look in to the area of fault tolerance in parallel programming systems and develop a fault tolerant implementation of one of the major message passing libraries used in parallel programming.

Many attempts to solve the problem of fault tolerance in parallel computing have been made before, but none have addressed all of the issues completely. The fact that there is still no standard or even a commonly used fault tolerance system in use for general parallel programming (other than in special Grid Computing systems like Condor[2]), goes some way to stating the need for further work in the field. The essential issues that people have tried to address in the past are:

- transparency
- speed of recovery
- heterogeneity
- multi-user support
- message logging
- environment independent checkpointing
- load balancing
- dynamic resizing
- memory efficiency
- network load optimization
- and many more...

There is a good reason why nobody has tried to address all these problems in one go. Many targets work against others, for instance the goal of transparency often complicates the optimization targets. If you spend a long time trying to optimize for load on networks/processors/memory/storage, processes will inevitably take longer making performance suffer. This project is a short-term undergraduate project so can not hope to tackle all of the issues, when large teams of researchers take many months to cover single issues. This means that specific areas of interest had to be chosen to be addressed and improved.

The issues involved in the decisions laid out here are discussed in chapter 2, so please refer there for justification.

Most existing solutions to this problem use checkpointing techniques to provide a ‘rollback-and-recover’ mechanism in the event of failure. This seems

to be the best approach to fault tolerance and is indeed the main method used in conventional serial programming fault tolerance. The other approaches are either extremely complicated or don't quite handle some essential problems. Checkpointing is covered in detail in section §2.4. The checkpointing methods so far employed are predominantly transparent checkpointing systems. From a programmers point of view transparency is often convenient, as it saves code having to be rewritten. However, it is not always the best approach. Transparency requires generalisation of problems down to a basic common level that is always correct. In most cases this means that transparent checkpointing requires complete copying of all state information for a process including the current position of any active I/O streams and the entire memory area of the process. This type of checkpointing can occur at any interval, meaning that checkpoints could be taken at the most inappropriate times.

Another solution is to let programmers participate in the act of checkpointing by suggesting (or forcing) places to perform checkpoints rather than letting a system decide itself. There are existing systems that rely on the applications themselves to explicitly request checkpoints and they work fairly well. The question of what to checkpoint can also be addressed, should everything be checkpointed blindly or should an educated state save be made?

This project will attempt to address the issues of user-directed checkpointing, where the user specifies the data necessary to be saved and the locations where checkpoints should be taken within the code. The existing implementations of the two major message passing libraries are highly complicated and extensive, modifying the existing code to add the fault tolerance features would be a major effort. Developing a new interface from scratch would be a project in itself and is undesirable as the existing interfaces are well known and sufficient. So to demonstrate the techniques a basic implementation of MPI has been produced, that supports all the basic features and additionally has fault tolerance with user-directed checkpointing and system-wide recovery built-in. All types of node failure are dealt with from single-node to multi-node failures. Basic non-heterogeneous process migration is also addressed in this system. Areas not being addressed are heterogeneous process migration, I/O streams and other external checkpointing and rollback, and general efficiency issues.

The software has been developed on Linux systems only, but could possibly be ported to another platform with a little work (portability is addressed in section §5.2). The code is written entirely in ANSI C, and therefore can be compiled and used in a standard way with most existing parallel applications written mainly in C or Fortran. A Fortran wrapper to the library would be

reasonably trivial to implement at a later date.

1.3 Assumed Knowledge

It is assumed that the reader has a strong understanding of the following...

- general programming (particularly the C language), if the reader needs a better overview of the C programming language they are referred to Kernighan & Ritchie’s book “The C Programming Language” [26].
- some understanding of general parallel programming using the MPI Library, (a brief overview of the MPI standard will be covered, but not in great depth). The reader is referred to the MPI Standard [3] for more details of programming with MPI.
- knowledge of the TCP/IP communication system is required, especially the basics of client-server and peer-to-peer applications. The reader is referred to Richard Stevens book, “TCP/IP Illustrated” [27] if they need further information on this area.
- as FTMPI makes programs threaded, a knowledge of the way threading can effect programs would be useful. The interested reader should read Magee & Kramer’s book “Concurrency” [29].
- a familiarity with the Linux operating system and the way it handles processes would be useful to.

1.4 Report Outline

This report is laid out in to the following sections.

Chapter 1: Introduction to the project, with a general overview of what the project aims to achieve.

Chapter 2: Background issues involved with the project, such as an overview of both parallel programming and checkpointing. Together with a detailed analysis in to what has already been done to solve this problem.

Chapter 3: This chapter describes the design and implementation of this projects solution to the problem, detailing the technical issues involved and the way that the software works. The chapter also contains the results of testing the system.

Chapter 4: An analysis of the performance of the software with comparisons to existing systems.

Chapter 5: An appraisal of the project, with conclusions as to what can be drawn from the work, how it compares to other existing products and how it could be improved, fixed and extended at a future date.

1.5 Project Website & Code Usage

FTMPI has a project website at the following locations:

- <http://www.doc.ic.ac.uk/~kao98/ftmpi/> (short term)
- <http://www.orbling.org/ftmpi/> (longer term)

The website contains full copies of this report in postscript (.ps), Adobe Acrobat PDF (.pdf) format, DVI format (note you will need the graphics files separately), and the original LaTeX document file.

The code for FTMPI is also present on the site, the code structure is the same as is documented in the implementation system (§3.3.1), but a brief overview will be presented here for users.

There are three directories making up the source code distribution:

- **bin/**

Directory containing the harness script, ‘`ftmpirun`’, and the test programs. A script is also contained in the directory, ‘`allcomp`’, which compiles the test programs.

- **include/**

This directory contains all of the header files for the system, if you wish to use FTMPI in your program, you should include the file “`ftmpi.h`” from this directory.

- **lib/**

This directory contains all of the source for the FTMPI library, there is a make file in this directory that can be used to compile the library. Simply type the following to compile the library:

```
home:/ftmpi/lib > make clean
home:/ftmpi/lib > make
```

The library should compile on any standard linux distribution, note that the library uses the ‘**pthread**s’ library.

If you wish to compile a program with the FTMPI library, then as well as including the main FTMPI header file, you should also link your program to the FTMPI library and the **pthread**s library. A sample command line would be:

```
home:/src > gcc -I/ftmpi/include -L/ftmpi/lib -o myprog myprog.c
               -lftmpi -lpthread
```

The extensions to the standard MPI interface for controlling the check-pointing are documented in section §3.2.5.

This site should eventually also contain a selection of papers and software on the issue of fault tolerance in parallel programming.

Anyone is welcome to use this code, and comment on it. Comments should be sent to **karl.orbell@ic.ac.uk** or **karlorbell@hotmail.com**.

Chapter 2

Background Issues & Related Work

2.1 Parallel Programming

As mentioned in the description of the problem (§1.1), parallel programming is a method of accelerating the execution of code via duplication of resources, and executing code simultaneously (in parallel). Today large parallel computing systems are in use all over the world, from the massive super-computer MPP style systems, to a simple dual processor SMP (Shared Memory Processor) system on an average user's desktop. At a basic level, large multi-processor systems, in whatever configuration and form, are often used simply to run multiple, unrelated, programs simultaneously. This level of parallelism is unsophisticated, as the programs do not alter the way they operate or execute any faster, there are just multiple programs running alongside each other. This type of system is common place on single processor machines as well, as it can be emulated by an operating system technique called *multi-tasking*. Multi-tasking works simply by sharing a processor among many processes (tasks), switching between them at timed (or otherwise) intervals. In this way some apparently parallel programs can be written using threads (light-weight processes, one program - but multiple independent processes within it, sharing the same memory).

In order to really exploit multi-processor machines and get programs to execute faster by expanding the processing power, the programs themselves must be altered to work in a very different way. In order to parallelize a program's execution, the code must be analysed for parallel execution opportunities. These are places in the code that could be executed simultaneously.

For instance, many scientific applications have to perform the same calculations on large quantities of data, if the calculations don't depend on each other then they could all be done side-by-side with enough processors. These programming techniques require a lot of communication between processors to coordinate the work they do, as usually there are sections where things can be done separately, but also times where the program must act in a serial manner, and have access to the calculations from all of the processors. The communications take the form of data passing, program control and synchronization, and are usually done via some means of message passing.

Over the years many parallel systems have been used, originally most had their own custom message passing interfaces and libraries. However this meant that most code had to be programmed freshly for any new parallel computer it was to run on. The industry soon realised that a standard parallel programming methodology and interface were required for all parallel machines. If the programming interface is standardized, and the interface is implemented for all parallel machines, then the actual programs do not need to be rewritten in order to be used on many different platforms. Over time two interfaces have become predominant within the industry with almost all parallel machines implementing them - MPI and PVM.

2.2 MPI and PVM

In the late 80's and early 90's, PVM: Parallel Virtual Machine [13] was born. This project aimed to provide a common interface for programmers to produce parallel programs with, whilst at the same providing another very useful function. PVM is a system that allows a network of heterogeneous nodes to be viewed as a single "virtual" machine. It effectively makes any number of nodes look like a single MPP. Its basic operation is very simplistic, it is effectively a task manager with a message passing communication system between tasks. PVM's main objective is interoperability of platforms, with a strong focus on ease of use, and heterogeneity. Its use is very widespread and indeed many later platforms are based upon it [7, 10, 11, 9]. PVM's main strength lies in its dynamic nature, any task can add new nodes or tasks to the virtual machine and any task can be removed at any time. Therefore PVM is able to dynamically adapt to the size and scale of problems, this allows for a degree of built-in fault tolerance as the system is already dynamic with respect to nodes coming and going from a live system.

A short while afterwards in the early 90's, the Message Passing Interface (MPI) forum was established. This forum consisted of around 60 of the top

professionals from the world of high performance computing and parallel programming. Their objective was to produce a message passing interface layer that would become a standard over all machines, so that a general programming approach to all parallel programming could be developed. Their main concerns were performance, portability and generality. They also wished to produce a large library of standard primitives and functions to standardize much of the parallel programming world. They spent a few years producing a specification, and eventually in 1994 released the first specification of MPI1 [3]. MPI was very popular and because of its highly portable nature, most manufacturers produced MPI implementations for their hardware quite quickly. The performance of the interface, in many implementations, was very good indeed. However in the design, the main considerations of performance and portability restricted them towards a very tight specification in certain places, to allow all manufacturers to highly optimize their implementations. This resulted in MPI being fixed to a static process model. An MPI application had to start and finish as a single process, there were no dynamic capabilities. In the second incarnation of MPI, MPI-2 [4], the static nature of the system has been addressed, but not to the level of PVM. MPI-2 is capable of adding and removing nodes while an application is active, but it still doesn't have any process management abilities as they deliberately left this out again, due to the extensive portability requirements set by the standard.

There have been many comparisons between PVM and MPI(-2) as they are seen by many people to perform the same task [14, 15], but at heart the two systems have slightly different purposes. PVM's purpose is to provide a virtual machine, a unifying structure that presents a very constant and dynamic interface to every node that it serves. PVM's keyword is *interoperability* - by this they mean flexibility and heterogeneity. MPI is a message passing interface, pure and simple. Emphasis should be placed on the word interface - MPI is not an implementation as PVM is, it is a specification for a protocol that can be implemented in many ways. Their keyword is *portability*, a critically essential quality - but a restrictive one never the less. MPI is a communications standard for parallel programming, not an artificial environment as PVM is. Some people are finding that they need both the flexibility and fault tolerance of PVM, and the power and control of MPI, so in recent years hybrid systems have been evolving, using both techniques [7, 8].

MPI is very powerful and can handle any type of parallel programming, plus there is a wealth of knowledge and examples available for MPI. However, MPI is not very fault tolerant at all, and has no built-in mechanisms for

even detecting faults easily, let alone solving them. When others have tried to make fault tolerant versions of MPI, they have always ended up writing custom implementations of it, as most current implementations are either far too complex, far too large or just unable to have fault tolerance retrofitted to them. The most common and popular incarnation of MPI - MPICH [5] (the “CH” stands for Chameleon, representing the adaptability (portability) and speed of the implementation) is very good, but very large and would be difficult to modify. Indeed the CoCheck project [8] states that rather than adapt MPICH they implemented their own version of MPI - tuMPI (which takes its higher level API interface from MPICH, but its communications layer from NXLib). MPICH has recently developed a new backend daemon server that provides a degree of dynamic process management and fault detection to their version of MPI, it is called MPD (Multi-Purpose Daemon) [6]. This MPD module is essentially an MPI implementation with a similar feature set and purpose as PVM.

2.3 Parallel System Failure

A parallel application can fail for a vast number of reasons, so it is important to declare what type of failures we are interested in dealing with.

In this document the concept of a failure refers to a single node ceasing to be actively processing, for whatever reason, other than a software error. Bugs can only be repaired by recoding the application, and no amount of process migration or checkpointing will help with that. Indeed, this case has to be explicitly guarded against. There is no sense in continually trying to restart applications that have a bug present, as a loop will be created. The failures can be defined as a breakdown in communications between nodes. To this end, to detect failures, all that needs to be done, is for multiple machines to monitor each others conditions (this need not always be done with polling, general message exchange, or lack of, may be sufficient). If a process does not reply to communications, then the process should be marked as dying, if all nodes agree that a process is not available for a length of time (dependent on individual circumstances) then it should be considered dead, and action should be taken to recover the process.

Processes may die for reasons other than communication breakdowns, that aren't necessarily software faults - such as running out of memory. An advanced load balancing and scheduling system, such as the MIST scheduler [10], might be able to migrate this process to a machine with more memory,

and use the checkpoints to re-establish the program. However, in general this is outside of the basic node failure problem.

But how do we cope with these failures? If a node fails, you could just repeat the work elsewhere. This is a possibility, and indeed this is often the case in modern systems, however if the process had been running a long time this may result in a significant amount of lost work. If this process was part of a team of nodes, with something waiting for all their results in order to progress further, the delay to restart one of the processes and wait for it may be exceptional - doubling the length of time in the worse case. What was needed was a method of saving the state of programs at certain intervals, so that in the event of failure we could start from a more recent position and minimise the loss of work. To this end the concept of checkpointing arises.

2.4 Checkpointing

Checkpointing is a way of taking a snapshot of activities at any point in time, so that in the event of failure, the task can be resumed from that point. It is basically saving state, for later resumption. Checkpointing can be done in a number of ways, although it can be very difficult depending on how much information needs to be saved.

Most parallel programs work by nodes exchanging messages with each other, exchanging data, telling each other what stage they are at, etc. One way of checkpointing is to log all of the messages and interaction between tasks, and if a process fails, and needs to be resumed, all that needs to be done is replay the messages that it received from the log to get it to the point in the program where it was before. If messages are sent frequently, then this approach may be sufficient and not lose much work. However it assumes that a process' entire state can be described by what messages it has sent and received - ie. it knows nothing of the outside world other than the message interface. This means it cannot rely on non-message state information, such as the content of files, external conditions (time, sensor conditions, network traffic, etc.). It is often possible to use this approach, and indeed many projects have done so, but the amount of data transferred between applications can be very great, and so message logs can grow very large.

Another approach is to use process level checkpointing on the machine in question. This requires saving the state of the program in terms of its current workspace, heap and execution stacks on the machine. This type of

checkpointing works very well for single processes, and can be made to work for parallel applications given a little thought. If the checkpoints are taken on all machines in the collective, at a fixed stage in execution, then in the event of any failures, they can all be brought back to the last checkpoint before a problem, and restarted from there. Practically all of the existing single machine checkpointing systems use this approach to checkpointing. The most well known, original, checkpointer of this type for Linux is `Libckpt` [16], which implements full transparent checkpointing, as well as being the first to introduce user-directed and incremental checkpointing.

The debate between using transparent or user-directed checkpointing is rather extensive, and there are existing checkpointing systems that use both methodologies (indeed, a hybrid of the two is becoming common). Many people think that what is best is a seamless system that can be used to allow any existing program to have fault tolerance capabilities, just by linking to a new library or using a new system that has transparent checkpointing capabilities - in order that no code has to be modified and yet the program has new resilience. If this could be done well then this would typically be preferable, as modifying code requires a lot of effort. Transparent checkpointing, requires a great deal of work to achieve, and is not overly efficient, as the automated system has no means of differentiating between necessary and unnecessary state information, nor does it have any way of picking a good point in the program's execution to perform a checkpoint.

A lot of the more efficient (and successful) checkpointing systems are user-directed (non-transparent), here the programmer must help the checkpointing system in some way. The best way of helping the system is by specifying either when or where to take checkpoints. Taking them only at key points can save a lot of unnecessary work and many details can safely be ignored when checkpointing. Note that a drawback of non-transparent checkpointing is that it often needs the programmer to aid in recovery as well, as it is not always as straight forward to recover from specific points using specific information. If it becomes too complicated, the programmer effectively has to write their own checkpointing utility for every program, this is not very useful. Out of the existing user-directed checkpointing systems, there are two basic approaches to data checkpointing and restoration. The first (as adopted by `Libckpt`) is to perform a standard 'all-inclusive' checkpoint, but exclude certain memory areas from the checkpoint, when it is known that their content is not required. The second method is to not store the heap at all, and just store the data that the programmer requests, and explicitly states is necessary. Under either system it is the programmer's responsibility to ensure that all of the data needed for a recovery is

stored. Almost all user-directed checkpointers take checkpoints at explicit locations in the code, where the programmer knows the checkpoint will be most effective.

One general issue with any system, is whether or not the system requires any modifications, or privileged access, to the local operating systems or hardware of the machines in use. Interfacing directly with the OS can be very useful for checkpointing, as it allows the systems to optimize themselves significantly, by reading page tables and disregarding unnecessary information. Additionally this gives some advantages in process recovery after migration, as you can do things like retain process id's and other OS level only modifiable data. Also, if you need to operate multi-user systems with full security, and still sustain recoverability, then you may need full access in order to change the user allocation for new processes when they are restored (MPVM, which is a multi-user PVM implementation does this [11]).

A number of the single machine checkpointers attempt to do a lot of the traditionally kernel level tricks in user-space by looking at all sorts of user-space visible information. The `‘/proc’` file system on Linux makes a lot of this possible. *ESKY* [17, 18] and *CKPT* [19] lead the way here, with provision for extracting used memory pages from `‘/proc’` together with handling dynamic libraries and various other complications. *Libckpt*'s incremental checkpointing feature also performs data storage optimization by only storing memory pages that have been altered since the last checkpoint, it does this by detecting page faults on memory that is accessed. This type of process handling is obviously highly unportable and quite complex.

When trying to checkpoint a threaded application there are additional issues which must be addressed. A threaded application behaves a bit like a parallel program in its own right, so similar techniques to parallel application checkpointing must be used on threaded programs. Threaded programs are handled in Linux by `glibc`, so in order for threads to be restored to their original state (including thread id's running orders and so on), `glibc` has to be modified slightly to allow for thread synchronisation and recreation. A LinuxThread compatible checkpointing library has been produced, `libtckpt` [20], that does this - although it presently does not work with the latest Linux kernel (version 2.4).

Additional thought and care has to be taken towards the effect the programs have on external resources in between checkpointing. If they modify files in a way that would alter results on a repeat pass, then this needs to be taken care of, along side the checkpoints, a copy of applicable files would have to be taken (or some form of copy-on-write mechanism established).

Note that this is only needed for records that are first read, then written to, after a checkpoint - if a record is not written to after a checkpoint there is no need to store that information. Because of this, most real file checkpointing is done using an incremental system. Logs of all file changes since the last checkpoint are taken (the actual file is not modified during execution). After a successful checkpoint the changes from the logs are written out to the file, in order that consistency is always guaranteed. The problems with files are of course also present, in a reasonably identical fashion, with any I/O stream, including displays, user intervention and other peripheral activities - these however can be significantly harder to rollback. Many checkpointing systems implement a very basic form of I/O stream restoration, simply by keeping track of what streams are open when a checkpoint is taken and what their statuses are (current file pointer, flags, etc.) and making sure that they are in the same state on restoration.

A major issue of checkpointing a program is the degradation it has on performance, taking a checkpoint may require storing an awful lot of data - this can take some time. More over, on a parallel machine, you usually have to take a checkpoint at the same stage over the whole network, this often requires all processes to be synchronized before the checkpoint. This means making all of the processes reach the same point, possibly delaying a number of nodes, then taking the checkpoints and then resuming. A way around taking synchronous checkpoints is to take the checkpoints asynchronously, this can be tricky as you have to find a way to take an accurate fixed-position checkpoint, whilst the system is still active and in motion. The only way to do this, is really to have the system perform some sort of copy-on-write mechanism to preserve the state at the point of checkpoint, whilst allowing the system to continue. This approach is used in Fail-safe PVM [9], which discusses the issue of asynchronous checkpoints at length. One of the main ways in which it can be implemented on a UNIX based system is to perform a `fork()` at the time of checkpoint, which internally implements a copy-on-write system, preserving a copy of the current process state for use.

2.5 Process Migration

With a parallel system, the failure of one node is often unrecoverable, or at the very least it will take an unacceptable length of time for the original machine to resume. This leads to the requirement of process migration. A failed process (for whatever reason) may need to be resumed on an alternative node than the one it was previously working on. Leaving aside the issue

of heterogeneous systems for the moment, assuming all nodes are effectively identical, this means that the checkpointing data of each node needs to be available to the entire pool of processors. It is not necessarily possible therefore to store the checkpointing information on a local disk on each node as that may well be unavailable after a failure. Therefore each node (possibly including spare nodes that are not in use) must have access to all checkpoint data. This requires the checkpoint data to be stored either as a copy on each machine (requiring massive duplication and a heavy load on the network) or stored in a central location - such as a file server - where they can all reach. If you are replicating the stored information on to each node then the machine immediately becomes un-scalable as the length of time and disk space a checkpoint takes are proportional to the number of nodes. Even storing on a global filing system, this may be true, as each node must utilize the network placing a load on the interconnections, as well as using the file servers' physical disks which may have to operate serially. However this is unfortunately unavoidable, the checkpoint data is required anywhere so it must be sent everywhere. A problem with having a central storage space is that this then becomes a critical link, which may itself need duplicating. A way of partially solving the scaling problem with this technique, whilst retaining a reasonable level of migrational support, is to only replicate checkpoint files to a fixed number of machines. This reduces the level of migration to only machines that have a copy of the checkpoint, but still provides some resilience.

Process migration is in itself very useful, especially in systems where you are making use of idle processing power. If a machine suddenly becomes non-idle you have to relinquish control over it, this could effectively count as a node failure - after all it has been lost. The ability to migrate a process is therefore exceptionally useful in this situation, as the process in hand can simply be moved elsewhere. This principle can be extended to the task of load balancing. Certain nodes in the network may out-perform others, for whatever reasons, and it may be possible to move processes from poorly performing nodes on to high performance nodes to improve the execution speed. Alternatively the load balancing can be used to implement priorities, for instance, if a process is of a low priority, but is going to take some time, it could be moved from a high power processor on to a lower powered one, freeing the better equipment for more urgent applications. This is particularly important as many applications can dynamically alter in terms of their complexity as they explore different scenarios, so their requirements may change as they execute.

2.6 Heterogeneity

An important issue in the modern NOW (Network of Workstations) based parallel computation grids, is the issue of heterogeneity. In a large network of machines, it would be a huge restriction if all of the machines had to be effectively identical environments. After all, most computers effectively do the same things they compute, make calculations, and process data. The underlying hardware, or even operating system, does not in most cases alter the dynamics of a program. There is no semantic reason why the processes cannot run in different environments, as long as they communicate with each other in an agreed format. There are general issues with programming across different environments, the format of data structures and messages need to be clearly defined. Luckily this issue has been well addressed, and there are many standards that solve the issue of compatibility of programs.

One of the most useful standards as far as message passing between parallel applications is concerned, is the proliferation of the TCP/IP protocol as the networking standard all over the world. This protocol is standard on almost all computing environments today, resolving the issue of fixed format communications between the machines. Providing programmers take care to ensure the data they send across this network is similarly standardized, there should be no issues there. However in the area of parallel fault tolerance there are some big problems with checkpointing. With message logging checkpointing, if the messages have been sorted out so that they are uniform then there should be no additional problems. However in the case of process state checkpointing, which is sometimes necessary, the problem is large and obvious. The internal state of programs is normally stored as details about the state of the process at a point in time, including stack, and memory contents, details regarding open communications channels, etc. This information is inherently fixed to the specific architecture that the application is executing on. It is not possible to take this checkpoint and use it on a different environment, as it would be meaningless there.

There are three main approaches to dealing with this problem. The first is to get the application to store only what it needs to recover its state itself, storing the data in some common format that all environments will understand. This requires the programmer to be actively involved in the checkpointing procedure, effectively rewriting the checkpointing library for each application that requires it. This approach may even require different code on each architecture, so for a general approach it is not good. The second approach is to use universal checkpointing formats, which basically store the whole process checkpointing data in a way that can be converted to

work with any binary, in all environments. This can be very difficult to do, and time consuming, not to mention the fact that it usually requires compiler support on a per-environment basis. The last method is one proposed by Ssu and Fuchs with their project *PREACHES* [12].

PREACHES method of approach is to generate checkpoints for each environment in a network, for each process during execution. This means that at any stage environment-dependent checkpoints are available for each process for all available environments. This allows full process migration, using homogenous checkpointing methods, which are often highly optimized, making this method very quick. It is implemented by effectively duplicating all of the processes across the different nodes, so that in effect all processes run on all nodes. If this was really the case, everything would just run much slower and you would end up with the network being no better than a single node (and almost certainly significantly worse). However, their approach only replicates the skeleton of processes on machines other than the master. These slave processes don't actually do any calculations, they just have to maintain state-lock with their master. This is done by communications being used to keep the run-time environment of the slaves the same as the masters. This means that their data must be consistent, and their execution position in the programs must remain the same, therefore every time a master process enters or exits a new procedure the slave must follow. In practice the slave does not have to maintain position exactly, many movements are not necessary to follow and if you know where checkpoints will not occur then you can eliminate a lot of calls. Process migration is then very easy, all that needs to be done is for a slave process to become a master, a very quick procedure. This approach has been mentioned in-depth because it is a novel solution and one of the only effective ways of achieving heterogeneous process migration with any speed. A drawback of this method is that it can require a lot of memory on the executing machines.

Chapter 3

The Software - FTMPI

This chapter details the design and development of FTMPI, a fault tolerant, skeleton MPI implementation.

3.1 Requirements

Before the software can be developed, clear requirements must be laid out, detailing exactly what the software must do. From the research conducted in chapter 2, it was decided that this project would focus on the issue of producing a version of MPI, that had built-in user-directed checkpointing facilities and full fault detection and recovery (as specified in the “aims and objectives” section §1.2). There follows a list of all the requirements for the project, along with justifications.

- The MPI implementation must support at least the following basic set of functions. This is the minimum set of functions that are required to conduct almost all parallel programming problems. More advanced primitives can be constructed from these base primitives. For more detailed explanation of these primitives see the MPI Standard [3].

`MPI_Init` Initializes MPI system from arguments passed to the program, as well as setting up any underlying fault-tolerance devices that are added.

`MPI_Finalize` Shuts down the MPI systems, along with fault-tolerance devices.

MPI_Initialized Checks whether or not **MPI_Init** has been called successfully.

MPI_Finalized Checks whether or not **MPI_Finalize** has been called successfully.

MPI_Comm_size Returns the size of the given communicator set.

MPI_Comm_rank Returns the rank number of the node that calls it, within the given communicator.

MPI_Send Basic message sending primitive, takes a message and delivers it to another node.

MPI_Recv Basic message receive primitive, attempts to receive a message from another node. Blocks if no message is available.

MPI_Barrier Main synchronization function, blocks until all nodes within the given communicator (set of nodes) call the function.

MPI_Wtime Timing function, returns some time value as accurate as possible.

MPI_Wtick Returns the resolution of the time given by **MPI_Wtime**.

The following additional commands are also desired, although not absolutely required as they can be implemented using the basic primitives. **MPI_Probe** is very difficult to implement with **MPI_Recv**, its inclusion is therefore more important than the other two.

MPI_Probe Checks to see if a message with a given source, or tag is currently available. If not, blocks until one is.

MPI_Reduce Global *fold* operation, over the given communicator.

MPI_Bcast Distribute data to all nodes within a communicator.

- The system (FTMPI) is to be written entirely in ANSI Standard C. This is because almost all implementations of MPI are written in C or FORTRAN, with a trend of people moving towards C. C is an extremely well known and portable language with compilers on almost every platform. It is also one of the few languages that has sufficient low level control of the system to extract operating system specific information, such as process state and memory information.
- The system is to be developed for use on the Linux operating system (kernel 2.4), there is no real need for portability to other systems for this demonstration project. But where portability can be included it should be.

-
- As Linux and the C programming language are being used, the FTMPI software should be implemented as a code object library that can be linked to any C program (and possibly other languages) in a standard fashion. A basic MPI program should compile and run correctly if using the FTMPI header file and linked to the library.
 - The software should not require a daemon process on all machines, although a start-up program that starts the MPI program running on a given machine set, and handles failures/restarts should be devised. It should operate as close as possible to the way the ‘`mpirun`’ command does in most MPI implementations.
 - All communications made by the software should be done via TCP/IP messages, with the programs acting as a peer-to-peer network. The system should try not to keep too many ports open simultaneously, certainly no more than a couple per machine in the collective. Parameters may however be passed to the client at startup through the command line.
 - The implementation is not required to have more than one node per physical machine (or IP address). This is because it would increase the complexity to deal with multiple nodes on one machine, as the nodes would have to share communication ports. Besides the fact that other than on multi-processor machines, multiple processes on one machine would not be likely to be useful.
 - In the event of a node failure (see section §2.3 for definition of a node failure) or multiple node failures. All surviving nodes must shut themselves down within a given time period, as cleanly as possible, ready to be restarted if checkpointing data is available.
 - The system must provide library function facilities to specify what data is required to be stored during a checkpoint, and an ability to order a checkpoint to be performed at any point. The system should take a checkpoint and store it to wherever the given working directory is. It is ideal if the working directory is a world-accessible file server, as this will allow the checkpointing data to be retrieved, even when the original node is no longer available. The library functions should be kept in the style of the MPI standard interface, so that the programmer feels they are part of the same API.
 - The checkpoints must be able to be used to restore the entire MPI environment, and restore the data the client application requested. Control

should ideally return back to the client at the point after the checkpoint was taken, and the client application should know whether or not a restore was performed.

- Process migration should be supported on homogenous machines, if the checkpointing data is available from previous machines. Therefore the MPI system must be able to start-up with a new machine set (although for simplicity the number of machines must remain constant). Client applications should be aware that the physical machines could alter after a restore.
- All nodes in the collection should be synchronized for a checkpoint, so that they can all take checkpoints simultaneously. It over complicates matters if you do not synchronize the nodes as you cannot guarantee that on a restart, every node is in the condition it was at the time of the global checkpoint.
- All machines must successfully complete a checkpoint in order for it to be considered a valid checkpoint.

3.2 Design

The majority of the design decisions are taken directly from the requirements section, but the specifics of the system need specifying here.

A diagram showing the external interfaces of the FTMPI library is shown in Figure 3.1. The interface is three-way, via a C library to local MPI programs, via a file server or local disk for checkpoint data, and via TCP/IP to other machines.

3.2.1 MPI System Limitations

The MPI interface implemented by FTMPI, includes all of the primitives specified in the requirements section (§3.1) - including the non-compulsory functions `MPI_Probe`, `MPI_Reduce` and `MPI_Bcast`. However, there are some limitations to the system from a complete MPI implementation.

Firstly, communicators, which are much used in MPI programming, are not entirely implemented. The underlying communicator system is present, but is presently restricted to only the global environment `MPI_COMM_WORLD`.

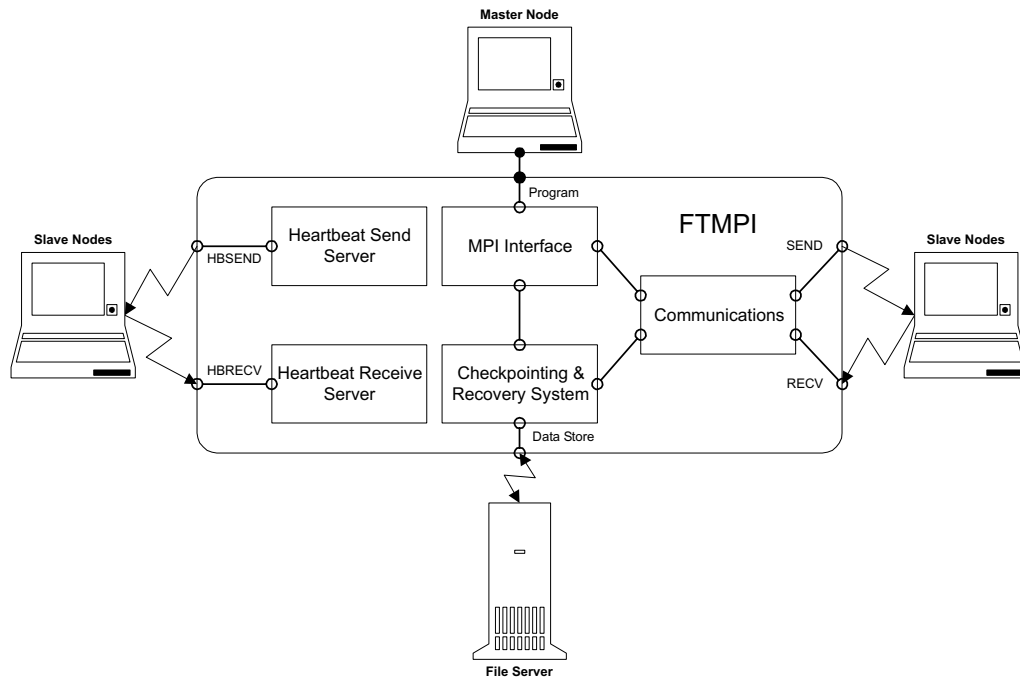


Figure 3.1: FTMPI Interfaces

This is especially true for the checkpointing functions which have to be performed over the entire global set of nodes, as a partial set would not be recoverable. The concept of communicators has been left in the interface to FTMPI, as it is needed for compatibility with the MPI standard and it would be useful to have the system extended to cope with a full communicator system at a later date.

The `MPI_Reduce` command also differs slightly from the standard definition. Because of the barrier-like nature of reduce, it is necessary to send signals back to the slave nodes (a reduce operation normally has a master target node for the calculated value, and slave nodes that just supply data) to let them release. As messages are going back to the slave nodes, the return message might as well contain the result. So in effect `MPI_Reduce` actually operates like `MPI_Allreduce`, returning the value to all nodes (if the destination buffer is defined on slave nodes). This is an added feature and does not affect the call semantics of `MPI_Reduce` at all (see Figure 3.2). Also, presently not all collective operator functions are implemented - it would not be difficult to add new functions, but at present only `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, and `MPI_PROD` are implemented.

For the sake of convenience (to the FTMPI implementation), not all the standard MPI data types are implemented by FTMPI. All of the basic types

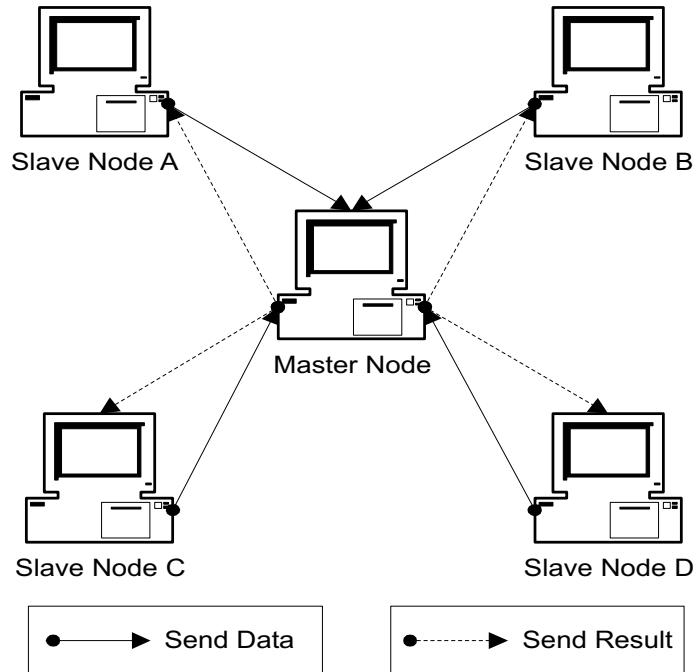


Figure 3.2: MPI Reduce dataflow

are implemented, but none of the combined data types like `MPI_LONG_INT` are included as they are not needed, because the collective operations `MPI_MAXLOC`, `MPI_MINLOC` are not implemented. The `MPI_PACKED` data type is also not supported, as the packing functions are not yet implemented.

Because of the proposed implementation of message sending (see below §3.2.3), `MPI_Send` will actually be non-blocking if the destination node is operational. This is within the specification as it does not explicitly state that `MPI_Send` should be blocking. No blocking version of `Send` currently exists in this implementation as it would require a significant redesign of the message passing system.

Apart from the differences specified above, FTMPI should operate in exactly the same manner as a normal MPI implementation.

3.2.2 System Startup

Although the programs that are linked with FTMPI are daemon-less, and can be run independently, it is desirable to have a facility that can automatically take a parallel program and a set of machines to execute it on, and get the program running.

Most MPI implementations use some form of startup program, that takes as parameters the number of nodes to run the program on, and a list of machines/processors to use for allocating the programs to. FTMPI is no different. MPICH [5] has a utility script called ‘`mpirun`’ that is called as follows.

```
mpirun -np <# nodes> -machinefile <machines> <program> <parameters>
```

Where ‘<# nodes>’ is the number of nodes required, ‘<machines>’ is the filename of a file containing a list of machines (1 per line), and ‘<program>’ and ‘<parameters>’ are the fully qualified program name to run, together with the standard parameters to be passed to that program.

This program is actually a shell script that processes the given machine file, and spawns the given MPI program (with additional command line parameters) on the specified number of machines. It starts-up the processes by choosing a set of machines, usually just the required number in the order they appear in the file, and using SSH (or another remote command interface) to start the program on an alternative machine. Note that in the basic mode the program specified must be at the same path location on each machine the program is expected to run on. ‘`mpirun`’ actually creates a file in the working directory containing the details of all the nodes in the machine and passes this filename on to each of the programs as its separate parameter.

FTMPI uses a very similar approach to MPICH, it too has a shell script that performs the same functions. The script is called ‘`ftmpirun`’ and is written in Perl [30, 31]. Perl is a very advanced scripting language that is particularly well suited to this task, and is usually a standard installation on all Linux machines, so it should not be a problem to use it. ‘`ftmpirun`’ takes the following parameters.

```
ftmpirun <machines> <# nodes> <w. dir> <program> <pars> [-restore]
```

Where ‘<# nodes>’ is the number of nodes required, ‘<machines>’ is the filename of a file containing a list of machines (1 per line), ‘<w. dir>’ is the working directory for the program (this is passed verbatim to the program, so must be valid on all clients), and ‘<program>’ and ‘<pars>’ are the fully qualified program name to run, together with the standard parameters to be passed to that program. The optional ‘`-restore`’ option specifies that `ftmpirun` should try to restore the application from checkpoints if they exist.

`ftmpirun` attempts to start the processes on the machines in question using SSH, all slave connections are spawned and the master node is always local. The programs are executed with their normal parameters and additionally they have FTMPI specific details appended to the parameter list. The `MPI_Init` function strips these extra parameters from those the program receives (for this reason `MPI_Init` should always be the first call in a parallel program).

Incoming arguments are scanned for a `-ftmpi` option, if it is found then the arguments after `-ftmpi` give the initial world state from the calling processes perspective.

```
-ftmpi <w. dir> <rank> <size> <rank/address pairs per node>
```

Where `<w. dir>` is the working directory for saving checkpoints, `<rank>` and `<size>` are the processes rank (identifier within the group, in the range $0..<size>-1$) and the number of nodes in the system. The `'rank/address'` pairs are a list of all of the nodes within the system, the rank first followed by the address (standard domain name) - there must be a complete set of rank/address pairs or the program will not function correctly. The `-ftmpi` and subsequent parameters are removed from the `'argv'` list before it is returned and the `'argc'` value is decremented accordingly. Additionally if the argument list is terminated by the following option:

```
-ftmpirecover <file>
```

Then a restore is attempted, using the checkpoint file specified by `<file>`.

The parameters are passed via the command line, rather than in a file as the command line is far cleaner and does not require all programs to have a common working directory. Also in the event of failure, having extra files lying around would not be helpful.

Once the FTMPI system has processed the startup parameters, it can start up all the necessary subsystems and return control to the calling program, either by a return from `MPI_Init` or as a return from a restore routine.

Before the subsystems are started it is necessary to note the time the system is starting, as many of the servers need to know how long the system has been running at any point. The `MPI_Wtime` function returns the time in seconds since the system was started, using the time that is set at this point as a base.

3.2.3 Message Passing System

The primary objective behind any MPI implementation, is to send and receive messages between nodes. Under MPI, a message is simply an item of data (in a known format) to be sent from one node to another in some fashion. Each message must be marked with the source node of the message, the destination node, (both nodes should be given as ranks within the global communicator), and a ‘tag’ field, as well as the data. The send and receive primitives, allow messages to be sent or received to/from a given node, and optionally a ‘tag’ value can be specified which, in the case of a receive call, only accepts messages that have been sent with that tag value. To prevent against messages being received out of the order they were sent, all messages should have a message identifier number (id), which is a sequence number, specific to each node. That way messages can be sorted on receipt, so that messages from any given node are returned in the original order they were intended. For the data section of each message, it is sufficient to know what type of data is being sent, together with how much of it. Data types are as defined by the MPI Standard. [3]

Messages are to be sent and received using TCP/IP connections. Each machine should setup a listening port, on which to receive messages from all other nodes. This port should be a standard known port on all machines. When FTMPI initializes it should setup the listening port (the server port), so that it is immediately available for other nodes to send messages to it. When a machine needs to send a message to another node, it should setup a connection to the remote machine’s server port and then proceed to send the message to it. As there can sometimes be a delay between processes starting up, repeat attempts to connect should be made for a small amount of time, with a gap between each attempt, if necessary.

As messages can arrive at anytime from other machines, it is necessary for the message receive handler to be active at all times. As the mechanism for receiving messages is part of the main program, continuous polling is not an option (at least not as part of the main process). The only options are: using some form of interrupts to jump to the receive handler when a connection is made or a message arrives; or creating a separate receive handler thread that can act as a dedicated message reception server. The interrupt system is not ideal as it may hamper the operation of the main process as it affects the way the process handles itself, also it is not particularly easy to implement. The dedicated receive thread is relatively simple to implement and disjoins the message reception from the main code, making it autonomous. The thread can simply be started up during initialization and forgotten about.

All incoming messages should be handled correctly.

As sends are not blocking, any process may acquire a large queue of messages waiting to be read by it, in the event that it does not read the values as they come in. This is quite normal, and needs to be dealt with. For this reason all incoming messages must be stored on a message queue. Any reads needing to be done by the MPI system can then be performed on this queue, allowing multiple messages to exist simultaneously. Because multiple threads will be modifying the state of the queue, all actions on the message queue must be mutually exclusive (using a combination of mutexes and condition variables). An additional bonus of using the thread-based synchronization facilities (mutexes and conditions), is that it becomes easy to implement a blocking receive, as the calling process can simply block until the queue becomes non-empty (which can be implemented using a condition variable).

The message queue has to differ from a standard queue structure (FIFO) in a number of ways. Firstly, when messages are inserted on to the queue, they must be kept in the message sequence order that they were originally sent in. This order is on a per node basis, and is simply a numerical sequence given by the messages' message id. Therefore instead of blindly adding messages to the tail of the queue, messages must be inserted into the queue at the correct position, which in almost all circumstances should be the end of the queue. The only condition is that messages from the same source are in the correct order within the queue. Secondly, messages are not necessarily removed straight from the front of the queue. Messages can be requested from the queue, specifying a combination of the source of the message and the tag of the message. There are also wildcard values which stand for any general purpose message (`MPI_ANY_TAG`), and any source (`MPI_ANY_SOURCE`). In order to implement special functions that require communication like `MPI_Reduce`, `MPI_Bcast`, and `MPI_Barrier`, tags above the wildcard value are used, these can only be requested explicitly from the queue. This means that all tag values from zero to `MPI_TAG_UB` (the upper bound for tag numbers) are general purpose tag values, this is in keeping with the MPI standard.

Because of these requirements, when a message is requested from the queue, the queue must be searched and the first matching entry returned. This also changes the problem of the queue blocking on empty, to the queue blocking if there is no matching message.

3.2.4 Heartbeat System

One of the main requirements of a fault tolerant system, is to quickly detect faults and act on them (taking faults as the definition given in section §2.3). There are two approaches to detecting network failure style faults, a lazy approach and a pro-active approach.

The lazy approach is fairly simple, only register network failures if, in the normal course of your programs execution, you try to communicate with another node and it does not respond in the correct manner. This would be fine if communications could be guaranteed to be occurring often, but if communications are far apart then a machine could have failed for some time before it is noticed. When a failure is detected the machine can only be restarted from the last checkpoint, so any extra delay in detecting a failed node is just extra lost work.

The pro-active approach is different, get all the nodes to talk to each other (or at least to another) on a regular basis - just to let the other nodes know you are still alive. This can be implemented in the form of a simple message sent periodically from one node to another, the message contents are generally irrelevant, it is the fact that the message has been received that is important. Because the messages are used to indicate that nodes are still alive, these messages are called heartbeats.

These heartbeats must be sent regularly to ensure that they are useful. Because they need to be sent on time, they can not afford to be held up by other activities within the application. So it is a good idea if separate send and receive threads for heartbeats are setup. One thread can simply send heartbeats out every interval to the nodes that require heartbeats, the other can deal with receiving heartbeats and detecting when a node has died (ie. a heartbeat hasn't been received for more than a specified time, which should be a little longer than the heartbeat period). In the event that a node dies the heartbeat receive thread should shut down the application. If all nodes send and receive heartbeats with each other then all live nodes should exit within a set period of time from any other node (or combination of nodes) failing. The shutdown should notify any surrounding environment (like `ftmpirun` on the master node) that the termination was abnormal and if there are checkpoints a restoration should be attempted.

3.2.5 Checkpointing & Restoration System

The main additional component of FTMPI over a standard MPI implementation is its ability to perform checkpoints, thus enabling restoration of incomplete programs that terminated due to node failure. FTMPI's checkpointing mechanism is entirely user-directed, the user must specify where to perform a checkpoint and what data is required to be saved. Because a parallel checkpoint requires synchronization of the system to be effective, it makes sense that the checkpoints are only taken at explicitly requested positions in the code. There are also other advantages to taking checkpoints at requested positions that are detailed in the checkpointing section of the background research chapter (§2.4).

There are two reasons why FTMPI requires the programmer to specify what data needs to be stored for each checkpoint, the first reason is a positive design decision, the second is an enforced design restriction.

The first reason is that given in the research (§2.4). Storing only the data that is explicitly requested (ie. actually needed) usually reduces the size of the checkpoint and provides a form of optimization. This is a good thing, but requires a significant amount of programming effort, if it cannot be predominantly automated.

The second reason occurs because FTMPI is threaded. Threaded programs are very hard to checkpoint, as they are effectively parallel programs in themselves. As stated in the research, proper thread checkpointing would require modifications to the underlying thread implementation, which on Linux would require modifying `glibc`, which is out of the scope of this project. At first it was thought that it may be possible to ignore the threads and just pause them for the duration of the checkpoint (not an easy operation), so that they don't modify anything, and then restore without the threads as the threads need to be reinitialized on a restore anyway. The idea here was that a normal checkpoint could be taken, storing the full contents (minus that which was deliberately excluded) of the heap and program data areas. If that was possible then much of the data selection process could be automated, as is done in most single machine checkpointers [16, 18, 19]. However, this would inadvertently restore the internal state of the threading library for the program with the rest of the programs data, if the threads were not recreated exactly as they were at the time of the checkpoint then this data would be corrupted and no thread operations would work. Because of this, it is not possible to store the heap or data areas, but just what the programmer tells us to save. The only way of automating this further would be via some sort of compiler assistance that looks for dead variables and saves the rest.

A possible alternative would be to carefully (re)store only the pages from the heap that did not contain threading library data. This might be possible if the ‘`mmap`’ tables for the process were carefully studied and the threading library section excluded. More details on `mmaps` can be found in the “Linux Device Driver” book by Rubini & Corbet. [21]

When a checkpoint is taken on a parallel application, the application must be synchronized so that the checkpoint can be guaranteed to be performed at the same time on all machines, so that all machines can be restored to the same point in time.

In order to synchronize the system for a checkpoint the following steps must be performed.

1. All nodes must call the checkpointing routine in the code (at roughly the same time)
2. An `MPI_Barrier` call or equivalent is performed to make sure that all of the nodes in the system are waiting to perform a checkpoint. As the barrier is a blocking operation, when the barrier call returns all nodes can be assumed to be externally synchronized and performing their checkpoint.
3. Because the individual programs are threaded, all of the threads must be paused, or at least guaranteed to not be altering any of the system. So the threads must at least come to a standstill. The only thread that will try to perform a checkpoint is the main application thread, so it is simply a matter of pausing the message reception thread, and the two heartbeat threads. The exact mechanics of a thread pause are best left to the implementation section (§3.3).
4. Once the threads are paused, the process is totally synchronized and a checkpoint can be taken, again the mechanics of the checkpoint mechanism should be left for the implementation section (§3.3). The only condition is that if a restore is performed, it should restart at this point, but without the following steps.
5. Once the checkpoint has been taken, the threads should be unpaused
6. Then another barrier call should be made, so that all nodes can confirm that they have performed a checkpoint successfully.
7. If all nodes have successfully performed a checkpoint, then any old checkpoint files should be removed (unless they are desired for some form of multi-stage rollback).

Finally, the interface for the checkpointing system needs to be specified. The interface is kept as near as possible to the same style as the original MPI interface, in fact all the code for FTMPI should be kept looking like the MPI interface, that way there is no clash of styles anywhere within the code.

The following functions are exported to the programmer from the checkpointing library. Each function is given with its prototype, and a description of how it is used.

int FTMPI_Perform_Checkpoint(**void**);

This function is called without parameters, it attempts to perform a checkpoint across all machines in the system, storing the current state of the data buffer for later retrieval. When it returns, the value it returns indicates whether the function is returning successfully (a checkpoint has been taken successfully - `FTMPI_CHECKPOINTED`), returning in error (checkpointing failed - `FTMPI_CKPTERROR`), or returning from a restore - `FTMPI_RESTORED`. In the event that the function returns from a restore the data buffer will already be in existence, with the data originally given to it for the last checkpoint, the application must then restore its data. The application should also note that because of process migration, the physical machine the program is running on, may have changed.

void FTMPI_CKPT_AllocBuffer(**long** dsize);

void FTMPI_CKPT_ReAllocBuffer(**long** dsize);

void FTMPI_CKPT_DeAllocBuffer(**void**);

These functions deal with the creation, alteration and destruction of the data buffer used to store checkpoint data. Both the allocation functions take as their argument, the size requested (in bytes) for the data buffer. On return the buffer will be that size. The reallocating function alters the size of an existing data buffer without destroying the data within it. The de-allocation function simply frees the data buffer once it is finished with - this is particularly important when a restore is being performed, once the data is restored, the buffer should be de-allocated.

void FTMPI_CKPT_ResetBufPtr(**void**);

long FTMPI_CKPT_SeekBufPtr(**long** offset, **int** how);

The data buffer is intended to function like a random access file, these two functions control the pointer to the current position in the buffer for the Add and Get functions. The reset function simply resets the pointer to the beginning of the buffer. The seek command moves to a position within the buffer given by an offset from a position specified by the 'how' parameter. This parameter can take the following values.

- FTMPI_SEEK_SET = offset is absolute position in buffer
- FTMPI_SEEK_CUR = offset is relative from current position
- FTMPI_SEEK_END = offset is subtracted from the end of buffer

On return the seek function returns the previous value of the buffer pointer. To find the current position of the buffer without altering the pointer you can call the function as follows.

```
FTMPI_CHKPT_SeekBufPtr(0, FTMPISEEK_CUR)
```

```
void FTMPI_CHKPT_AddToBuffer(void *var, long vsize);  
void FTMPI_CHKPT_GetFromBuffer(void *var, long vsize);
```

These functions either add or retrieve data from the buffer, the parameters are in both cases a pointer to a piece of data and the size in bytes of memory to transfer. Both functions increment the buffer pointer by the size of data transferred, and fail silently if the operation would try to read or write past the end of the buffer.

This interface provides all the necessary commands to implement check-pointing and restoration from a user's point of view.

3.3 Implementation

In this section, some of the intricacies of the FTMPI library code will be explained. The majority of the source code is self-explanatory however, and well commented, so a line-by-line explanation of the code should not be required. There is no real order to the items that need discussing, so they are explained here in a fairly random order.

3.3.1 General Structure

The FTMPI library code is organised into two sections, header files and source code modules. These are in the directories 'include' and 'lib' within the source code directory tree. At present the library is made up from the following source code files.

1. Header Files

- `include/collate.h`

This is the collective operations module header file, which defines the internal collective functions.
- `include/ftckpt.h`

This is the checkpoint system header file, it defines the internal checkpointing functions (the external ones are in `ftmpi.h`), and the checkpoint header structure. This is also the location of the hardwired checkpoint yield time.
- `include/ftmpi.h`

This is the main public header file, for inclusion in any programs that wish to use the FTMPI library. It defines all of the programmer visible constants and data types as well as containing the definitions for all external methods.
- `include/ftmpi_impl.h`

This is the main internal header file as it defines most of the internal constants and data types. In particular it covers the communicator functions, constants and data types that are used by most of the library.
- `include/message.h`

This file mainly contains the definition of a communication message for use in message queues and for transmitting across the network.
- `include/queue.h`

Contains declarations for all the basic queue functions, and defines the message queue type.
- `include/server.h`

This file defines most of the communication system functions and is the location for the hardwired port numbers and heartbeat intervals.

2. Source Code Modules

- `lib/barrier.c`

Contains the definitions for the `MPI_Barrier` call.
- `lib/collate.c`

This file contains the code for `MPI_Reduce` and the internal master collective function, `FTMPI_Collate`, along with all of the collective operation functions.

- `lib/commfuncs.c`

This is a very large code file as it contains all of the routines for handling the communicator objects, which contains all of the state information regarding the FTMPI system.
- `lib/dtype.c`

Defines the size (in bytes) of all of the MPI data types.
- `lib/finalize.c`

Contains the finalization routines, which shut down the FTMPI system.
- `lib/ftckpt.c`

This is the checkpoint subsystem module, it contains all code for taking checkpoints, including data buffer handling, stack and process state saving, thread pausing routines, and the restoration procedures.
- `lib/init.c`

This module contains the `MPI_Init` command, it processes the command line sets up the FTMPI system, initializing all the subsystems and starting a checkpoint restore if necessary.
- `lib/message.c`

This file contains the message comparison functions that are used for searching and inserting with message queues.
- `lib/queue.c`

This is the queue module, all the functions that can be accessed outside of this module are exclusive access only, so this module is multi-thread compatible.
- `lib/sendrecv.c`

This module contains the `MPI_Send`, `MPI_Recv`, `MPI_Probe` and `MPI_Bcast` functions.
- `lib/server.c`

This file contains most of the communication subsystem code. It contains routines for message sending and receiving, and the heartbeat servers.
- `lib/time.c`

This module provides the `MPI_Wtime`, and `MPI_Wtick` commands. Presently time is given using the `'gettimeofday()'` function, and as such the resolution is in microseconds.

3. Auxiliary Utilities

- `bin/ftmpirun`

This is the Perl script that is used to start-up parallel applications that use FTMPI on multiple machines.

C Program		
FTMPI Interface		
Communications	Heartbeat Servers	Checkpointing
TCP/IP		Disk I/O
pthreads		
Linux		

Figure 3.3: FTMPI System Structure

3.3.2 Communicator System

Even though communicators are not fully implemented in FTMPI, as only `MPI_COMM_WORLD` is available, the communicator object is very important to the internal operation of the FTMPI system.

The communicator object is the main storage area for all state information regarding the virtual machine that makes up the communication system. The communicator stores all of the local information about this node, such as the nodes rank, machine name, local IP address, current message sequence number, etc. But, that is not the main information it stores.

The communicator stores a linked list of node objects, each one of these objects contains all of the state information regarding the local nodes communications with the remote note. This is not just the nodes rank and machine name, it is also the file descriptors for all open sockets between the machines, the machines IP address, the latest heartbeat times for that node and other information.

All put together the communicator represents a repository of the FTMPI subsystem state information. It tracks open communications and deals with heartbeat checking.

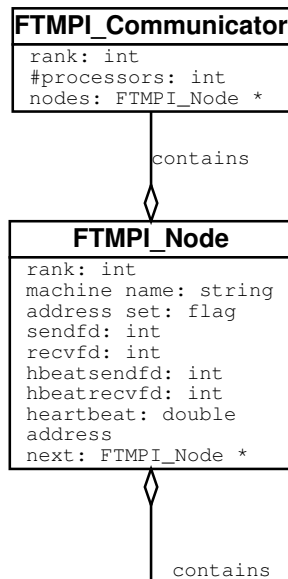


Figure 3.4: Communicator Structure

3.3.3 Message Passing System

The message passing system is the main component of the MPI system, and is essentially split into two areas: message transmission and message queue handling.

Message Transmission

Messages are transmitted and received in FTMPI by way of TCP/IP connections between the nodes. As sending and receiving are handled separately in the code, the explanations of the way they are handled will be separate here.

When a message needs to be sent, the `FTMPI_Server_SendMsg` function is called. This function attempts to send the given message straight away to the destination specified by the message (sends are non-blocking so the send must be performed immediately - there are no send message queues as yet). The procedure is simple; the communicator object is checked to see if an open socket for sending messages is available for the chosen destination, if it isn't then a new connection is created between the machines. If there is a problem creating a connection between the machines then the connection will be reattempted up to a preset maximum number of tries. It will give up after that many attempts. If a connection is established then the new file descriptor for the socket is recorded in the communicator object for later use and the connection kept open. Now that a connection is established, the message is sent over the connection. Note that the message is always sent in

two parts, the main message is sent with the size field set to the length of the data, and then the data is sent. This is to allow the receiving end to find out how big the data attached to a message is before receiving it, so that a buffer can be allocated for it.

Note that it is possible that a send operation on a connection will generate a SIGPIPE fault if the other node dies. Normally SIGPIPE signals kill the application so these signals must be trapped and ignored. If the signals are ignored then the TCP/IP send command returns a value to indicate a SIGPIPE fault, if this happens the connection should be considered closed and the socket descriptor removed from the communicator records. Whilst this probably indicates a node failure, it doesn't necessarily, as it is possible for a TCP/IP connection to be broken and re-established, so it is best to let the heartbeat system deal with detecting failed nodes. Also the SIGPIPE signal will only be returned if the machine on the other end is still alive and has the TCP/IP stack operating on it. No signal will be received if the machine is completely dead, the send will either return an error or block, so the heartbeat system really will be needed to terminate the application.

Message reception is handled by a separate thread, this thread is basically a `poll()` loop. `poll()` is a Linux kernel function that takes a set of file descriptors and monitors them for activity, returning when activity is seen on one or more of the file descriptors or after a specified timeout. The reception thread therefore maintains an array of file descriptors (in this case socket descriptors) starting off with only the descriptor for the listening server socket which is bound to port `FTMPI_SERV_PORT`, and adding all incoming message connection sockets as and when they are created. When `poll()` returns, all of the file descriptors are checked to see which ones have activity on them. If the listener socket has received a connection then the connection is accepted and the new socket added to the `poll` list (as well as being recorded in the communicator). If one of the message reception sockets has activity on it, this indicates an incoming message, the message header is read in and space is allocated for the data to accompany the message, then the data is read in. Once the message has been received it is inserted on to the incoming message queue. This action will wake any MPI receive functions that are blocked waiting for a message to arrive.

It is possible that during a receive, an EOF will be received on a message socket. This indicates that the other end has terminated the connection. Like with the sending problem of SIGPIPEs, the EOF will only be received if the other machine is still alive (not the application, just the TCP/IP stack) so can not be used as a signal of node failure on its own. In the event of the EOF, all that happens is that the function descriptor is removed from the

poll array and the communicator is updated to show that the connection has been closed.

Message Queues

To facilitate the special insertion and searching facilities that the message queue needs to provide, the queue is implemented as a doubly linked list. This is so that searching can be performed in both directions, from front to back, and vice versa. Forward searching is needed to match the earliest message with specific `TAG` and `SOURCE` values. Whilst reverse searching is needed to insert messages in the correct order (specified by message id - individual to each node), just in case messages go out of order.

The message queue implements two types of retrieval function, `Remove` and `Peek`. The only difference between the two functions is that `Peek` does not remove the message from the queue, it merely takes a pointer to it whilst leaving it on the queue. Blocking and non-blocking versions of both these functions exist, at the moment only the blocking versions are used as both `MPI_Recv` and `MPI_Probe` are blocking calls. Both `Peek` and `Remove` are actually implemented using the same internal function, `FTMPI_Queue_Contains`. This function checks to see if a message of the required type exists on the queue and returns the message if it does, the function additionally removes the message, if told to, when it returns it.

3.3.4 Heartbeats

The heartbeat system is essentially very simple, there are two threads, one for sending heartbeats and one for receiving them.

The heartbeat sending thread operates in a very similar manner to the message sending routine. At every preset time interval (specified (in seconds) by `FTMPI_HBEAT_INTERVAL`), a heartbeat message (which is simply the source of the message along with the local time) is sent to every node in the system. The communicator object stores whether or not a connection is already open for heartbeat messages with the destination server. If a connection is not open, then one is opened - storing the connection details back in the communicator. Once a connection is available, the message is simply sent across the connection in one go. Once all of the heartbeats have been sent, the thread tries to sleep until the next set of heartbeats needs to be sent. The sleep is implemented using a timeout on a never used thread condition variable, this allows the thread to sleep cleanly without disturbing other threads.

The heartbeat receiving thread is more complicated. Like the message receiving thread a `poll()` structure is used, starting off listening to port `FTMPI_HEARTBEAT_PORT`. New connections are added as necessary, and the communicator object is kept up-to-date on what connections are currently open. Additionally the heartbeat thread has to detect timeouts on heartbeats from the other nodes. This can happen in one of two ways, either no communications at all are received for the maximum time (possibly indicating total network failure), in which case total node failure should be assumed; or a single node has not sent a heartbeat for the maximum time. The way single (or multiple) node failures are detected, is by checking all of the current heartbeat times against the current time at the end of every `poll()` loop, this way any activity within the global timeout time, will cause a check on the timeout values for the other nodes.

In the event of any heartbeat timeout, the heartbeat receive thread should shutdown the system by calling the `FTMPI.Server.NetworkFailure` routine, which closes up the sockets and terminates the program, returning an error condition to any surrounding environment like `ftmpirun`.

3.3.5 Collective Operations

Currently there are three types of collective operation that are supported by FTMPI: the barrier, the broadcast and the reduce operation. The implementation of each of these will be detailed here. In most MPI implementations, these routines would not be implemented using the general purpose Send and Receive primitives, at least not directly. As the more advanced implementations tend to perform collective operations in highly optimized ways that minimize communications between nodes. FTMPI performs all these operations by building on the basic send and receive primitives, this is for the sake of simplicity.

MPI_Barrier

The barrier method is quite simple, when it is called, a dummy message (containing the nodes rank), with the `TAG` set to the special barrier tag value, is sent to all the other nodes in the communicator. Then the routine blocks until it has received similar barrier messages from all the other nodes. As the send primitive is non-blocking, all of the messages will be sent off at the start of the routine so that other machines can immediately read those messages when they call the barrier function. As the last node to call the barrier sends

out the barrier messages, the other nodes will start to return from the barrier call.

This method could be implemented without having to have all nodes talk to all other nodes, that would be better as it would be more scalable, but it complicates the algorithm considerably and the aim of this project was to have a basic working MPI implementation, not a highly optimized one.

MPI_Bcast

The broadcast operation is really very simple to implement, as it is not supposed to be a blocking call on the master node. All that happens when the broadcast is called, is that the master node sends the data in the given buffer to all other nodes in the communicator, the slave nodes just perform a receive for the transmitted value from the master. Indeed on the slave nodes, the broadcast routine could seamlessly be replaced with an `MPI_Recv` call.

MPI_Reduce & General Collective Operations

In FTMPI the `MPI_Reduce` function, calls an internal function, `FTMPI_Collate`, which can be used for a large number of the collective operations which perform some folding function.

The collate operation basically performs a folding operation on all of the data supplied by each node in the communicator, using the requested function (which must be one of the predefined MPI operator functions). On the master node (the target node for the final result and also the main calculating node) the data from all the other nodes is collected and folded into the final value, node by node, starting with the local node. Once all nodes have been processed and the final value is known, the final value is sent back to all the other nodes - this has the effect of making the collective operations act like a barrier as they are supposed to. It also has the nice added bonus of providing all nodes with the final result.

Each of the collective functions has to handle all of the different data types it can be applied to, using an operation with a data type that can not be handled by it is an error.

3.3.6 Checkpointing System

Checkpointing a program is never a simple operation, and having a threaded program that also has communications with other machines complicates mat-

ters further. As explained in the design section (§3.2.5), a checkpoint in FTMPI is composed of a barrier across the nodes, pausing the threads, taking a local checkpoint, un-pausing the threads, and a another barrier to finish. Here a detailed description of how these steps are taken will be given.

Thread Pausing - Internal Synchronization

In order to take a local checkpoint, it is necessary to have the program completely inactive, other than the actual checkpointing function. In a threaded program, the threads share each others memory space, so the actions of one thread can affect the environment for all the others - therefore in order to take a checkpoint, all threads other than the one taking the checkpoint (which incidentally has to be the master thread) must be inactive.

Fortunately, it is not difficult to make a thread inactive, as there are a number of functions that pause threads for various reasons. The most common function of this type is the condition variable wait function (in the `pthread` library this is `pthread_cond_wait()`). This pauses the calling thread until another thread notifies it to wake up, (by generating a signal on the condition variable). So in order to pause all threads other than the master thread, all that is needed is for the slave threads to all call the wait function on a condition variable that only the master thread will cause a signal for. The threads will all sleep until the master thread signals the condition variable, telling the threads to wake up again.

The main issues here are, how do the slave threads know when to call the wait function, and how does the master thread know when all of the slave threads are waiting? In order to deal with this, a combination of 2 variables, 2 condition variables and a mutex is used.

- int **FTMPI_numberPaused** - count of threads currently paused
- int **FTMPI_Paused** - pause request flag, other threads must yield if set to `FTMPI_PAUSED`
- mutex **pauseMutex** - main mutex used for exclusive access to the variables: `FTMPI_numberPaused` and `FTMPI_Paused`
- condition **pausedCount** - condition variable, used to wait for all threads to pause
- condition **pauseWanted** - used to hold/unhold paused threads

The mutex variable (`pauseMutex`) is used to ensure all access to the two variables used for thread synchronization is exclusive.

In order to request a pause, the master thread sets the `FTMPI_Paused` variable to `FTMPI_PAUSED`, and then waits for the slave threads to become paused. It knows when all threads are paused by keeping track of the number of threads that have paused already. This is done by keeping a count using the `FTMPI_numberPaused` variable, and waiting on the `pausedCount` condition variable which is signalled each time a thread pauses. Once all slave threads have paused, the master thread can proceed.

Periodically, the slave threads check to see if a pause has been requested by checking the state of the variable `FTMPI_Paused` to see if has been set by the master thread to `FTMPI_PAUSED`, indicating that a pause is desired. If a pause is desired, then the slave thread increments the count of threads that are currently paused (`FTMPI_numberPaused`) and then signals the master thread using the `pausedCount` condition variable which the master thread is waiting on, it then waits on the `pauseWanted` condition variable, until the master thread wishes to wake it up.

Once a checkpoint has been made, the threads need to be unpaused. To do this the master thread sets the `FTMPI_Paused` variable to `FTMPI_UNPAUSED` and then signals all of the slave threads (via a broadcast) on the `pauseWanted` condition variable to tell them to wake up. It then waits on the `pausedCount` condition variable for the number of paused threads (`FTMPI_numberPaused`) to drop to zero again. As slave threads unpause they decrement the number paused counter and signal the master thread.

There are three threads at present other than the master thread, these are the message reception server, and both the heartbeat servers. All these threads need to check to see if a pause request has been made very regularly, otherwise a checkpoint could take a very long time waiting for the threads to synchronize. The way this is implemented, is via the use of yield points within each thread. A yield point, is a place in the thread execution loop, where the thread volunteers to sleep if it is desired (it yields to the desire of another task). This terminology is borrowed from the world of cooperative multi-tasking, where this technique is employed heavily.

All three servers already spend most of their time sleeping, waiting for either incoming activity (in the case of the receive threads) or waiting to send the heartbeats. Unfortunately this complicates matters, as the threads have to be brought out of their normal sleeping patterns very regularly in order to attempt to yield to the master thread.

In the case of the message reception server, this is not very difficult at all,

as there are no timed waits in that code, the `poll()` call simply sleeps until there is activity. `poll()` provides a facility for a timed wakeup, so this facility is used to wake up after the preset yield interval, (given in milliseconds by `FTMPI_CHECKPOINT_YIELDTIME`), the yield call is at the top of the loop after any activity has been dealt with. Note that this may well yield before the elapsed interval, but that is fine, the interval is a maximum time between yields.

The heartbeat servers are more complicated, as they both already have timed waits within them. As the thread yield interval is significantly smaller usually than the other timed waits the threads are performing, the timed waits are integrated by forming the main waits using many yield interval waits. For both of these, a second countdown timer is implemented that uses the yield intervals as ticks, and performs the normal action associated with the original timed wait, when the countdown gets to zero. The heartbeat receive server is simply a poll loop, so this can be done in roughly the same way that the message reception server is handled. The heartbeat sending server is more confusing, as it uses a rather more complex timed wait system (using condition variables), so calculating what time the thread needs to wake up is a little confusing. A lot of the confusion in the code surrounding these timed waits comes from the fact that each of the various timing systems (of which there are three, `gettimeofday()`, pthread's `timedwait`, and `poll()`), use different time units. `gettimeofday()` uses seconds and microseconds ($s, \mu s$), pthreads uses seconds and nanoseconds (s, ns), and `poll()` uses milliseconds (ms)!

During testing the interval time for a yield (given in milliseconds by `FTMPI_CHECKPOINT_YIELDTIME`) was set to $500ms$ (half a second). This value seemed to be tolerable, as thread synchronisation can then take no longer than half a second, and the yielding doesn't impact too much on the performance. This may need to be made smaller though if checkpoints need to be taken more regularly.

Local Checkpoint Mechanism

Once the application is entirely synchronized, both externally across the MPI system, and internally between the threads, a local checkpoint should be taken. The routine that handles the local checkpoint in FTMPI is called `FTMPI_Checkpointing_WriteCheckpoint`. This routine takes as an argument, a filename to save the checkpoint to, and returns one of three condition codes, stating either a successful checkpoint, a failed checkpoint or that a restore has been performed.

The local checkpointing routine starts by attempting to create the new checkpoint file, if that works then the checkpoint can be taken. At this point a call is made to the `setjmp()` routine, this routine is fundamental to the checkpoint. When called, it stores a copy of the current register state (including program counter) to a buffer called the ‘jump buffer’. At a later point this jump buffer can be used to restore the state of the registers and start executing code from the point the `setjmp()` routine was called. The way to detect whether a `setjmp()` routine is returning from a normal call to it or from a restore, is via the return value which is zero if a restore has been made. When `setjmp()` returns to the checkpointing routine, the next step is for the header to be created and written to the start of the file. The header simply contains the nodes rank, the number of nodes in the machine, the duration the system has been running (ie. the current value of `MPI.Wtime`), the size of the data buffer, the size and current position of the stack, and the size of the heap - as well as the jump buffer of course. After the header is written, a copy of all the rank/address pairs for all the nodes is placed in the checkpoint - this is so that any machine changes can be detected. If a data buffer exists, then that is then written out to the checkpoint, and lastly the current stack is added to the end of the checkpoint file. The file is closed, and the checkpointing routine can return successfully.

The method of saving the stack needs some attention, as it is a delicate matter. The way FTMPI saves the stack is an old method used in `Libckpt` [16]. To save the stack, you must find out where it is and how big it is. Luckily in Linux the stack starts at a virtual fixed location, and grows downwards from there into the stack memory area. To find the top of the stack a little trick is used. Dummy local variables are placed at the beginning and end of the list of local variables used for the function that needs to save the stack. As local variables are stored on the stack when a function is called these dummy variables can be used to find the location of the top of the stack. Once the top and bottom of the stack are known, the stack can simply be read off and saved, the address of the top of the stack will also need to be restored on restart, so that must be saved too.

An alternative method to finding the stack is used by ‘`esky`’ and ‘`ckpt`’ [18, 19], which is rather more sophisticated. It reads the memory areas of the program (stack included), by reading the memory map information from ‘`/proc/self/maps`’, and locating the stack segment from this list. The stack can be found reasonably easily as the stack base is at a fixed location. This memory map technique is certainly a good idea, but requires a lot more work to implement, and still has associated risks. A sample memory map file is shown in Figure 3.5, the stack is the last entry - note the `libc` and `pthread`

library entries. Further information on memory maps can be found in the “Linux Device Driver” book, the memory map chapter is available free online. [21]

```

start      stop      perm offset          inode  mmap file
-----+-----+-----+-----+-----+-----
08048000-0804f000 r-xp 00000000 00:12 4007460
           /homes/kao98/lab/fourthyear/pft/ftmpi/bin/mmapdisp
0804f000-08050000 rw-p 00006000 00:12 4007460
           /homes/kao98/lab/fourthyear/pft/ftmpi/bin/mmapdisp
08050000-08052000 rwxp 00000000 00:00 0
40000000-40015000 r-xp 00000000 03:05 4474    /lib/ld-2.2.2.so
40015000-40017000 rw-p 00014000 03:05 4474    /lib/ld-2.2.2.so
4002f000-4003d000 r-xp 00000000 03:05 4495    /lib/libpthread.so
4003d000-40045000 rw-p 0000d000 03:05 4495    /lib/libpthread.so
40045000-40158000 r-xp 00000000 03:05 4479    /lib/libc.so.6
40158000-4015e000 rw-p 00112000 03:05 4479    /lib/libc.so.6
4015e000-40163000 rw-p 00000000 00:00 0
bf200000-bf201000 ---p 00000000 00:00 0
bf201000-bf400000 rwxp 00001000 00:00 0
bf400000-bf401000 ---p 00000000 00:00 0
bf401000-bf600000 rwxp 00001000 00:00 0
bf600000-bf601000 ---p 00000000 00:00 0
bf601000-bf800000 rwxp 00001000 00:00 0
bffffc000-c0000000 rwxp fffffd000 00:00 0

```

Figure 3.5: Sample ‘/proc/self/maps’

Restoration

When an application dies leaving a checkpoint available, it will need restoring at a later date. The restoration procedure is reasonably simple in FTMPI, as not much needs to be restored.

If a restore needs to be performed, the program is called with an additional argument, ‘-ftmpirecover’, and a filename of a checkpoint to use. This checkpoint file must be a checkpoint file for the same program (if the binary has been altered the restore will not work), with the same number of nodes in the system, and must be a checkpoint for a machine that had the same rank as it now holds. If all these conditions are met then a restore is attempted. A restoration follows the normal initialization procedure, up to the

point where the subsystem servers are brought back online. At this point `FTMPI_Checkpointing_Recover` is called to attempt to recover the program.

The recovery routine first reads in the header from the checkpoint file, and validates it to check to see if the checkpoint file is compatible with this machine setup (as described above). If the checkpoint file is valid, then the rank/address pairs are compared with the current system to detect any change in machines. If there are any changes, at the moment this is simply displayed to the user as debugging information, however, this could be used to make other changes that may need to be done. At this point the communication subsystems need to be started up, but before this is done the original server time (`MPI_Wtime` value) needs to be restored, so that any programs using the time functions are not corrupted. A problem with restoring the old time is that it affects the heartbeat system, as all of the heartbeats will be set to zero seconds, whilst the clock could read a vast amount of time more, causing all of the heartbeats to timeout as soon as the server starts up. Because of this the server time restoration, also sets all the latest heartbeat counters to the same time, which is equivalent to the heartbeat system starting up fresh.

Once the communication subsystems are operational, a barrier call across the nodes is used to check that they are all ready to restore from the checkpoint. Once all machines are ready to restore, the restore is performed.

The restore routine firstly recreates the data buffer as it was when the checkpointing routine was originally called, restoring the buffer from the checkpoint file. Then the heap size is set back to its original value (although this is possibly unnecessary, as the heap contents is not saved), and a stack and register state restore is attempted. As the active stack will be destroyed when the old stack is recreated, any variables that need to be passed to the stack restore routine must not be passed on the stack, so global variables are used. The information that is needed is the jump buffer, the file descriptor for the checkpoint file (the stack still needs to be read in), and the stack size and location.

The stack restore routine (`FTMPI_Checkpointing_RestoreStack`), must first get the current stack to be at least as big as the original stack, to do this it uses local dummy variables again to monitor the current stack size, and repeatedly calls itself (increasing the size of the stack with its temporary variables each time) until the stack is of a sufficient size. When the stack is big enough, the old stack is read in and then a `longjmp()` is performed with the old jump buffer, which should hopefully restore the system.

One danger with restoring a stack is that you can modify the stack you

are restoring with the calls that are used to restore it. This is dangerous, so methods have been proposed to deal with this. The ‘`ckpt`’ library by Victor Zandy [19], has a novel technique. When the stack needs to be restored, a temporary stack is created and via the use of `setjmp()`, and altering the jump buffer, the stack pointer is changed to the new stack and the stack restore routine is called (using the new stack), which restores the original stack, safe in the knowledge that the old stack is not being corrupted. If this approach was used the main stack would have to be grown, as done above, before the new stack was used for the main restoration. This feature may well be worth adding to FTMPI, at present the code works as is, so has not been altered.

3.4 Testing

As stated already, FTMPI is intended to be nothing more than a MPI implementation with fault tolerance. Therefore in order to prove program correctness it is only necessary to demonstrate that the MPI interface works correctly, that faults (as defined earlier (§2.3)) are detected within a certain time threshold, and that if a valid checkpoint has been taken, the system can be recovered.

3.4.1 Test Programs

In order to test the FTMPI system, a number of test programs were developed. There are five test programs in the existing test suite, two are simple tests of the MPI interface, the other three are increasingly complex parallel applications that might be used in the mathematical world. The three test applications are all long running and have checkpointing code built in, to test the fault recovery abilities of the system.

The MPI interface test programs are:

- **inittest**

This program simply tests the most basic MPI functions. It tests the `MPI_Init` interface to make sure that the command line interface works correctly. The program tests basic communications, by getting all slave nodes to send a message to the master node, containing some dummy data, the program then exits. It also checks the following routines:

- `MPI_Comm_rank`
- `MPI_Comm_size`
- `MPI_Initialize`
- `MPI_Finalize`
- `MPI_Finalize`

- **collective**

This program tests all of the collective functions in the MPI subsystem. To do this it starts off by using a call to `MPI_Barrier` to synchronize the nodes, then performs a continual loop, incrementing a number by $(\text{rank} + 1)$ and performing a `MPI_Reduce` for every possible collective operation, every so often. The results of the reductions are displayed for checking. `MPI_Wtime` and `MPI_Bcast` are also tested by taking the time on the master node and broadcasting it to all of the slaves, so that each output stage can be identified by a time on each machine.

The parallel application test programs are:

- **passnum**

This program is a dummy example of a peer-to-peer communicating program. To demonstrate the principle it passes a number around every node in the system, using a ring pattern. Each node passes the number to the node with a rank one more than its own. Every node increments the number by their rank plus one each time. This loop runs until the program is terminated. The program is also setup to perform a checkpoint every 1000 loops, the only data that needs to be stored for the checkpoint is the current number that the node in question is on, as all of the other information can be recovered from the MPI system.

- **pi**

This program is taken straight from a parallel programming exercise from a course on parallel programming held in this department this year. It is an implementation of the Monte-Carlo method of calculating the value of π (Pi).

The Monte-Carlo method of calculating π is essentially quite simple. A square is taken and a circle of the same radius as the square is placed inside the square (see Figure 3.6). Then random points are taken within

the square continuously and a count is kept of what percentage of points lay within the circle. Whether or not a point is in the circle can be decided by calculating whether or not the following predicate is true.

$$x^2 + y^2 < r^2 \text{ where } P(x, y) = \text{Point, and } r = \text{radius}$$

A more indepth explanation can be found at the website given in the bibliography. [22]

The program is parallel to increase the speed of the calculation, as this is a test program a reduction is performed at preset intervals across the system, to give a current estimate of π . At a greater interval a checkpoint is performed across the system, so the program can be recovered from that point on failure.

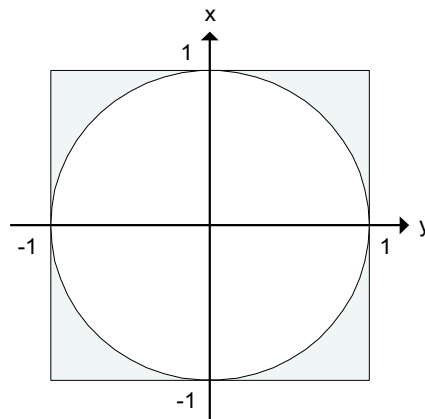


Figure 3.6: Monte-Carlo approximation of π

- **cgs**

CGS stands for Conjugate Gradient Solver, this is an iterative system solver, commonly used in many mathematical and scientific applications. A description of the algorithm is outside of the scope of this report, but this program is used to show that FTMPI can be used for more advanced applications. [23]

A significant difference between this program and the others is that there is a substantial difference between the execution of the master node and the slave nodes. The master node does the majority of the work and the slave nodes act as matrix multiplication servers. The slave nodes are given a matrix section at the start of the program and

for the rest of the programs execution they wait to receive vectors which they multiply and return the result.

To implement the checkpointing on this application, the slave nodes need to be sent a message to notify them to perform a checkpoint when the master node decides one should be performed. The checkpoint locations within the master program are taken at intervals, at the end of processing iterations. A vast quantity of information is needed to be saved as the matrix can be very large.

3.4.2 Tests

A number of tests must be performed to check if the system operates as planned. As mentioned earlier, the tests check the functionality of the MPI system, the fault detection (ie. the heartbeat system), and the ability to perform a checkpoint and recover. As it is a good idea to see the programs output executing independently, on each machine in the system, the automated start-up harness, `ftmpirun`, is not used for these tests. Instead, the programs are all started by hand, (given the correct inputs), that way the output is separate and easy to view.

MPI System

FTMPI's primary specification is to be a working basic MPI implementation, and therefore if the MPI implementation doesn't work, the rest of the system is effectively useless. It is therefore necessary to test the operation of all the MPI operations.

■ `inittest`

The basic MPI operations for setting up the MPI system, and sending and receiving basic messages, are tested using the '`inittest`' test program. To test this program, it was run over three machines with mixed parameters in the command line as well as the FTMPI parameters. The output from each is displayed below.

- Master Node - rank 0 - moa

```

kao98@moa:/ftmpi/bin >./inittest 1 2 -ftmpi . 0 3
 0 moa 1 texel23 2 texel24 3 4
Initial parameters [15] : ./inittest 1 2 -ftmpi . 0 3
 0 moa 1 texel23 2 texel24 3 4
Initialized? No
Initial parameters [5] : ./inittest 1 2 3 4
Initialized? Yes
Size: 3 Rank: 0
Received 5 from 1.
Received 10 from 2.
Finalized? No
Finalized? Yes

```

- 1st Slave Node - rank 1 - texel23

```

kao98@texel23:/ftmpi/bin >./inittest 1 2 -ftmpi . 1 3
 0 moa 1 texel23 2 texel24 3 4
Initial parameters [15] : ./inittest 1 2 -ftmpi . 1 3
 0 moa 1 texel23 2 texel24 3 4
Initialized? No
Initial parameters [5] : ./inittest 1 2 3 4
Initialized? Yes
Size: 3 Rank: 1
Sent 5 to master.
Finalized? No
Finalized? Yes

```

- 2nd Slave Node - rank 2 - texel24

```

kao98@texel24:/ftmpi/bin >./inittest 1 2 -ftmpi . 2 3
 0 moa 1 texel23 2 texel24 3 4
Initial parameters [15] : ./inittest 1 2 -ftmpi . 2 3
 0 moa 1 texel23 2 texel24 3 4
Initialized? No
Initial parameters [5] : ./inittest 1 2 3 4
Initialized? Yes
Size: 3 Rank: 2
Sent 10 to master.
Finalized? No
Finalized? Yes

```

For each test the initialization routine correctly removed all of the FTMPI parameters, taking the total number of parameters from the original fifteen, down to the ‘real’ five parameters. The initialization flag and the finalization flag checks work. The *size* and *rank* are correct for each machine. Both the slave nodes send ($\text{rank} * 5$) to the master node, which the master node receives correctly. This test also nicely illustrates the format of the FTMPI parameters. The working directory is set to the current directory, ‘.’, and the values following are the rank & size, followed by the rank/address pairs.

■ collective

The ‘inittest’ program checks the basic MPI operations, but does not check the more advanced MPI system facilities, namely the collective operations: `MPI_Barrier`, `MPI_Bcast`, and `MPI_Reduce`. These functions are tested using the ‘collective’ program.

- Master Node - rank 0 - moa

```
kao98@moa:/pft/ftmpi/bin > ./collective -ftmpi . 0 3
 0 moa 1 texel23 2 texel24
Performing barrier...
[2.027714s]
 collective (50): max: 150 min: 50 sum: 300 prod: 750000
[2.032405s]
 collective (100): max: 300 min: 100 sum: 600 prod: 6000000
[2.036936s]
 collective (150): max: 450 min: 150 sum: 900 prod: 20250000
[2.041420s]
 collective (200): max: 600 min: 200 sum: 1200 prod: 48000000
[2.045924s]
 collective (250): max: 750 min: 250 sum: 1500 prod: 93750000
[2.050470s]
 collective (300): max: 900 min: 300 sum: 1800 prod: 162000000
[2.054997s]
 collective (350): max: 1050 min: 350 sum: 2100 prod: 257250000
[2.059566s]
 collective (400): max: 1200 min: 400 sum: 2400 prod: 384000000
[2.064157s]
 collective (450): max: 1350 min: 450 sum: 2700 prod: 546750000
[2.068760s]
 collective (500): max: 1500 min: 500 sum: 3000 prod: 750000000
```

- 1st Slave Node - rank 1 - texel23

```
kao98@texel23:/ftmpi/bin > ./collective -ftmpi . 1 3
  0 moa 1 texel23 2 texel24
Performing barrier...
[2.027714s] reducing (50): 100
[2.032405s] reducing (100): 200
[2.036936s] reducing (150): 300
[2.041420s] reducing (200): 400
[2.045924s] reducing (250): 500
[2.050470s] reducing (300): 600
[2.054997s] reducing (350): 700
[2.059566s] reducing (400): 800
[2.064157s] reducing (450): 900
[2.068760s] reducing (500): 1000
```

- 2nd Slave Node - rank 2 - texel24

```
kao98@texel24:/ftmpi/bin > ./collective -ftmpi . 2 3
  0 moa 1 texel23 2 texel24
Performing barrier...
[2.027714s] reducing (50): 150
[2.032405s] reducing (100): 300
[2.036936s] reducing (150): 450
[2.041420s] reducing (200): 600
[2.045924s] reducing (250): 750
[2.050470s] reducing (300): 900
[2.054997s] reducing (350): 1050
[2.059566s] reducing (400): 1200
[2.064157s] reducing (450): 1350
[2.068760s] reducing (500): 1500
```

As you can see, the slave nodes performed the reduce operations at the correct times, with the times from the master node clearly displayed. The master node displays the results of the collective operations correctly at each stage, so they must work. The barrier operation did work, as all systems paused at this point until all machines were active - there is no way to show this in the report unfortunately.

Fault Detection

Whilst it may appear that the issue of fault detection should go hand in hand with the issue of checkpointing, they are actually separate. As detecting the faults can be done (and is done) independently of the checkpointing mechanism. It therefore makes sense to test this facility separately.

In FTMPI the fault detection system is essentially the interaction of the heartbeat servers. To test the correct operation of these servers, all that is necessary, is to see what happens when nodes fail.

The easiest way to simulate the failure of a single machine, is to kill the application process on that node, in such a way that the program is not informed about the death. On Linux this is luckily very simple to do, sending a SIGKILL signal to the process is sufficient, this can be done from the console of a live program by pressing `Ctrl-C`, or by sending the signal to it using the `kill` program. The only difference between this type of failure, and a full machine failure, is that the machine's TCP/IP stack is still operational after the failure, so any connections with other machines are properly closed. As the heartbeat system operates on the principle of timeouts only, this extra condition does not affect the tests. Of course it is wise to do some 'real' tests where the machine is actually made to fail, by switching it off or otherwise. A network failure can also be simulated by simply removing the network cable on a machine; this should have the same effect as if the machine was switched off.

With regard to the heartbeat system, there is no difference between the master node and the slave nodes, so the death of any node is equivalent to the death of any other - this means that it doesn't matter which node is killed for a test. For the purposes of all testing, the interval between heartbeats is defined by `FTMPI_HBEAT_INTERVAL` which was set to five seconds. The timeout for not receiving a heartbeat is by default set to double this time period, ie. ten seconds. For most long running programs, ten seconds is reasonably insignificant in terms of lost work, so this value is fine. For shorter running programs, it may be necessary to reduce the heartbeat interval - the timeout must always be slightly higher than the heartbeat interval as it takes time to send the heartbeats.

In the event that more than one node fails, the remaining nodes should timeout within ten seconds of the first failure. Technically, if two or more nodes fail simultaneously then the node that actually causes the other machines to quit should be the one with the highest rank - this is because the algorithm checks for timeouts in rank order. In practice, however, it could

be either; especially if both were trying to send a heartbeat at the time of failure. As the outcome is random, and it is technically very difficult to simulate the simultaneous failure of multiple nodes, it will not be done. The only tests needed to be done are a single node failure (any will do), and a multi-node failure, one machine a couple of seconds after the other. The final stages of output for the surviving nodes will be listed below. As the `passnum` test application has no output for slave nodes (other than error messages), it is ideal for this test. For the purposes of testing, the heartbeat timeout debugging information was switched on, this shows the time on the node, compared with the time of the latest heartbeat from each node. This can be used to see a node steadily decaying. In each example output, only the final three heartbeat checks are shown to give an indication of the final times.

- Single Node Failure

For this test the master node is killed (the only one with non-error output), and the output from the slave nodes is shown.

– Master Node - rank 0 - moa

```
kao98@moa:/ftmpi/bin > ./passnum -ftmpi . 0 3
0 moa 1 texel23 2 texel24
...
checking heartbeats [0] 12.040750 <=> [1] 6.999110
checking heartbeats [0] 12.040750 <=> [2] 8.003570
checking heartbeats [0] 12.550858 <=> [1] 6.999110
checking heartbeats [0] 12.550858 <=> [2] 8.003570
<<dead>>
```

– 1st Slave Node - rank 1 - texel23

```
kao98@texel23:/ftmpi/bin > ./passnum -ftmpi . 1 3
0 moa 1 texel23 2 texel24
...
checking heartbeats [1] 20.016686 <=> [0] 11.003835
checking heartbeats [1] 20.016686 <=> [2] 19.000577
checking heartbeats [1] 20.526681 <=> [0] 11.003835
checking heartbeats [1] 20.526681 <=> [2] 19.000577
checking heartbeats [1] 21.036678 <=> [0] 11.003835
checking heartbeats [1] 21.036678 <=> [2] 19.000577
Node 0 has not sent a heartbeat recently, aborting
```

– 2nd Slave Node - rank 2 - texel24

```

kao98@texel24:/ftmpi/bin >./passnum -ftmpi . 2 3
 0 moa 1 texel23 2 texel24
...
checking heartbeats [2] 20.037414 <=> [0] 11.003786
checking heartbeats [2] 20.037414 <=> [1] 17.000100
checking heartbeats [2] 20.541886 <=> [0] 11.003786
checking heartbeats [2] 20.541886 <=> [1] 17.000100
checking heartbeats [2] 21.051925 <=> [0] 11.003786
checking heartbeats [2] 21.051925 <=> [1] 17.000100
Node 0 has not sent a heartbeat recently, aborting

```

The output from these nodes, shows that node 0 - the master node - was the cause of the failure. This was repeated a number of times, and the time taken from the time the master node was killed, to the slave nodes exiting, was measured. The longest measured duration was 9.5s, which is below the 10s maximum. Although the clocks are not synchronized, the processes were started reasonably close together in time, so the clocks are almost accurate. The clocks show that in this particular example, the master node died at 12.5s and the slave nodes died at roughly 21s (the clocks are only roughly accurate to half a second - this is because the heartbeats are checked at each yield interval).

- Multi Node Failure

To test the multi-node failures the secondary slave node will be killed, and the master node will be killed shortly afterwards (about 1-2 seconds). The minimum amount of time that a failure will take to detect is the heartbeat interval time itself. As the timeout value is double the heartbeat interval, so if a heartbeat was just due to be sent when the node failed, it will take the length of a heartbeat interval to be detected.

– Master Node - rank 0 - moa

```

kao98@moa:/ftmpi/bin >./passnum -ftmpi . 0 3
 0 moa 1 texel23 2 texel24
...
checking heartbeats [0] 5.002989 <=> [1] 2.563716
checking heartbeats [0] 5.002989 <=> [2] 1.307744
checking heartbeats [0] 5.508759 <=> [1] 2.563716
checking heartbeats [0] 5.508759 <=> [2] 1.307744
checking heartbeats [0] 6.018669 <=> [1] 2.563716

```

```

checking heartbeats [0] 6.018669 <=> [2] 1.307744
<<dead>>
- 1st Slave Node - rank 1 - texel23

kao98@texel23:/ftmpi/bin >./passnum -ftmpi . 1 3
0 moa 1 texel23 2 texel24
...
checking heartbeats [1] 13.574519 <=> [0] 10.003633
checking heartbeats [1] 13.574519 <=> [2] 4.313481
checking heartbeats [1] 14.084518 <=> [0] 10.003633
checking heartbeats [1] 14.084518 <=> [2] 4.313481
checking heartbeats [1] 14.594518 <=> [0] 10.003633
checking heartbeats [1] 14.594518 <=> [2] 4.313481
Node 2 has not sent a heartbeat recently, aborting
- 2nd Slave Node - rank 2 - texel24

kao98@texel24:/ftmpi/bin >./passnum -ftmpi . 2 3
0 moa 1 texel23 2 texel24
...
checking heartbeats [2] 4.067897 <=> [0] 1.365517
checking heartbeats [2] 4.067897 <=> [1] 3.564584
checking heartbeats [2] 4.577867 <=> [0] 1.365517
checking heartbeats [2] 4.577867 <=> [1] 3.564584
checking heartbeats [2] 5.517898 <=> [0] 1.365517
checking heartbeats [2] 5.517898 <=> [1] 3.564584
<<dead>>

```

As expected the secondary slave node was the one that registered the fault, as it was terminated a couple of seconds before the master node. It can be seen that the secondary slave node was terminated at 5.5s, whilst the master node was terminated at 6s. The primary slave died at 14.5s on its clock - which is out of sync with the other nodes as the time of the last heartbeat from the master node was 10s, not 6s.

Checkpointing & Recovery

The checkpointing and recovery functionality should obviously be tested in conjunction, as to test that a checkpoint was successfully taken, it should be tried, ie. a restore must be performed. For these tests the 'pi' program will be used, as it is a real life application, and performs checkpoints containing

static data and data that changes every time a checkpoint is taken. It is also sufficiently long running that a number of checkpoints will be taken over the duration of the program's execution.

For the purposes of this test, the secondary slave will be killed just after a checkpoint has completed across all the nodes. That way the restart position is known, if successful.

■ Checkpointing

The secondary slave is killed shortly after the second checkpoint across the nodes has been performed. The output at the times of the checkpoints (given by debug code, with timing information) is given, with surrounding output from the master node given, so that the restore can be compared to the checkpoint.

- Master Node - rank 0 - moa

```
kao98@moa:/ftmpi/bin > ./pi 100000000
-ftmpi . 0 3 0 moa 1 texel23 2 texel24
3 nodes - 33333333 tests (33333334 local tests)
...
Estimated PI at 13.149367 secs (after 26999997 tests):
 21208351 = 3.141978
Estimated PI at 14.178192 secs (after 29999997 tests):
 23565175 = 3.142024
ckpt: done 3.912365 secs
(1.909480s [0.035682%->48.841966%] in thread sync)
Estimated PI at 10.065443 secs (after 17999997 tests):
 14140523 = 3.142339
...
Estimated PI at 14.178192 secs (after 29999997 tests):
 23565175 = 3.142024
ckpt: done 0.859172 secs
(0.511131s [0.154102%->59.645217%] in thread sync)
Estimated PI at 16.063849 secs (after 32999997 tests):
 25920589 = 3.141890
...
Estimated PI at 19.143357 secs (after 41999997 tests):
 32989373 = 3.141845
Node 2 has not sent a heartbeat recently, aborting
```

- 1st Slave Node - rank 1 - texel23

```
kao98@texel23:/ftmpi/bin > ./pi 100000000 -ftmpi . 1 3
  0 moa 1 texel23 2 texel24
...
ckpt: done 3.911679 secs
  (3.910273s [0.030243%->99.994299%] in thread sync)
ckpt: done 0.858951 secs
  (0.856510s [0.164387%->99.880203%] in thread sync)
Node 2 has not sent a heartbeat recently, aborting
```

- 2nd Slave Node - rank 2 - texel24

```
kao98@texel24:/ftmpi/bin > ./pi 100000000
 -ftmpi . 2 3 0 moa 1 texel23 2 texel24
ckpt: done 3.912212 secs
  (0.509206s [0.030571%->13.046379%] in thread sync)
ckpt: done 0.858809 secs
  (0.857204s [0.128317%->99.941431%] in thread sync)
<<dead>>
```

The first line of output after the last checkpoint is taken on the master node states the estimated value of π ‘after 32999997 tests’. This line is output at time 16.063849s. The last line of output from the master node, before the fault detection kills it, states that it has got up to 19.143357s and has made 41999997 tests.

The ‘ckpt: done’ lines from the debug output give interesting timing information, which will be detailed further in the evaluation chapter (ch. 4). That line states the total duration of the checkpoint, (from initiation to completion, including all stages of synchronization), as well as stating what region of time the local checkpoint was taken in. As you can see the first checkpoint took around 3.9s, and the second checkpoint took around 0.85s. The local section of these checkpoints always started very early in to the checkpoint procedure, but on the longer checkpoints there was a long period after the threads unpaused before the checkpoint finishes. This indicates that either the second barrier is taking a long time to complete; this may be because the message reception servers take time to restart; or one of the machines is getting stuck on the unpausing procedure.

Further analysis will be given in the evaluation chapter (ch. 4).

■ Recovery

In order to recover from a dead checkpointed process, it is necessary to pass the additional ‘-ftmpirecover’ option to FTMPI. The checkpoint file-names are presently a combination of the rank of the process and the date and time at the point of checkpoint.

- Master Node - rank 0 - moa

```
kao98@moa:/pft/ftmpi/bin > ./pi 100000000 -ftmpi . 0 3
0 moa 1 texel23 2 texel24
-ftmpirecover .FTMPIckpt_000_20020609002341
attempting to recover: .FTMPIckpt_000_20020609002341
ckpt: Process restored.
retrieving data...
data retrieved.
Estimated PI at 18.644075 secs (after 32999997 tests):
25921960 = 3.142056
Estimated PI at 19.668733 secs (after 35999997 tests):
28278025 = 3.142003
Estimated PI at 20.693144 secs (after 38999997 tests):
30635998 = 3.142154
...
```

- 1st Slave Node - rank 1 - texel23

```
kao98@texel23:/ftmpi/bin > ./pi 100000000 -ftmpi . 1 3
0 moa 1 texel23 2 texel24
-ftmpirecover .FTMPIckpt_001_20020609002343
attempting to recover: .FTMPIckpt_001_20020609002343
ckpt: Process restored.
retrieving data...
data retrieved.
...
```

- 2nd Slave Node - rank 2 - texel24

```
kao98@texel24:/ftmpi/bin > ./pi 100000000 -ftmpi . 2 3 0
moa 1 texel23 2 texel24
```

```
-ftmpirecover .FTMPIckpt_002_20020609002339
attempting to recover: .FTMPIckpt_002_20020609002339
ckpt: Process restored.
retrieving data...
data retrieved.
...
```

The checkpoint files (all being valid), are correctly restored and the program starts running again. The first line of output from the master node after the restore has been performed is at 18.644075s, and as desired, has the same number of tests as the line immediately preceding the checkpoint in the first run of the application. Note that the time is 2.5s higher after the restore, than the time after the checkpoint. This is because it takes a longer amount of time to perform a restore than to take a checkpoint. However, only about 4s of work have been lost in the recovery.

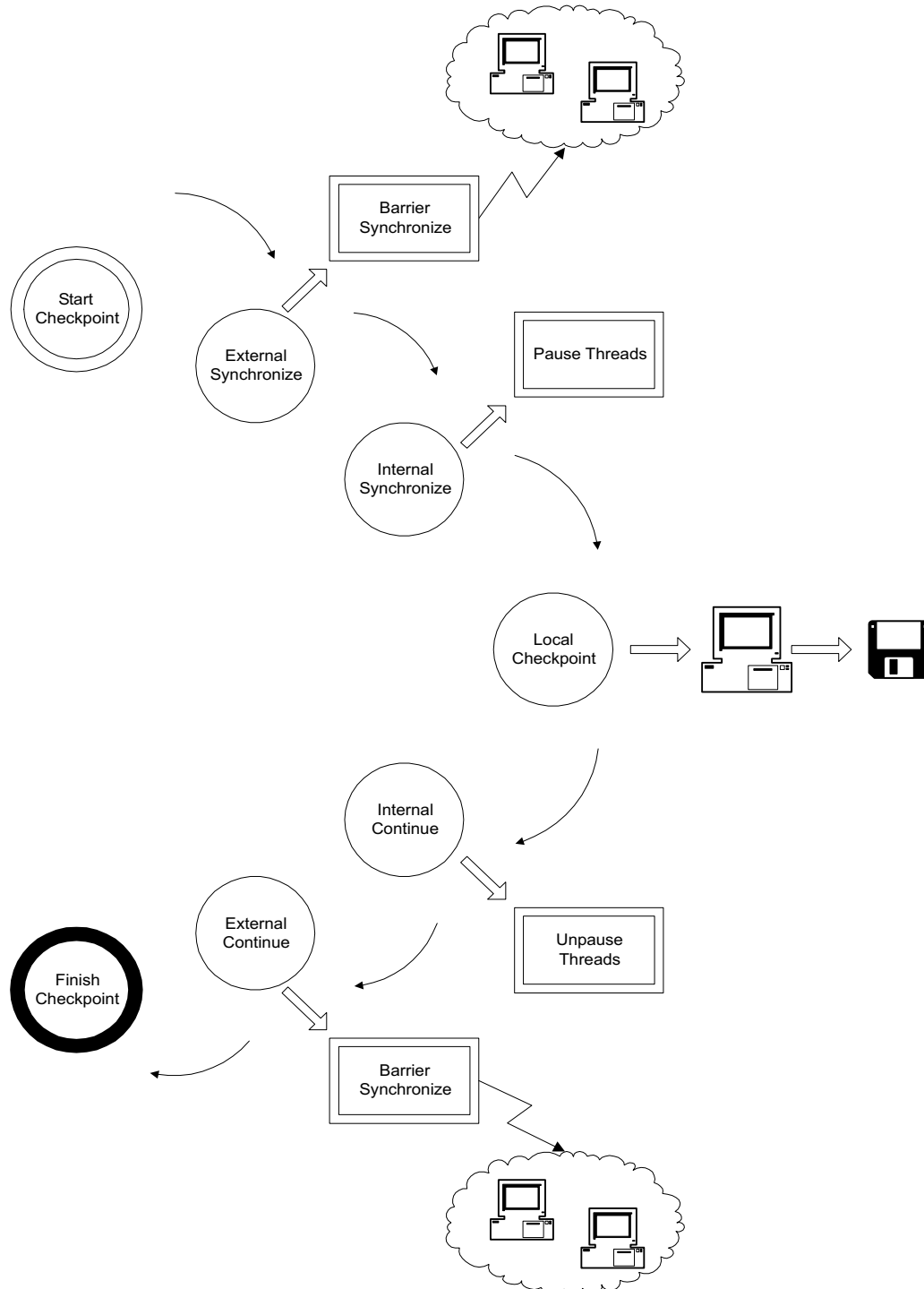


Figure 3.7: Checkpointing Procedure

Chapter 4

Evaluation

4.1 Overview

The evaluation chapter of this report is concerned predominantly with showing the performance of FTMPI. The analysis of how the project compares with other existing systems, what ideas were successful, and which did not live up to expectations is reserved for the general appraisal within the project conclusions chapter (§5.1).

4.2 System Performance

From the outset, no performance targets were set for the FTMPI implementation. The research goals were in other areas, such as examining the techniques employed in parallel checkpointing. However, it is always interesting to see the performance of a system, and how it compares to existing systems. In particular the introduction of checkpointing to programs induces an overhead, this overhead should be looked into and a model produced. Performance analysis also has the benefit of indicating problems with the program, areas where the performance is not good, may need improving.

For all of the performance testing the ‘`pi`’ test program will be used. This program is sufficiently long running to analyse performance without being too complex to separate the issues of the application from the FTMPI system. Additionally, the ‘`pi`’ program already has considerable timing information in its standard output, so it makes a good performance analysis program in its own right. The only additional information displayed is minimal performance statistics from the checkpointing routines.

For all the timing experiments, the `pi` program will perform 200,000,000 (200 million) iterations over the parallel system. The experiment will be run on three nodes and then four nodes - this is to see if general performance differs when the system size increases. For all tests a cut-down version of the program output will be given, with a discussion of the interesting features of the output following each section.

At some point in the future, it would be a good idea to convert some standard industry parallel test applications over to use FTMPI, with strategically placed checkpoints coded in. These test applications, that can be found in test suites like SPEC Bench [24] and Bench Web [25], put a system through a realistic load test, which can accurately show in a standardized fashion how well systems perform on real applications. The reason this has not yet been done, is that the programs in question are often very large, complex programs, and adding checkpointing to them in a user-directed fashion would take far too long for the duration of this project, it should be attempted at a later date.

4.2.1 Raw MPI Performance

In order to analyse the performance degradation inflicted by checkpointing a program, it is necessary to know the ‘raw’ performance of the system without checkpointing. In order to measure this, the checkpointing code from the `pi` program was removed, and the base cases were measured.

In order to compare the performance of the checkpointing system employed by FTMPI, to other existing MPI implementations. It is necessary to compare the raw performance of FTMPI’s MPI implementation to other MPI implementations without checkpointing. The most commonly used implementation of MPI is MPICH (as described in section §2.2) [5]. MPICH is a well established and a reasonably optimized implementation of MPI, so a direct performance comparison between MPICH and FTMPI would give an indication for how FTMPI’s checkpointing system would perform within other MPI implementations. As MPICH is such a well known implementation, performance comparisons between it and other implementations already exist, so a comparison with MPICH is like comparing with a base case - making other comparisons less essential.

As the checkpointing code has already been removed from `pi`, the code uses only MPI standard calls. Therefore the only change needed between the MPICH and FTMPI versions of the program was to the name of the include file, and a recompile.

The output from the base case tests follows, the programs were run on the same set of machines, at a similar time, with practically the same load, so that the tests are fair.

- FTMPI - 3 Nodes

```
kao98@moa:/ftmpi/bin >./pinockpt 200000000
-ftmpi . 0 3 0 moa 1 texel23 2 texel24
3 nodes - 66666666 tests (66666668 local tests)
Estimated PI at 4.889924 secs (after 14999997 tests):
 11781179 = 3.141648
Estimated PI at 9.778096 secs (after 29999997 tests):
 23561923 = 3.141590
...
Estimated PI at 63.852587 secs (after 194999997 tests):
 153156386 = 3.141670
Estimated PI (after 200000000 tests): 157081964 = 3.141639
The operation took 65.483510 ticks (0 ckpts)
```

- MPICH - 3 Nodes

```
kao98@moa:/ftmpi/bin >mpirun -np 3 -machinefile texels
/homes/kao98/lab/fourthyear/pft/ftmpi/bin/pinockpt 200000000
3 nodes - 66666666 tests (66666668 local tests)
Estimated PI at 4.206711 secs (after 14999997 tests):
 11781360 = 3.141697
Estimated PI at 8.324581 secs (after 29999997 tests):
 23563033 = 3.141738
...
Estimated PI at 54.064623 secs (after 194999997 tests):
 153154813 = 3.141637
Estimated PI (after 200000000 tests): 157082112 = 3.141642
The operation took 55.410974 ticks (0 ckpts)
```

- FTMPI - 4 Nodes

```
kao98@moa:/ftmpi/bin >./pinockpt 200000000
-ftmpi . 0 4 0 moa 1 texel23 2 texel24 3 texel25
4 nodes - 50000000 tests (50000000 local tests)
Estimated PI at 4.890492 secs (after 19999996 tests):
```

```

15708554 = 3.141711
Estimated PI at 9.781320 secs (after 39999996 tests):
31419099 = 3.141910
...
Estimated PI at 49.204781 secs (after 199999996 tests):
157081377 = 3.141628
Estimated PI (after 200000000 tests): 157081380 = 3.141628
The operation took 49.206869 ticks (0 ckpts)

```

- MPICH - 4 Nodes

```

kao98@moa:/ftmpi/bin >mpirun -np 4 -machinefile texels
/homes/kao98/lab/fourthyear/pft/ftmpi/bin/pinockpt 200000000
4 nodes - 50000000 tests (50000000 local tests)
Estimated PI at 4.153177 secs (after 19999996 tests):
15710678 = 3.142136
Estimated PI at 8.329304 secs (after 39999996 tests):
31417244 = 3.141725
...
Estimated PI at 41.563190 secs (after 199999996 tests):
157076491 = 3.141530
Estimated PI (after 200000000 tests): 157076494 = 3.141530
The operation took 41.564261 ticks (0 ckpts)

```

As expected the MPICH implementation is slightly faster than the FTMPI implementation, but surprisingly not by much. For the three node test case, MPICH is ten seconds faster than FTMPI (55s versus 65s), 18% faster. For the four node test case, MPICH is just under eight seconds faster than FTMPI (41s versus 49s), 20% faster. The results from these tests are shown in Figure 4.1, and also included as part of later results in Table 4.1.

The results seem to show that FTMPI is around 20% slower than MPICH, which was to be expected. The timings coming from the programs were very consistent, MPICH was always, at every step of the program, the same percentage ahead (time wise) of FTMPI - indicating that the performance difference is uniform over the system, and is not particular to any section of the code, but merely to general message transmission and reception.

4.2.2 Checkpointing Performance

The introduction of checkpointing to a program, will almost certainly effect its performance. The only checkpointing systems that don't noticeably ef-

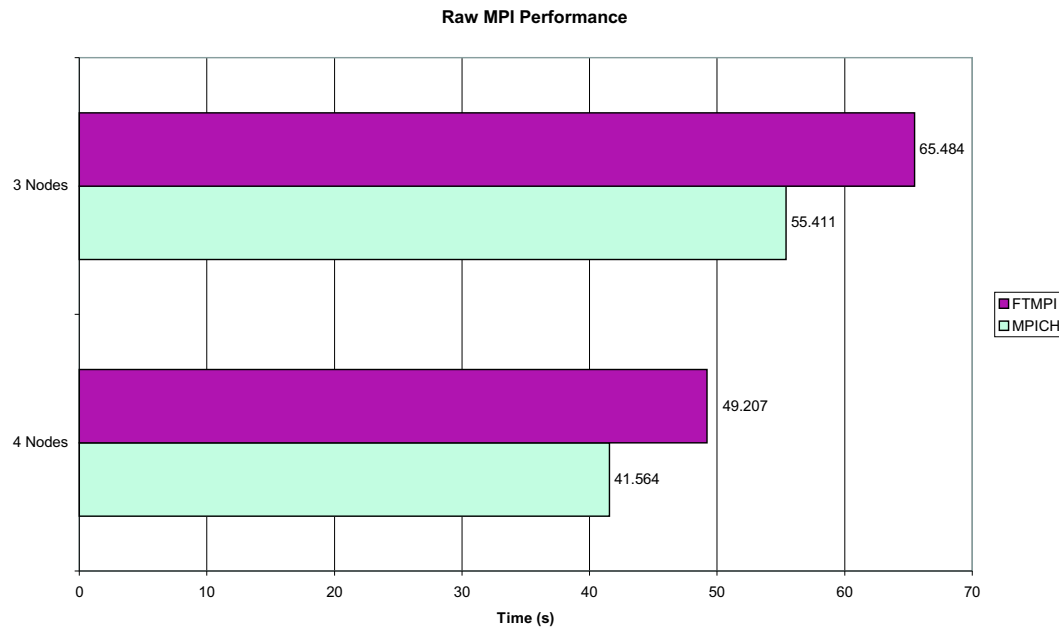


Figure 4.1: FTMPI v MPICH - Raw Performance

fect performance, are the systems that perform asynchronous checkpointing (where the system continues whilst the checkpoint is being performed), those systems are highly complicated however, and are not being examined. The aim of this section is to see what overhead is imposed by the checkpointing system of FTMPI, and to examine the checkpointing mechanism for performance problems.

As before, the experiments are all conducted on both a three node set, and a four node set. Minimal output is shown from each node for the tests, the lines of interest being the final duration of the program, and the checkpoint timing diagnostic lines (lines stating with ‘ckpt:’).

In the `pi` program, checkpoints are taken at regular intervals in the iterative cycle. They are taken when each node gets to the same number of iterations, so the interval must be set with the number of nodes and the problem size in mind if you wish to control how many checkpoints are taken during the test. The checkpointing interval is specified by the variable `CKPT_STEP` in the program, the values for this variable are listed with each test. The tests will be performed with both a 2 checkpoint (checkpointing at every third of the program), and a 4 checkpoint (checkpointing at every fifth of the program) approach.

- 3 nodes - 2 checkpoints - CKPT_STEP 26666667

– Master Node - rank 0 - moa

```
kao98@moa:/ftmpi/bin > ./pi 200000000
-ftmpi . 0 3 0 moa 1 texel23 2 texel24
3 nodes - 66666666 tests (66666668 local tests)
Estimated PI at 4.909600 secs (after 14999997 tests):
 11783134 = 3.142170
...
Estimated PI at 24.578328 secs (after 74999997 tests):
 58904070 = 3.141551
ckpt: done 1.566524 secs
(0.508809s [42.664523%->75.144651%] in thread sync)
Estimated PI at 30.390095 secs (after 89999997 tests):
 70681234 = 3.141388
...
Estimated PI at 50.262637 secs (after 149999997 tests):
 117808248 = 3.141553
ckpt: done 4.727756 secs
(2.577147s [24.351574%->78.862572%] in thread sync)
Estimated PI at 58.751089 secs (after 164999997 tests):
 129587726 = 3.141521
...
Estimated PI (after 200000000 tests): 157074614 = 3.141492
The operation took 70.195859 ticks (2 ckpts)
```

– 1st Slave Node - rank 1 - texel23

```
kao98@texel23:/ftmpi/bin > ./pi 200000000
-ftmpi . 1 3 0 moa 1 texel23 2 texel24
ckpt: done 0.898879 secs
(0.897640s [0.104241%->99.966403%] in thread sync)
ckpt: done 3.577458 secs
(0.507955s [0.009700%->14.208469%] in thread sync)
```

– 2nd Slave Node - rank 2 - texel24

```
kao98@texel24:/ftmpi/bin > ./pi 200000000
-ftmpi . 2 3 0 moa 1 texel23 2 texel24
ckpt: done 0.902383 secs
(0.896683s [0.495244%->99.863583%] in thread sync)
```

```
ckpt: done 3.579312 secs
      (3.575645s [0.095633%→99.993183%] in thread sync)
```

- 3 nodes - 4 checkpoints - CKPT_STEP 13333335

– Master Node - rank 0 - moa

```
kao98@moa:/ftmpi/bin > ./pi 200000000
-ftmpi . 0 3 0 moa 1 texel23 2 texel24
3 nodes - 66666666 tests (66666668 local tests)
Estimated PI at 4.901887 secs (after 14999997 tests):
 11782096 = 3.141893
Estimated PI at 9.815689 secs (after 29999997 tests):
 23565836 = 3.142112
ckpt: done 3.857484 secs
      (2.528596s [34.447479%→99.997874%] in thread sync)
Estimated PI at 17.252432 secs (after 44999997 tests):
 35347585 = 3.142008
...
Estimated PI at 27.076334 secs (after 74999997 tests):
 58908853 = 3.141806
ckpt: done 1.696036 secs
      (0.901038s [46.869288%→99.995401%] in thread sync)
Estimated PI at 32.899445 secs (after 89999997 tests):
 70691748 = 3.141856
...
Estimated PI at 42.734685 secs (after 119999997 tests):
 94258422 = 3.141947
ckpt: done 1.879604 secs
      (0.877718s [0.051128%→46.748092%] in thread sync)
Estimated PI at 49.514954 secs (after 134999997 tests):
 106037716 = 3.141858
Estimated PI at 54.422451 secs (after 149999997 tests):
 117818198 = 3.141819
ckpt: done 2.259746 secs
      (0.496465s [60.841130%→82.811077%] in thread sync)
Estimated PI at 60.267473 secs (after 164999997 tests):
 129600643 = 3.141834
...
Estimated PI (after 200000000 tests): 157087170 = 3.141743
The operation took 71.719634 ticks (4 ckpts)
```

– 1st Slave Node - rank 1 - texel23

```
kao98@texel23:/ftmpi/bin > ./pi 200000000
-ftmpi . 1 3 0 moa 1 texel23 2 texel24
ckpt: done 2.533392 secs
(1.528346s [0.182996%->60.511046%] in thread sync)
ckpt: done 0.902183 secs
(0.508552s [0.087565%->56.456617%] in thread sync)
ckpt: done 1.878927 secs
(1.877990s [0.037894%->99.988025%] in thread sync)
ckpt: done 0.885847 secs
(0.474395s [0.047977%->53.600678%] in thread sync)
```

– 2nd Slave Node - rank 2 - texel24

```
kao98@texel24:/ftmpi/bin > ./pi 200000000
-ftmpi . 2 3 0 moa 1 texel23 2 texel24
ckpt: done 2.530055 secs
(2.527764s [0.039604%->99.949052%] in thread sync)
ckpt: done 0.902406 secs
(0.506405s [0.135194%->56.252396%] in thread sync)
ckpt: done 1.879315 secs
(1.877189s [0.037354%->99.924228%] in thread sync)
ckpt: done 0.926393 secs
(0.883984s [4.547206%->99.969343%] in thread sync)
```

• 4 nodes - 2 checkpoints - CKPT_STEP 20000001

– Master Node - rank 0 - moa

```
kao98@moa:/ftmpi/bin > ./pi 200000000
-ftmpi . 0 4 0 moa 1 texel23 2 texel24 3 texel25
4 nodes - 50000000 tests (50000000 local tests)
Estimated PI at 5.189225 secs (after 19999996 tests):
15707757 = 3.141552
...
Estimated PI at 20.985766 secs (after 79999996 tests):
62829612 = 3.141481
ckpt: done 3.383507 secs
(0.513704s [0.032333%->15.214923%] in thread sync)
Estimated PI at 29.549819 secs (after 99999996 tests):
78536659 = 3.141466
```

```

...
Estimated PI at 45.082303 secs (after 159999996 tests):
 125662235 = 3.141556
ckpt: done 2.269917 secs
  (0.504084s [0.046433%->22.253589%] in thread sync)
Estimated PI at 52.531149 secs (after 179999996 tests):
 141371263 = 3.141584
...
Estimated PI (after 200000000 tests): 157079198 = 3.141584
The operation took 57.705764 ticks (2 ckpts)

```

– 1st Slave Node - rank 1 - texel23

```

kao98@texel23:/ftmpi/bin >./pi 200000000
-ftmpi . 1 4 0 moa 1 texel23 2 texel24 3 texel25
ckpt: done 3.383234 secs
  (0.513979s [0.034907%->15.226851%] in thread sync)
ckpt: done 2.269666 secs
  (0.504812s [0.039653%->22.281340%] in thread sync)

```

– 2nd Slave Node - rank 2 - texel24

```

kao98@texel24:/ftmpi/bin >./pi 200000000
-ftmpi . 2 4 0 moa 1 texel23 2 texel24 3 texel25
ckpt: done 3.382769 secs
  (3.381612s [0.025334%->99.991132%] in thread sync)
ckpt: done 2.269261 secs
  (2.268027s [0.040454%->99.986075%] in thread sync)

```

– 3rd Slave Node - rank 3 - texel25

```

kao98@texel25:/ftmpi/bin >./pi 200000000
-ftmpi . 3 4 0 moa 1 texel23 2 texel24 3 texel25
ckpt: done 3.383280 secs
  (0.510693s [0.035705%->15.130317%] in thread sync)
ckpt: done 2.269636 secs
  (0.508043s [0.052872%->22.437210%] in thread sync)

```

- 4 nodes - 4 checkpoints - CKPT_STEP 10000001

– Master Node - rank 0 - moa

```

kao98@moa:/ftmpi/bin > ./pi 200000000
-ftmpi . 0 4 0 moa 1 texel23 2 texel24 3 texel25
4 nodes - 50000000 tests (50000000 local tests)
Estimated PI at 5.171462 secs (after 19999996 tests):
15707394 = 3.141479
Estimated PI at 10.356002 secs (after 39999996 tests):
31417409 = 3.141741
ckpt: done 3.426443 secs
(1.424557s [0.039283%->41.614672%] in thread sync)
Estimated PI at 18.979670 secs (after 59999996 tests):
47122636 = 3.141509
Estimated PI at 24.158693 secs (after 79999996 tests):
62832022 = 3.141601
ckpt: done 2.624696 secs
(0.529747s [0.040233%->20.223409%] in thread sync)
Estimated PI at 32.117011 secs (after 99999996 tests):
78537820 = 3.141513
Estimated PI at 37.342333 secs (after 119999996 tests):
94247644 = 3.141588
ckpt: done 1.457634 secs
(0.543905s [1.685608%->38.999845%] in thread sync)
Estimated PI at 43.962622 secs (after 139999996 tests):
109954473 = 3.141556
Estimated PI at 49.144433 secs (after 159999996 tests):
125662492 = 3.141562
ckpt: done 0.636715 secs
(0.635125s [0.234642%->99.984923%] in thread sync)
Estimated PI at 54.959161 secs (after 179999996 tests):
141372570 = 3.141613
Estimated PI at 60.152194 secs (after 199999996 tests):
157084998 = 3.141700
Estimated PI (after 200000000 tests): 157085001 = 3.141700
The operation took 60.154138 ticks (4 ckpts)

```

– 1st Slave Node - rank 1 - texel23

```

kao98@texel23:/ftmpi/bin > ./pi 200000000
-ftmpi . 1 4 0 moa 1 texel23 2 texel24 3 texel25
ckpt: done 3.426352 secs
(0.515078s [0.047047%->15.079887%] in thread sync)
ckpt: done 2.624527 secs

```

```

    (1.620073s [0.034063%→61.762253%] in thread sync)
ckpt: done 1.433240 secs
    (0.510273s [0.057632%→35.660392%] in thread sync)
ckpt: done 0.636755 secs
    (0.634356s [0.195523%→99.818769%] in thread sync)
- 2nd Slave Node - rank 2 - texel24

kao98@texel24:/ftmpi/bin >./pi 200000000
-ftmpi . 2 4 0 moa 1 texel23 2 texel24 3 texel25
ckpt: done 3.426345 secs
    (3.423604s [0.044742%→99.964744%] in thread sync)
ckpt: done 2.624542 secs
    (2.621553s [0.033682%→99.919796%] in thread sync)
ckpt: done 1.433781 secs
    (0.509579s [0.098899%→35.639822%] in thread sync)
ckpt: done 0.636756 secs
    (0.513722s [0.211541%→80.889540%] in thread sync)
- 3rd Slave Node - rank 3 - texel25

kao98@texel25:/ftmpi/bin >./pi 200000000
-ftmpi . 3 4 0 moa 1 texel23 2 texel24 3 texel25
ckpt: done 3.426362 secs
    (3.423566s [0.054460%→99.972858%] in thread sync)
ckpt: done 2.624093 secs
    (2.622789s [0.038871%→99.989177%] in thread sync)
ckpt: done 1.433311 secs
    (1.431518s [0.106048%→99.980953%] in thread sync)
ckpt: done 0.636744 secs
    (0.633641s [0.213430%→99.726107%] in thread sync)

```

If, for the moment, we restrict our attention purely to the total duration of the programs, then we see that the overhead imposed by the checkpointing is not very high. Table 4.1 summarises the total run times of the tests, this information is also shown in more detail in Figure 4.2. The overhead values given in Table 4.1 are with respect to the base case (no checkpointing) with FTMPI, the overhead for the base case is with respect to the MPICH performance.

The overheads for the four node tests are substantially larger than the overheads for the three node tests. This is because the speed of a checkpoint

is roughly constant regardless of the size of the system, as the speed of the barrier synchronization is usually dwarfed by the speed of the thread synchronization. The less time a program takes to execute, the more of an overhead checkpointing will be. As this program runs faster with more nodes, it makes sense that the overhead also grows larger, as it is fractionally of much larger impact to the program. The increase in the overhead from two checkpoints per program to four however is not very significant in either case. Rather oddly, the checkpoints often seem to increase in speed with progressive checkpoints in these tests. There is no reason for this, as far as the code is concerned, as the checkpoints should remain at roughly a constant speed, but the checkpoints often get faster as more checkpoints are taken. For instance, if we look at the last set of program output from the fourth slave node, for four nodes, four checkpoints, we see that the checkpoint duration is steadily decreasing at each stage. This must be due to random chance.

Nodes	Implementation	Checkpoints	Time (s)	Overhead %
3	MPICH	0	55.411	-
	FTMPI	0	65.484	18.18%
		2	70.196	7.20%
		4	71.720	9.52%
4	MPICH	0	41.564	-
	FTMPI	0	49.207	18.39%
		2	57.706	17.27%
		4	60.154	22.25%

Table 4.1: Total Run Times

The fact that the time taken to perform a checkpoint varies, is no surprise, as the activities that make up the checkpoint are not constant themselves. The areas that make up the duration of a checkpoint are:

- the time taken from the first node to call the checkpointing routine, to the last node calling it
- the time taken to perform a barrier across the network, either side of the local checkpoint
- the time taken to pause, and unpause the threads locally
- and the time taken to actually perform the local checkpoint

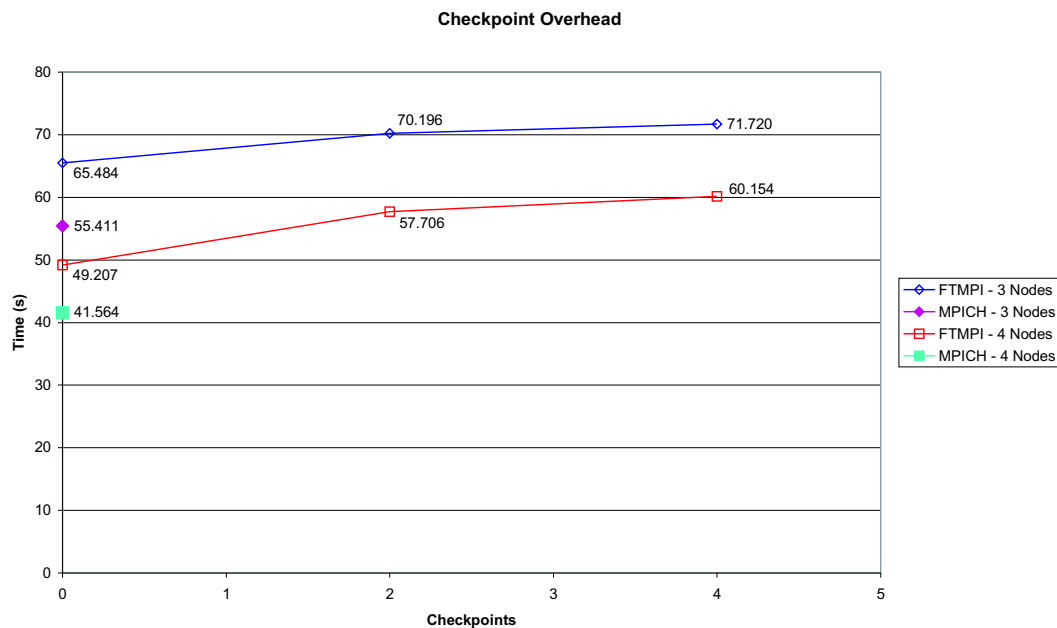


Figure 4.2: Checkpoint Overhead

In order to get some idea of what is causing the fluctuations, timing information was output after every checkpoint within the tests. Each line is of the form:

```
ckpt: done #t# secs (#l#s [#s#%->#f#%] in thread sync)
```

Where `#t#` is the total duration of the checkpoint, `#l#` is the duration of the local section, and `#s#` and `#f#` are the start and finish positions of the local section within the entire checkpoint (given as a percentage of the way through).

A summary of the average total checkpoint durations, and the average percentage of time spent in the local routine is given for each for each test in Table 4.2.

For each individual checkpoint across the system, it is quite clear that the overall time of the checkpoint, (which is roughly consistent over all machines, as it should be), is set by the machine which takes longest in its local section. There is almost always, in all but the quickest checkpoints, a machine that spends nearly 100% of the time (or at least above 95%) in its local section, holding up the other nodes in the system. The local section is made up of

pausing and unpausing the threads and taking the local checkpoint. The actual checkpoint itself is a very quick, basic, operation, which should not really take much longer than the time it takes to output the data buffer, specified by the program, to the disk. Therefore, it should be assumed that the problem lies within the thread pausing routine, and that this is the major bottleneck in checkpointing speed.

The thread pausing routine seems to take longer than expected on occasion to complete, as the yield points are set to once every half second, for each thread. The pause should theoretically take no more than half a second, at its quickest, this is what happens, however, the internal pause section can take nearly three and a half seconds in the worst case. Something is not quite right here, what is wrong is unknown at present. Although the most likely explanation is that a thread is not yielding quite as often as it should do, due to being occupied with sending or receiving messages (no yield points are made whilst any of the threads are working, only in between work), perhaps this needs to be altered.

In all cases, checkpoints seem to average out somewhere between one and three seconds in duration for these tests. As `pi` stores relatively little data with each checkpoint, this time would correspond roughly with the base execution time of a checkpoint. There is little difference if any between the speed of checkpointing, with three or four nodes, so the scaling should not be an issue. A little oddity is that the master node appears to take slightly longer to perform the checkpoint than the slave nodes in this program, why this is is unclear. The master node neither stores more information than the others, nor operates in any different manner, it has exactly the same amount of communication and work to do as the rest.

Because checkpointing with FTMPI appears to be a very constant time operation, it is possible to produce a basic model of the overhead incurred by checkpointing. If you have a program that performs ' N ' checkpoints during its operation, where its normal execution time is ' T ' seconds, the base overhead of the checkpointing system (maximum synchronization time) is ' b ' seconds, and at each checkpoint the data buffer takes ' d ' seconds to save, then the overhead ' O ' (as a fraction) is given by:

$$O = \frac{T + N(b + d)}{T} - 1$$

This assumes that the checkpoint has roughly constant time, regardless of system size (number of nodes), as the local section of the checkpoint dwarfs the remote section.

# Nodes	Checkpoints	Node	Avg. Time (s)	Avg. Local %
3	2	A	3.147	43.50%
		B	2.238	57.03%
		C	2.241	99.63%
	4	A	2.423	46.84%
		B	1.550	67.55%
		C	1.560	87.83%
4	2	A	2.827	18.69%
		B	2.826	18.72%
		C	2.826	99.96%
		D	2.826	18.74%
	4	A	2.036	49.71%
		B	2.030	53.00%
		C	2.030	79.01%
		D	2.030	99.81%

Table 4.2: Checkpoint Durations

Chapter 5

Conclusions & Future Work

5.1 General Appraisal

The main goal of FTMPI, was to produce an implementation of the MPI interface that had fault tolerance by means of user-directed checkpointing, this has been achieved. However, this section details how good FTMPI actually is, compared to existing techniques, and in terms of how well it solves the problem.

The basic implementation of the MPI interface within FTMPI, is at present, greatly reduced from the entire specification. The MPI interface as standard has some 214 commands, FTMPI only implements 14 of these, representing the most basic set required for normal parallel programming. For simple applications this basic set is sufficient, and in fact, is sufficient for almost any parallel programming, as the other commands can be built from the existing ones.

A notably lacking feature within the FTMPI system is the provision for communicators and groups. Communicators are a very useful tool for splitting up a large set of machines into smaller subsets, this is essential if you need to use collective routines like `MPI_Barrier` and `MPI_Reduce` over less than the entire set of nodes. The lack of this functionality is certainly the biggest drawback to the MPI subsystem of FTMPI. However it would not require a great deal of work to add these features, as the basic communicator infrastructure is already established within the code. Additionally many of the collective operations within the MPI standard have yet to be implemented, such as `MPI_Gather`, `MPI_Scan` and `MPI_Scatter`. These routines can be implemented by hand, using the existing primitives, but it would be

best if they were included in the system. Other desired missing functionality includes `MPI_Wait`, and the `MPI_PACKED` data type and associated functions. A minor difference between FTMPI and a normal MPI implementation is that FTMPI is not designed to work with a single node, although in practice it may well do so, the execution is undefined. FTMPI is supposed to be for parallel programming, and there is no parallelism in a single node machine, so it is not felt necessary to support this explicitly. See the future work section (§5.3) for more information on enhancements to the MPI subsystem.

The decision to pass the MPI system state to each program, entirely via the command line, may also not be a particularly good idea. Even with a small number of nodes the command line gets very long. If there were a vast quantity of nodes in the system, the command line would be exceptionally long and some environments might not be able to handle it correctly. MPICH [5], solves this problem by putting all of the MPI system configuration in to a globally accessible file which all the programs read. This is not entirely desirable, as globally accessible storage space may not be available. An alternative to this would be for the system configuration to be passed to the master node (which is always on the same machine as the startup script) via a local file, and the master node distributing the information out to the client nodes somehow. If all of the slave nodes were supplied with the master machine details, then they could initially setup their MPI systems to just include the master, for the sake of receiving the rest of the MPI system structure, then reconstructing the master communicator to include all of the nodes. This approach would solve both issues of local files and command line length.

As the MPI interface of FTMPI is identical to the standard MPI interface, most simple MPI programs written in C can be ported to FTMPI simply by changing the included MPI header file, and recompiling with the correct libraries. This makes FTMPI highly usable as almost no code modification is required to use its basic facilities; the checkpointing code addition is another matter.

FTMPI uses a totally user-directed checkpointing approach, as stated before, this means that not only is the location of the checkpoint explicitly made, the data saved with the checkpoint is also specified explicitly. The theoretical benefits of user-directed checkpointing, compared to transparent checkpointing, have already been laid out in the background material (§2.4), but whether or not the benefits are worthwhile is a matter of some debate.

The idea of specifying checkpoint locations is definitely a good one, not only does it massively simplify the task of synchronizing the parallel system,

as the system is guaranteed not to be in the middle of any other operation, it also gives the programmer some idea of when and where a program is currently able to be restored to, this is an underrated feature of fault tolerance. If for instance, you need to shut down a machine, but a long running parallel job is running on it, it would be nice to know when a checkpoint will be taken so that you can plan for when the shutdown can be made, or at least know how much work will be lost since the last checkpoint if you go ahead and kill the application. With the added bonus of being able to reduce the amount of data stored in a checkpoint, when you know what is not required at the point of checkpoint, specifying the checkpoint location is certainly preferable to transparent checkpointing. The only instance where transparent checkpointing would be better, is if there is either no good place to perform a checkpoint, and it might as well go anywhere in the program, or the existing code cannot be modified for whatever reason.

Explicitly constructing the saved data buffer in the program however, has not worked as well as could have been hoped. The task of adding checkpointing to a program is not difficult, but can be quite a tedious job if there is a large amount of data to be stored, restoring data is particularly arduous as all structures need to be reconstructed (often requiring a lot of work). Specifying which data to include in the checkpoint certainly does help the size of the checkpoint. If the checkpoint is placed in a good position within the program and the data buffer handled well, the size of the checkpoint can be insignificant compared to the entire memory space of the program. On reflection, schemes employed by checkpointers like `Libckpt` [16], where it saves the entire memory area, but unneeded (dead) areas can be excluded, may be better. They are certainly simpler from the programmer's perspective, but don't offer quite the same level of checkpoint size reduction. Additionally, with the threaded nature of an FTMPI program, heap saving is exceptionally difficult. A possible alternative solution would be to provide a "compiler assisted" form of the checkpointing, maybe by the use of a preprocessor. If the data that was required for checkpointing could be marked some how, then an auxiliary preprocessor could add the necessary data buffer storing and restoration code.

Whilst the checkpointing method works adequately, the existing recovery method is less satisfactory. At present, when a node fails, the entire system must stop and be restarted again, this is not really an automated process - so the system cannot be left unattended in the event of a failure. This is because persistent machine failures may cause problems, as the harness is relatively undeveloped and does not presently select alternative machines intelligently, or automatically. Other systems in current use lose less time

on failures, because they recover very swiftly. In most existing systems the recovery time is only different because of the lack of automation. But some systems, such as the PREACHES system [12], recover very swiftly indeed as they do not have to shut the system down in order to restart a single node, another node is simply generated on the fly. Whilst systems that can cope with a dynamically changing environment, without a full restart, can be very useful - they often introduce a lot of issues that the programmer has to be aware of. Due to this, systems like that are predominantly used for SPMD style parallel programming (eg. batch data processing), in contrast FTMPI is perfectly able to cope with full MPMD style programming, as there is no actual need to even have the same executable or architecture for every node, as long as the restore can be performed on a per rank basis with the same software.

5.2 Portability

At present the system is not completely portable, as it is coded entirely with the current Linux distribution and the Ix86 (Intel) architecture in mind.

The majority of the program is, however, portable. All of the source code is written in ANSI standard C, and the main library uses the common library linking mechanism employed by most C implementations, so should not be restricted to use only with 'gcc'. All communications are performed using TCP/IP, using the UNIX Sockets interface [28] which is implemented on most platforms. The threading of the code uses only standard 'pthreads' calls, so should be entirely portable to any environment supporting the pthreads library.

The areas that would need addressing, if the program was to be ported, are as follows:

- state saving & restoration

This is very particular to the architecture, and the operating system, the code is running on. If FTMPI was ported to another OS or architecture, the state saving and restoration code would almost certainly have to be altered, if not completely rewritten.

- poll() - socket selection

This program uses the Linux implementation of poll(), this routine is not available on all operating systems, and differs in functionality on

those platforms that do support it. A more portable implementation might have used the `select()` command for socket selection, but that command was not as suitable for the purpose here.

- `gettimeofday()`, and other `glibc` functions

The code makes use of a number of `glibc` functions, which are fairly standard on most systems. However there are often small differences between platforms that would need to be addressed if porting. Specifically it is known that the `gettimeofday()` routine is different on many systems and would need altering.

5.3 Future Work

This section outlines possible future enhancements to FTMPI, over and above the fixes listed in the “Problems, Bugs and Omissions” section below (§5.4), as these are enhancements, not fixes. FTMPI is presently only in its infancy, and as such, there is much that could be enhanced with the system.

5.3.1 MPI Implementation Extensions

As mentioned in the appraisal (§5.1), the MPI implementation is not yet complete, there are many commands in the basic MPI interface that have yet to be added. In particular, full support for communicators is almost essential, and therefore is *the* most important addition that can be made to the MPI implementation. The existing implementation of the communicators is partially complete, communicators do physically exist within the system and the structure is present for more communicators to be added. However, the notion of local group ranks, versus global ranks would need to be addressed, particularly as global ranks are presently used for communication addresses. At a basic level this could just be implemented by using references to the global communicator structure, but in practice it would be better to separate the global structure from the communicator system, and use groups entirely for all communicator functions, only referring to the real global data, with no master communicator object. The essential MPI commands needing to be implemented would be:

- `MPI_Comm_create` - creates a new communicator
- `MPI_Comm_free` - destroys an existing communicator

- `MPI_Comm_group` - extracts the group associated with the given communicator
- and all the `MPI_Group` command set, as these are needed for forming new groups from existing ones.

As for the inclusion of the more advanced collective operations, these can mainly be implemented using the send and receive primitives in the same way that the existing collective operations are performed. Specifically, `MPI_Gather` (and its variants) is a non-folding version of a reduce, it simply collects data. `MPI_Scatter` is not significantly different from a broadcast, and `MPI_Scan` is a modified version of a reduce. These useful additions would not be difficult to add.

5.3.2 Checkpointing Improvements

The method of checkpointing in FTMPI is presently not as advanced as it could be. As mentioned at various stages within this report, there are a number of techniques that could possibly be employed to enhance the functionality or the stability of the checkpointing mechanism.

One of the key problems with the checkpointing in FTMPI programs, was the fact that the individual programs are threaded. An obvious solution is to remove threading from the application, a non-trivial task as all the communication servers would have to work on interrupts and timer calls (see the message passing system design section (§3.2.3) for a discussion). The method of hacking the threading library to support real threaded checkpointing is highly difficult and not desirable, so the only remaining solution is the one employed by FTMPI. This precludes the saving of the heap though, as internal data from the threading library gets corrupted. There may however, be a way round this.

By using a very advanced method of extracting just the pages of memory we need from the heap, and excluding (at least for the restore) the pages associated with the threading library (or any other corrupted area), it may be possible to save and restore the heap without corrupting things. This method uses detailed scanning of the memory map tables for the program in question and requires some knowledge of what things are volatile and need to be protected. Debugging information within the code may be able to help with this. These maps can also be used to extract the stack from the system

in a more accurate manner, so their inclusion would be a useful enhancement to FTMPI. These matters are discussed further in the checkpointing implementation section (§3.3.6).

In the discussion of the restoration implementation (§3.3.6), an alternative approach to saving the stack was mentioned. This is where a temporary stack is used whilst the stack restoration is performed, in order to prevent the restored stack from being corrupted whilst it is being restored. This addition to the system is not particularly complex and could be added without significant modification to the existing code.

The appraisal (§5.1) states that the existing interface for saving data that is required in the checkpoint, is too cumbersome and long winded. A useful extension would be to provide a basic level of compiler-assisted checkpointing, by creating a preprocessor that could automatically generate the necessary data saving commands around any call to perform a checkpoint. As data requirements change from checkpoint to checkpoint this may need to be specified in quite an advanced way, possibly through some advanced macro language embedded in the code, or via an external script specifying data requirements at each checkpoint. A utility would need to be developed to implement this sort of checkpointing, which would be a very useful addition to the project.

The last set of improvements would be to the recovery harness, ‘`ftmpirun`’. This utility has been neglected in the development of FTMPI and does not handle recovery in a decent manner at present. The problem is serious enough that it is mentioned in the bugs section (§5.4) that follows. The utility must be expanded to deal with automated detection of failures, selection of checkpointing files and automated (unattended) restarts, intelligently picking machines so as to avoid nodes that have failed. It might also be an idea for the script to have knowledge of the architectures and performance of nodes, so that process migration across not entirely homogenous machines can be made, although this is verging in to the territory of a process management system (like MPVM [11]), which is not the aim of the FTMPI system.

5.4 Problems, Bugs and Omissions

Unfortunately, as with any large system, especially a non-production level system like this, FTMPI has a few problems with its current implementation. Most of these problems do not affect the general running of the system, but should be noted here.

- The first issue, as mentioned in the appraisal (§5.1), is the lack of a good automated restart mechanism. The fault reporting system should return greater detail, where possible, to the harness, and the harness needs to be adapted to provide a more intelligent restart system. In particular the harness needs to be able to remove currently unusable machines from its pool, and be able to test to see whether or not the program fails from a non-network failure (code bug), which should not be dealt with.
- At present the filenames of checkpoints are not entirely useful, they contain the rank and the time a checkpoint was taken, but not a lot more. At the moment it is not possible to run two checkpointing programs simultaneously, using the same working directory - as the checkpoint files for one system will be indistinguishable from the other. Additionally, the times on the files are not synchronized, so there is no immediate way to identify a set of checkpoint files for a particular checkpoint. At present, what is done, is to find the checkpoint file with the latest time for each node and use that for restarting, this is unsatisfactory. The checkpoints should be counted and a checkpoint identifier included in the checkpoint files, both in the file and in the filename.
- Currently, there is a small danger with restarts. If a restart is attempted before all of the nodes have died, then the systems will reconnect to each other and the system will appear to have come back online (as the heartbeat systems will be interacting again). As there is no session indicator, the systems will be unaware that they are at different stages and will inevitably lock-up or crash. To fix this problem the system needs to be extended with session indicators, so that messages from one session are not handled by another. A session indicator could be formed simply by generating a long random number at the start of each session and distributing it to the nodes via the startup mechanism.
- Another danger with restarts is that FTMPI currently does not detect whether or not the binary in use is the correct one for the checkpoint. If it is not, then the restore will be complete nonsense and the program will almost certainly crash straight away. To fix this, the checkpoint files should contain details about the executable to which they belong, possibly including the file name, and definitely the file size and a CRC (or other hash) of the file, to check for any changes. A recompiled binary, with any modifications will alter the location of many items within the program, so there is no room for error, therefore any change to the binary must invalidate the checkpoint.

- At present FTMPI has no security features. No checking is performed on packets to ensure they originate from where they are suppose to and it would be very simple for a machine to masquerade as part of the system or interfere with its operation. Most MPI implementations make no provision for security, so this is not a major problem. However, a secure system would be better, so this could be a useful future system improvement.
- As stated earlier, FTMPI does not at present support single node systems. Single node systems are often used in testing as a base case, so making sure that the system can handle this case would be very useful. As it stands the behaviour of the system when used with a single node is undefined, although it may work for some cases, however this cannot be guaranteed.
- Some MPI implementations allow multiple nodes to work on the same machine (this is particularly useful on SMP machines). FTMPI does not currently support this as it only uses TCP/IP using fixed ports for node-to-node communications, so cannot handle two programs on the same IP address. To solve this, nodes would have to be able to interact with each other on the same machine somehow, probably by inter-process communications and share the external communications system. This would require an extensive rewrite of the basic system, so is unlikely to ever be implemented.
- As noted in the checkpointing performance section (§4.2.2), the thread pausing mechanism is currently very slow at times. The maximum time for a thread to yield should be given by `FTMPI_CHECKPOINT_YIELDTIME`, but this is not being held to. As stated, this is probably because the system is being held up performing sends or receives at the time the checkpoint was requested, either that or there is a bug in the timing code. If the problem is the former, then extra yield points may need to be embedded inside the routines, although once a communication has started to be handled, it really needs to be allowed to finish.
- On occasion, there appears to be an unusual issue after a restore has been performed, in that some calls to `malloc()` fail, especially from within `glibc` routines. It is possible that the state of the allocated memory tables is being damaged at some point during the restore and is causing things to fail. The source of this problem is unknown at present, the heap is not touched by the program, so the fact that it can become broken after a restore is odd. The only possibility is that

some heap information is contained on the stack (such as a temporary variable on the stack, being stored on the heap). If this is the case then there is little that can be done, as without compiler assistance scanning the stack for pointers to the heap would be very difficult. The ‘cgs’ example program, is particularly plagued by this bug (probably due to its extensive memory requirements), and as such does not restore from checkpoints properly at present.

- Error handling within FTMPI is currently extremely basic, and non-existent in parts, especially within the MPI interface. Most of the functions do not perform standard checks on things like communicator objects (as only one communicator is valid anyway), and source destinations are currently unchecked. These types of semantic error handling are not entirely necessary within a prototype environment, but would be necessary if the program was to be made available for general use.

Whilst there may be additional bugs within the code, they have not come out during testing and the author is unaware of them.

Bibliography

- [1] I. Foster and C. Kesselman, eds. The Grid: Blueprint for a future computing infrastructure. Morgan Kauffman Publishers, USA, 1999
- [2] Condor Research Project: High Throughput Computing. University of Wisconsin-Madison
URL <http://www.cs.wisc.edu/condor/>
- [3] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230, University of Tennessee, Knoxville, Tennessee, March 1994
URL <http://www.mpi-forum.org/docs/mpi-10.ps>
- [4] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, Tennessee, July 1997
URL <http://www.mpi-forum.org/docs/mpi-20.ps>
- [5] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Technical Report ANL/MCS-P5670296, Mathematics and Computer Science Division, Argonne National Laboratory and the Department of Computer Science & NSF Engineering Research Center for CFS, Mississippi State University, February 1996
URL <ftp://ftp.mcs.anl.gov/pub/mpi/mpich/papers/mpicharticle.ps>
- [6] Ralph Butler, William Gropp, and Ewing Lusk. A Scalable Process-Management Environment for Parallel Programs. In 7th European PVM/MPI User's Group Meeting, pp. 168–175. University of North Florida & Argonne National Laboratory, Balatonfüred, Hungary, September 2000
URL <http://www.mcs.anl.gov/home/rbutler/mpd/mpd.ps>

- [7] G.F.Fagg and J.J.Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In 7th European PVM/MPI User's Group Meeting, pp. 346–353. Balatonfüred, Hungary, September 2000
URL <http://www.netlib.org/utk/people/JackDongarra/PAPERS/ft-mpi.pdf>
- [8] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In The International Parallel Processing Symposium, pp. 526–531. IEEE Computer Society Press, Honolulu, HI, April 1996
URL <http://wwwbode.informatik.tu-muenchen.de/~cocheck/IPPS96.ps.gz>
- [9] Juan Leon, Allan Fisher, and Peter Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, Carnegie Mellon University, February 1993
URL <http://www.cs.cmu.edu/afs/cs/project/cmcl/archive/Nectar-papers/93fspvm.ps>
- [10] J. Casas, D. L. Clark, P. S. Galbiati, R. Konuru, S. W. Otto, R. M. Prouty, and J. Walpole. MIST: PVM with Transparent Migration and Checkpointing. In The 3rd Annual PVM Users' Group Meeting. Pittsburgh, PA, May 1995
URL <ftp://cse.ogi.edu/pub/ogipvm/papers/mist.ps.gz>
- [11] Jeremy Casas, Dan Clark, Ravi Konuru, Steve Otto, Robert Prouty, and Jonathan Walpole. MPVM: A Migration Transparent Version of PVM. Technical Report CSE-95-002, Department of Computer Science and Engineering Oregon Graduate Institute of Science & Technology, February 1995
URL ftp://cse.ogi.edu/pub/ogipvm/papers/mpvm_TR.ps.gz
- [12] K. F. Ssu and W. K. Fuchs. PREACHES: Portable Recovery and Checkpointing in Heterogeneous Systems. In The 28th International Symposium on Fault-Tolerant Computing, pp. 38–47. Munich, Germany, June 1998
URL <http://dynamo.ecn.purdue.edu/~fuchs/fuchs/ftcsKFS98.ps>
- [13] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing. MIT

- Press, Boston, September 1994
URL <http://www.netlib.org/pvm3/book/pvm-book.html>
- [14] W. Gropp and E. Lusk. PVM and MPI Are Completely Different. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, September 1998
URL <http://www-unix.mcs.anl.gov/mpi/papers/archive/misc/pvmmpi-2.ps>
- [15] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. PVM and MPI: A comparison of features, 1996
URL <http://www.epm.ornl.gov/pvm/PVMvsMPI.ps>
- [16] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under Unix. Technical Report UT-CS-94-242, Department of Computer Science, University of Tennessee & Department of Computer Science, Princeton University, 1994
URL <http://www.cs.utk.edu/~plank/plank/papers/CS-94-242.ps.Z>
- [17] David Gibson. Generic User-space Checkpointing. CAP Research, Australian National University, Canberra, October 1999
URL <http://cap.anu.edu.au/~dgibson/esky.html>
- [18] David Gibson. esky: A slightly portable user-space checkpointing system. CAP Research, Australian National University, Canberra, October 1999
URL <http://cap.anu.edu.au/~dgibson/esky.html>
- [19] Victor C. Zandy. ckpt: A process checkpoint library. Computer Sciences Department, University of Wisconsin-Madison, April 2002
URL <http://www.cs.wisc.edu/~zandy/ckpt/README>
- [20] William R. Dieter and James E. Lumpp Jr. User-level Checkpointing of POSIX Threads. Technical Report CEG-99-004, University of Kentucky, June 1999
URL <http://www.dcs.uky.edu/~chkpt/pub/tr-CEG-99-004.ps>
- [21] Alessandro Rubini & Jonathan Corbet. Linux Device Drivers, chapter 13: 'mmap and DMA'. O'Reilly, 2nd edition, June 2001. ISBN 0-59600-008-1
URL <http://www.xml.com/1dd/chapter/book/ch13.html>

- [22] University of Aarhus Alexandru Csete. The Monte Carlo Method to approximate Pi
URL <http://www.daimi.aau.dk/~u951581/pi/MonteCarlo/pimc.html>
- [23] Jonathan Richard Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, August 1994
URL <http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.ps>
- [24] Standard Performance Evaluation Corporation (SPEC). SPEC Benchmark Systems
URL <http://www.specbench.org/>
- [25] Computer Science Department University of Tennessee. Bench Web
URL <http://www.netlib.org/benchweb/>
- [26] Brian W. Kernighan and Dennis M. Ritchie. The C Programming Language. Prentice Hall, 2nd edition, 1988. ISBN 0-13-110362-8
URL <http://cm.bell-labs.com/cm/cs/cbook/>
- [27] Richard Stevens. TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols. Addison-Wesley, 1996. ISBN 0-201-63495-3
URL <http://www.kohala.com/start/tcpipiv3.html>
- [28] Vic Metcalfe. Unix Socket FAQ
URL <http://www.developerweb.net/sock-faq/>
- [29] Jeff Magee and Jeff Kramer. Concurrency: State Models and Java Programs. Wiley, November 1999
URL <http://www.doc.ic.ac.uk/~jnm/book/index.html>
- [30] Perl. O'Reilly site, devoted to the Perl language.
URL <http://www.perl.com/>
- [31] Larry Wall et al. The Perl CD Bookshelf. O'Reilly, 1999