NTNU
Innovation and Creativity

Fault-tolerance for MPI Codes on Computational Clusters

Knut Imar Hagen

Master of Science in Computer Science
Submission date: June 2007
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Many applications, e.g. several seismic processing codes, need to be run over several weeks, if not longer. These applications become a challenge on larger computational clusters, since it is likely a hardware failure will occur during the execution of the program. These codes therefore need to be made fault tolerant.

This thesis' main focus is on how to handle the situation when a computer node in the cluster crashes for MPI (Message Passing Interface) applications.

MPI-2 defines error handling routines, but many lack in current MPI-1 implementations. This thesis hence includes a study of how to implement exception handling for such MPI codes. A given application will be used as a test case.

Assignment given: 20. January 2007
Supervisor: Anne Cathrine Elster, IDI

Abstract

This thesis focuses on fault-tolerance for MPI codes on computational clusters. When an application runs on a very large cluster with thousands of processors, there is likely that a process crashes due to a hardware or software failure. Fault-tolerance is the ability of a system to respond gracefully to an unexpected hardware or software failure.

A test application which is meant to run for several weeks on several nodes is used in this thesis. The application is a seismic MPI application, written in Fortran90. This application was provided by Statoil, who wanted a fault-tolerant implementation. The original test application had no degree of fault-tolerance –if one process or one node crashed, the entire application also crashed.

In this thesis, a collection of fault-tolerant techniques are analysed, including checkpointing, MPI Error handlers, extending MPI, replication, fault detection, atomic clocks and multiple simultaneous failures. Several MPI implementations are described, like MPICH1, MPICH2, LAM/MPI and Open MPI. Next, some fault-tolerant products which are developed at other universities are described, like FT-MPI, FEMPI, MPICH-V including its five protocols, the fault-tolerant functionality of Open MPI, and MPI Error handlers.

A fault-tolerant simulator which simulates the application's behaviour is developed. The simulator uses two fault-tolerance methods: FT-MPI and MPI Error handlers.

Next, our test application is similarly made fault-tolerant with FT-MPI using three proposed approaches: MPI_Reduce(), MPI_Barrier(), and the final and current implementation: MPI Loop. Tests of the MPI Loop implementation are run on a small and a large cluster to verify the fault-tolerant behaviour. The seismic application survives a crash of n-2 nodes/processes. Process number 0 must stay alive since it acts as an I/O server, and there must be at least one process left to compute data.

Processes can also be restarted rather than left out, but the test application needs to be modified to support this.

Acknowledgements

I would like to thank Dr. Anne C. Elster for being my advisor on this Master's thesis, and for giving me valuable input throughout the project.

I would also like to thank all the students in room ITV-458 (where I have been writing my thesis) for responding to some of my questions.

Thanks to Stig-Kyrre Foss at Statoil Research Center Rotvoll for giving us this interesting thesis assignment, and for letting us use their application as a test application in my thesis. Thanks to Jon André Haugen and Børge Arntzen, also from Statoil, for taking their time, explaining the source code of the test application to me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> Fault-tolerance is the ability of a system to respond gracefully to an unexpected hardware or software failure. There are many levels of fault-tolerance, the lowest being the ability to continue operation in the event of a power failure. Many fault-tolerant computer systems mirror all operations – that is, every operation is performed on two or more duplicate systems, so if one fails the other can take over.
>
> - Webopedia on "Fault-tolerance"

High-end high performance computing systems have today thousands of processors. This number is expected to grow for the next years, as processing power of a single processor levels off due to energy limits and quantum effects. The largest number of processors on a cluster today is 131 072, according to the Top 500 [1] list as of November 2006. A critical issue of systems consisting of such large numbers of processors, is the ability of the system to deal with processor and other failures. One may believe that this is a job of the operating system, and that the programmer should not have to worry about it. However, this is not currently the case, hence the motivation for this thesis.

1.1 Message passing and fault-tolerance

In the High Performance Computing (HPC) field, the most common programming paradigm for Computational Clusters is message passing between processes. When one or more of the processes fail, it is favourable that the whole application does not crash. Based on this mentality, there should be a mechanism which prevents this application from crashing. There are several approaches to implement this, but the most common requirements are that a failure can be detected, there is information available that allows the computation to continue, and that the computation can be restarted. One example is to handle error codes from the failed processes and restarting them with the previous known checkpoint as the beginning dataset. A checkpoint is, in brief, a copy

of the so far computed data, mainly saved to the harddrive. However, the basic idea with fault-tolerance is that if one process crashes, the other processes should continue their processing with minor disturbance.

1.2 Thesis goal and motivation

This thesis' main focus will be on how to handle the situation when a computer node in the cluster (or a process) crashes for MPI (Message Passing Interface) applications. MPI-2 defines error handling routines, but many lack in current MPI-1 implementations. MPI does not define fault-tolerance in either version of the standard.

This thesis topic was motivated by a larger seismic simulation done at Statoil Research Center Rotvoll, that runs for several weeks on a large cluster. On their large cluster there may often encounter node failures that interrupts and crashes their simulations that have been running for several weeks. This thesis analyses several fault-tolerant techniques and shows how to implement fault-tolerance for such MPI codes. This thesis covers fault-tolerance for a fairly straightforward case with a collection of smaller independent parts of a dataset. These parts are distributed among processors on a cluster, and parts assigned to failing processes will be re-distributed to the processes which are still alive.

1.3 Outline

- Chapter 1 has been this brief introduction.
- Chapter 2 describes different MPI implementations and related work which includes several fault-tolerant products and techniques.
- Chapter 3 describes the test application in this thesis. It also describes a simulator which simulates the behaviour of the test application. The simulator is made fault-tolerant with two different methods.
- Chapter 4 describes fault-tolerant techniques that are studied and analysed during this thesis work.
- Chapter 5 is results and discussion where the most important part is to make the test application fault tolerant. This chapter also contains several tests.
- Chapter 6 is the conclusion and describes future work.
- Bibliography is a list of citations in this thesis, and a description of them. Links are included where available.

- Appendix A is a source code listing of the simulator in three different versions. The first version simulates the behaviour of the original test application, the second is a fault-tolerant implementation with FT-MPI, and the third is a fault-tolerant implementation of error handlers.

- Appendix B is a source code listing of code snippets from the different fault-tolerant implementations of the test application.

- Appendix C is a listing of the output from the testing of the fault-tolerant test application.

- Appendix D gives a list of descriptions of what was gained from the different articles and web sites that are cited in this thesis.

- Appendix E is an electronic collection of files from this thesis. Description of these files can be found in the sheet called Appendix E and in the README file included in the zip file.

Chapter 2

Background Material

2.1 Computational cluster

A computational cluster is a collection of computer nodes connected in a network. The network connection can be very fast or slow, and the costs of building the network comes thereafter. A cluster may be contained of hundreds or even thousands of nodes. Every node may also contain a large number of processors. The processors may also be multicore, which yields that the total number of processes run simultaneously can be enormous. The top 500 list [1] is a list of the 500 most powerful computer systems in the world. A cluster may be homogeneous or heterogeneous, meaning a set of computers with the exact same hardware, or a set of computers with different hardware.

2.2 Message passing

Message passing for parallel programming is a sort of communication between processes, where the communication consists of processes sending messages to each other. These messages could be a triggering of a function, data packets or signals. Messages may be sent to a process on the same node, which means copying data inside the primary memory. The messages could also be sent to a process on another node, where the messages are sent over the network.

2.3 MPI

Message Passing Interface (MPI) [2, 3] is a library specification for message passing. It is the de facto standard for communication between the processes modeling a parallel program on a distributed memory system. MPI does not define the protocol to be used when passing messages over the network, it is an

application layer interface in the OSI model [4]. MPI's dual goals are high performance (scalability), and high portability. High productivity of the interface, in programmer terms, is not one of the key goals of MPI, and MPI is generally considered to be low-level. It expresses parallelism explicitly, rather than implicitly. MPI is a specification, not an implementation. MPI has Language Independent Specifications (LIS) for the function calls, and language bindings.

Most MPI implementations consist of a set of routines which are to be called from Fortran, C or C++, or by any other language which has an interface with these API's. Portability and speed are the main advantages of MPI compared to older libraries. Portability, because MPI has been implemented for almost every distributed memory architecture. Speed, because each implementation is in principle optimised for the hardware on which it runs. MPI is also supported on shared memory and NUMA [5] architectures.

MPI uses messaging between processes. Each process are assigned an unique rank identification, an integer from zero to the total number of processes minus one. There is both point-to-point and collective communication. Point-to-point communication is when a process sends some data to another process. Collective communication is when groups of processes communicate. An example of collective communication can be to find a maximum value of a specified variable which all processes holds, or to distribute an array among all processes. Collective communication is not very well suited for scalability, because the more processes in the group, the longer the communication takes to complete. A given group of processes is called a communicator, and the communicator used by the processes must be a part of the parameter list of the MPI functions that are either collective or point-to-point. These are specified as intra communicators. Inter communicators are communicators that lets processes in different communicators communicate with each other.

There are two versions of the MPI standard, MPI-1 and MPI-2.

2.3.1 MPI-1

The first version of MPI, 1.0, was released in June 1994. About 60 people from 40 organizations, mainly from the United States and Europe, had been meeting in the Message Passing Interface Forum (MPIF) since January 1993 to discuss and define a set of library interface standards for message passing. MPIF was not supported by any official standards organization. The forum led to a new definition, MPI-1.1 which superseded MPI-1.0. A side effect of MPI-2 standardization (completed in 1996) was clarification of the MPI-1 standard, creating the MPI-1.2 level. About 128 functions comprise the MPI-1.2 standard as it is now defined.

2.3.2 MPI-2

The MPI-2 standard describes additions to the MPI-1 standard and defines MPI-2. These include miscellaneous topics, dynamic process creation and management, one-sided communications, extended collective communications - two groups of processes, external interfaces, scalable file I/O, and additional language bindings. The LIS of MPI-2 specifies over 500 functions and provides language bindings for Fortran90, C and C++.

It is important to note that MPI-1.2 programs, now deemed "legacy MPI-1 programs," still work under the MPI-2 standard although some functions have been deprecated. This is important since many older programs use only the MPI-1 subset.

2.4 MPI Implementations

MPI is, as previous stated, a specification, not an implementation. In this section, some of the known implementations are described.

2.4.1 MPICH

MPICH [6] (also called MPICH1) is a freely available, portable implementation of MPI, developed at Argonne National Laboratory. It implements the MPI-1 standard. The current version of MPICH is 1.2.7p1 and was released on November 4th, 2005.

MPICH2 is, like MPICH1, a freely available, portable implementation of MPI, developed at Argonne National Laboratory. It implements the MPI-2 standard. The goals of MPICH2 are to provide an MPI implementation for important platforms, including clusters, SMPs, and massively parallel processors. It also provides a vehicle for MPI implementation research and for developing new and better parallel programming environments. The current version of MPICH2 is 1.0.5, released on December 13, 2006. MPICH2 replaces MPICH1 and should be used instead of MPICH1 except for the case of clusters with heterogeneous data representations (e.g., different lengths for integers or different byte ordering). MPICH2 does not yet support those systems (support is planned for 2007).

2.4.2 LAM/MPI

LAM/MPI [7] (Local Area Multicomputer) is an MPI programming environment and development system for heterogeneous computers on a network.

LAM has a full implementation of MPI-1 and much of MPI-2, and offers extensive monitoring capabilities to support debugging.

At time being, LAM/MPI is in a maintenance mode. Bug fixes and critical patches are still being applied, but little real "new" work is happening in LAM/MPI. This is a direct result of the LAM/MPI Team spending the vast majority of their time working on their next-generation MPI implementation - Open MPI. The team actually encourages to try migrating from LAM/MPI to Open MPI, because implementations should be converted without any problems. Open MPI contains many features and performance enhancements that are not available in LAM/MPI [8].

2.4.3 Open MPI

Open MPI [9] is a project combining technologies and resources from several other projects (FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI). Its goal is to build the best MPI library available. Open MPI is an implementation based on the MPI-2 standard. Open MPI provides a unique combination of novel features previously unavailable in an open-source, production-quality implementation of MPI. It has a full MPI-2 standards conformance.

The organizations (and newly-combined projects) contributing to Open MPI are Indiana University (LAM/MPI), the University of Tennessee (FT-MPI), and Los Alamos National Laboratory (LA-MPI). Additional collaborators are at Sandia National Laboratories and the High Performance Computing Center at Stuttgart (PACX-MPI).

2.5 Related work

2.5.1 FT-MPI

FT-MPI [10] is an abbreviation for Fault-Tolerant Message Passing Interface. It is developed at the ICL group at the University of Tennessee. There they have a project called HARNESS (Heterogeneous Adaptive Reconfigurable NEtworked SystemS) [11]. HARNESS is an experimental Metacomputing System aiming at providing a highly dynamic, fault-tolerant computing environment for high performance computing applications. As the HARNESS system itself is both dynamic and fault-tolerant (no single points of failure), it became possible to build a MPI plug-in with added capabilities such as dynamic process management and fault-tolerance. This plug-in is called FT-MPI.

FT-MPI is an independent implementation of the MPI 1.2 message passing standard that has been built from the ground up with both user and system

level fault-tolerance. FT-MPI allows developers to build fault-tolerant or survivable applications that do not immediately exit due to the failure of a processor, node, or MPI task. A number of failure modes are offered that allow a range of recovery schemes to be used that closely match different classes of parallel applications. FT-MPI is unique because it avoids restarting surviving nodes, which can be a considerable advantage on very large scale systems where rescheduling and restarting of the entire application is the only current option.

FT-MPI survives the crash of n-1 processes in a n-process job, and, if required, can respawn/restart them. However, it is still the responsibility of the application to recover the data-structures and the data on the crashed processes. The application discovers any errors from the return code of any MPI call, with a new fault indicated by MPI_ERR_OTHER. When there is a failure within a communicator, the communicator is marked as having an error. Once a communicator has an error state it can only recover by rebuilding it, using a modified version of one of the MPI communicator build functions such as MPI_Comm_create(), MPI_Comm_split() or MPI_Comm_dup(). The system behaves according to the given communicator modes and communication modes. These can be specified while starting the application. The four communicator modes are described next.

- SHRINK: The communicator is shrunk so that its data structures contains no holes. This means that a node has been completely left out, and this forces all the processes to change their ranks with a recall to MPI_Comm_rank().

- BLANK: This is the same as SHRINK, but this one, as its name implies, will leave blanks or gaps in the communicator where there used to be a process. These gaps can be filled in later. This means that all the processes retain their respective ranks. If a process communicates with a gap process, it will get an invalid rank error. The function MPI_Comm_size() will return the size of the communicator, including the gap processes.

- REBUILD: This is the most complex mode, as it forces the creation of new processes to fill any gaps. The new processes can either be placed in the empty gaps with their ranks, or the communicator can be shrunk, so the new processes are added at last with the highest ranks. This is used for applications that require a certain number of processes to execute.

- ABORT: This mode actually forces a graceful abort of the application if an error is detected. This cannot be avoided in the application, so the only option is to use of the three above communicator modes.

Next are the two communication modes described.

- NOP: No operations on error. This means that no user level message operations are allowed, and these will simply return an error code. This is used to allow an application to return from any point in the code to a state where it can take appropriate action as soon as possible.

- CONT: All communication that is not to the effected or failed node can continue as normal. Attempts to communicate with a failed node will return errors until the communicator state is reset.

2.5.2 FEMPI

FEMPI is an abbreviation for Fault-tolerant Embedded Message Passing Interface [12]. The project is in development at the High-performance Computing and Simulation (HCS) Research Laboratory at the University of Florida. FEMPI is a lightweight, fault-tolerant design and implementation of the common MPI standard. Fault-tolerance and recovery is provided through three stages including detection of a fault, notification of the fault, and recovery from the fault. FEMPI prevents the entire application from crashing on individual process failures. On a failure, MPI Restore, a component of FEMPI, informs all the MPI processes regarding the failure. The status of senders and receivers are checked before communication to avoid attempts to communicate with failed processes. If the communication partner (sender or receiver) fails after the status check and before communication, then a timeout-based recovery is used to recover out of the MPI function call. FEMPI, just like FT-MPI, survives the crash of n-1 processes in an n-process job, and, if required, can re-spawn/restart them. However, it is still the responsibility of the High-Availability Middleware to recover the data structures and the data on the crashed processes. A program written in conventional MPI can execute over FEMPI with little or no alteration.

FT-MPI, described in the previous section, attempts to provide fault-tolerance in MPI by extending the MPI process states and communicator states from the simple valid, invalid as specified by the standard to a range of states. The range of communicator states specified by FT-MPI helps the application with the ability to decide how to alter the communicator, its state and the behaviour of the communication between intermediate states on occurrence of a failure. FT-MPI provides for graceful degradation of applications, but has no support for transparent recovery from faults. The design concept of FEMPI is also largely based upon FT-MPI, although there are significant differences as FEMPI is designed to target embedded, resource limited, and mission-critical systems where faults are more commonplace, such as payload processing in space.

2.5.3 MPICH-V

MPICH-V [13, 14] is a research effort with theoretical studies, experimental evaluations and pragmatic implementations aiming to provide a MPI implementation based on MPICH (described in Section 2.4.1), featuring multiple fault-tolerant protocols. It is an automatic fault-tolerant library, which means that a totally unchanged application linked with the MPICH-V library is a

fault-tolerant application. The project is in development at the Laboratoire de Recherche en Informatique at Université de Paris Sud.

The MPICH-V project contains for the time being five versions with some dissimilarities. The first three are deprecated, but they are working on a new implementation of all these protocols inside a generic framework, the VCL version.

V1

The first implementation, called MPICH-V1 ch_xw, was released March 10 2003. It features a fault-tolerant protocol designed for very large scale computing using heterogeneous networks. Its fault-tolerant protocol is well suited for Desktop Grids and Global computing as it can support a very high rate of faults, but requires a larger bandwidth for stable components to reach good performance.

V2

The second implementation, called MPICH-V2 ch_v2, was released first on June 23 2003. It features a fault-tolerant protocol designed for homogeneous network large scale computing (typically large clusters). Unlike MPICH-V1, it only requires a very small number of stable components to reach good performance on a cluster. Its uncoordinated checkpoint protocol makes it suitable for large scale applications, where the large number of nodes induces a low mean time between failures.

VCausal

The next implementation, called MPICH-VCausal ch_cl, was released first on December 19 2003. It features a fault-tolerant protocol designed for low latency dependent applications which must be resilient to a high fault frequency. It combines the advantages of the other message logging protocols (thus providing computation progress even with high fault frequency) with direct communication and absence of acknowledgements (thus avoiding high latency impact).

VCL

The newest implementation for MPICH-1 is MPICH-VCL, called ch_v and was released on January 25 2006. This version features a fault-tolerant protocol designed for extra low latency dependent applications. The Chandy Lamport

algorithm [15] used in MPICH-VCL does not introduce any overhead during fault free execution. However, it requires restarting all nodes (even non crashed ones) in the case of a single fault. As a consequence, it is less fault resilient than message logging protocols, and is only suited for medium scale clusters. The intention is to implement all the protocols from the earlier versions in this ch_v implementation.

PCL

Another implementation was released May 01 2007. This implementation is MPICH-PCL, called pcl and features a Blocking Chandy Lamport fault-tolerant protocol in the MPICH-2 implementation. This consists of a new channel, called ft-sock, based on the TCP sock channel, and two components, a checkpoint server and a specific dispatcher, supporting large scale and heterogeneous applications. A migration capability is also developed. Computation is now able to restart from a given checkpoint wave.

2.5.4 Open MPI

Open MPI is described in general in Section 2.4.3, and now the fault-tolerant functionality [16] will be described.

Currently, the fault-tolerant functionality has not been implemented. It is being developed in separate non-public branches because it is still experimental and not yet considered stable. Researchers also need to preserve the ability to publish on the new ideas and techniques that are being developed. The following fault-tolerant techniques are planned to be supported in Open MPI:

- Coordinated and uncoordinated process checkpoint and restart. Similar to those implemented in LAM/MPI and MPICH-V, respectively.
- Message logging techniques. Similar to those implemented in MPICH-V.
- Data Reliability and network fault-tolerance. Similar to those implemented in LA-MPI.
- User directed, and communicator driven fault-tolerance. Similar to those implemented in FT-MPI.

Network failover and data reliability is scheduled to be released in version 1.2. Subversions of 1.2 has recently been released, but they still lack implementations of fault-tolerance. Rollback recovery with checkpoint/restart is scheduled to be released in version 1.3.

2.5.5 MPI Error handlers

William Gropp and Ewing Lusk [17] describes how to make a non-transparent fault-tolerant MPI application. They claim that fault-tolerance is a property of a program, not of an API specification or an implementation. There are several ways to approach fault-tolerance:

- Using checkpointing and restarting the application manually after an application crash. There are two ways to do this, and one is at the user level, and the second at the system level. The user level is the easiest as you in theory only save the computed data to disk, and recover it after a crash. There must be taken in consideration how often a checkpointing operation should be done, because it is a performance killer.

- Using a communicator is a fundamental concept in MPI. It is a distributed object that supports both collective and point-to-point communication. A failure of one process affects all the other processes in the communicator. This means that using only one communicator is a fragile case. When building a manager/worker application, inter communicators can be used. There the manager and a worker are assigned a communicator, and the manager and another worker are assigned another communicator. No collective operations are needed and the workers do not communicate with each other. If one worker dies, work can be assigned to another worker which has nothing to do at the moment. If MPI-2 is used, the function MPI_Comm_spawn() can be used to create the inter communicators, and when a worker dies, the function can be used to replace the dead worker.

- Modifying the semantics of certain MPI objects to be slightly different from those described in the MPI Standard. For example, a communicator can enter a state in which some ranks are defined and not others. In a standard MPI implementation a process's rank cannot change. They believe this is not the way to go in writing production applications, but rather use a more limited and consistent approach with existing programming style (described next).

- Rather than modifying existing semantics, extensions can be added to MPI that have semantics that support the writing of fault-tolerant programs. These should also be consistent with all existing MPI semantics. New functions could be added and co-exist with standard MPI functions. A central idea is the Process array object, which plays the role of a communicator, but differs in several ways. It is in some ways the same as the BLANK option in FT-MPI, which mark dead nodes, and leaves a gap in the communicator, which does not change the respective ranks of the processes.

2.5.6 Fault-tolerant matrix operation techniques

The paper of Anne C. Elster, M. Ümit Uyar and Anthony P. Reeves [18] is about algorithmic fault-tolerant matrix operation techniques. These have been especially tailored for efficient multi-processing on hypercubes. They present an optimal re-distribution of the data matrix upon a single processor fault for matrix-vector multiplication. Algorithmic fault-tolerance becomes a challenge, because of the switching topology, making use of interconnection links in all hypercube dimensions, where the interconnection involves two orthogonal sets of binary trees.

Low communication overhead is maintained, by focusing on re-distributing the load for each processor to minimize the effect on the remaining partial orthogonal trees. Applying the techniques discussed to any row-dependent grid problem which needs a re-distribution of data upon processor failure/failures can easily be done. The model they use assumes fault detection where faults are considered to be permanent faults in one or more of the processors.

Chapter 3

Test application

3.1 Description of test application

The test application to which fault-tolerance is to be implemented, is a MPI application which uses a seismic dataset captured from sonars on the sea level. The goal is to create an image of the sea ground and from this predict where to find oil. The application processes the information from sensors with a distribution of the dataset to several processes. The dataset contains several shots, and the processes pick a shot and process it, and then pick another. The processes can continue to pick shots for weeks, depending on how many shots there are in the dataset. There is no communication between the processes during the computation of one shot, only before and after the computation. When all the processes have finished computing their amount of shots, the results are gathered with the collective communication call MPI_Reduce() or gathered into a file. An example of the output is shown in Figure 3.1 and zoomed in at a few of the shots viewed in Figure 3.2. If one or more of the processes have failed during the computation time, the whole application crashes.

The test application is written in Fortran90, and is built with several libraries to make the code more usable and divided. One of the libraries is named FIO, and is an abbreviation of Fortran Input Output. In the FIO library there is a function called FioPfm(), and the application lets rank 0 call this function. This is a forever loop calling the MPI_RECV() subroutine, and based upon the messages received from the other processes, rank 0 acts as an I/O server for them. This means that rank 0 is a single point of failure, because the other processes may crash, and they can be left out of the communicator, but not rank 0. The other processes can be left out because the shots that have not been processed can be run again when everything else is finished.

If the failed processes were detected before the collective call MPI_Reduce(), the application crash could be prevented.

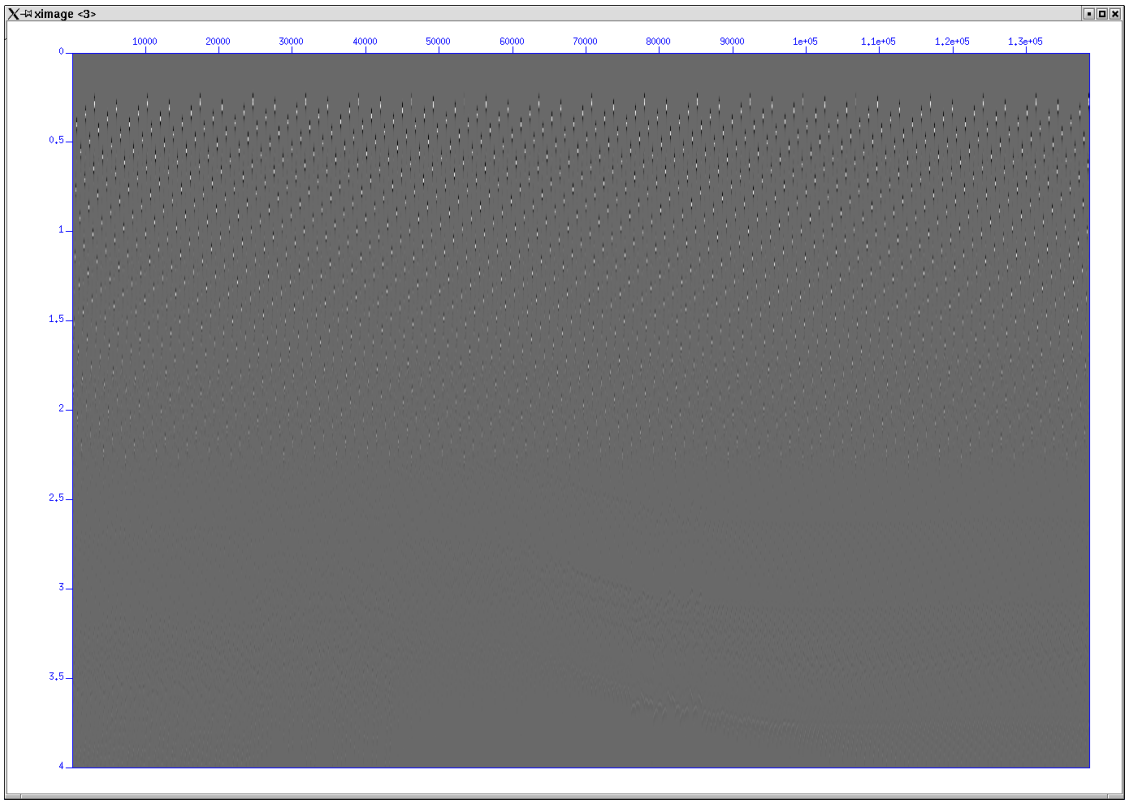

Figure 3.1: Illustration of an example of the output from the test application. The image shows the modeling of all the shots that are processed.

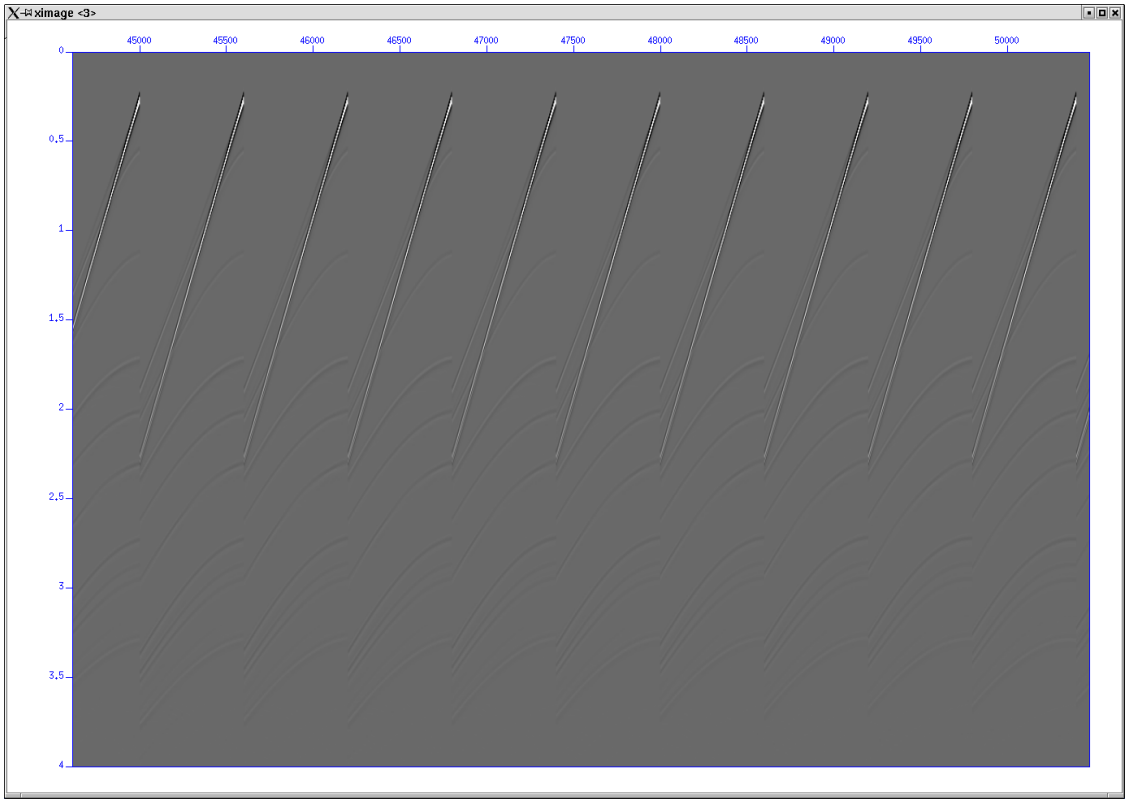

Figure 3.2: Illustration of an example zoom of the output from the test application. The image shows a few shots, a zoom of the whole modeling.

3.2 Simulator

The test application is a heavy code with a huge dataset, and it is meant to run for several weeks. Because of these statements, it is better to make a simulator which has a smaller dataset, spreads the dataset among several processes and computes for a certain amount of time. In the end the MPI_Reduce() will be called to gather results. During the computation time, some of the processes will be forced to fail. This means to exit before MPI_Finalize() has been called.

The original version of the simulator will not notice the crashed processes before MPI_Reduce() is called in the end, and the simulator will not be able to complete the MPI_Reduce() operation. This simulates the current behaviour of the test application. Other versions of the simulator is described in the following subsections. They have fault-tolerance implemented, and either restart or leave out the processes which have crashed before MPI_Reduce() is called. The fault-tolerance code can then be adapted to the test application.

3.2.1 Implementation specification

The computational idea of the simulator is to have a large dataset as a matrix, which is initialised as submatrices on each of the processes. There is a fixed value of the total matrix size defined in the code, and this size is divided among the submatrices. The computation is a rearranging operation on the submatrices that will be performed N times in a for-loop. This means that the first element in the submatrix is swapped with the last element, and the second element swapped with the second last, and so on. In the end, the operation will be reversed, so the matrix is the same as the original after each computation round. The number N can be adjusted to make the computation use more or less time, meaning more or less rounds of matrix rearranging. The simulator knows how many, but not which of the processes that will fail during the computation. The selection of the failing processes is randomised, and they will fail in a sequentially order during the computation. When the computation is finished, the first element in the submatrix is included in the MPI_Reduce() SUM operation, and the result is printed to stdout. This result is not shown in the original version if the simulator fails some of the processes, but the other versions make the simulator print the result and exit gracefully. The behaviour of the original simulator is illustrated in Figure 3.3.

To start the simulator, the following command line is used:

```
$ mpirun -np #procs simulator_original #procfail #loops
```

Where #procs is the number of processes to be spawned, simulator_original is the application name, #procfail is the number of random processes which will fail during the computation, and #loops is the number of times the rearranging operation is to be run on the submatrices. The original implementation can be found in Appendix A.1.

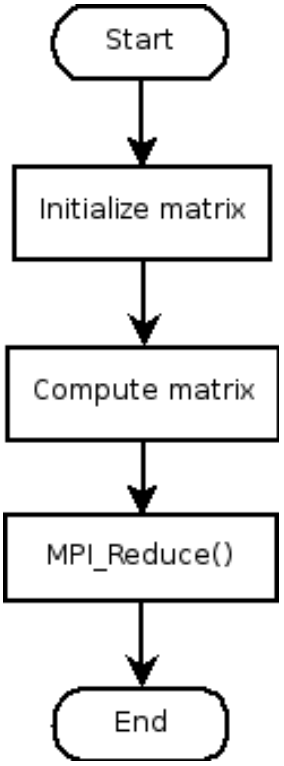

Figure 3.3: Illustration of the original version of the simulator. The submatrices of the processes are initialised. Next some computation is performed on the submatrices. During this computation, a given number of random processes fail. MPI_Reduce() is called in the end, and the simulator will crash if some processes have failed.

3.2.2 Implementation of FT-MPI

The FT-MPI version of the simulator uses the FT-MPI system, which means that FT-MPI must be installed on the computational cluster. FT-MPI is described in Section 2.5.1. In detail, after the processes have completed their computation of their submatrices, there is a MPI_Barrier() performed in a do-while loop which checks if there are failed processes. If there are, a new communicator will be created, and those processes which failed will be restarted. If some of these restarted processes fail, the do-while loop with MPI_Barrier() will once again rebuild the communicator and the failed processes will be restarted. If MPI_Barrier() does not return an error, all the processes may continue to the MPI_Reduce() operation, and the simulator quits gracefully. The behaviour of the simulator with the FT-MPI implementation is illustrated in Figure 3.4.

To run the simulator with FT-MPI, the following command line is used:

```
$ ftmpirun -np #procs -o -s simulator_ftmpi #procfail #loops
```

Where #procs is the number of processes to be spawned, simulator_ftmpi is the application name, #procfail is the number of random processes which will fail during the computation, and #loops is the number of times the rearranging operation is to be run on the submatrices. The FT-MPI implementation can be found in Appendix A.2.

3.2.3 Implementation of Error handlers

The meaning of error handlers is that a program can rescue itself if there occurs errors during the running of a MPI program. It is possible to use error classes and with this specify which error has occurred. An example is when process 0 is receiving data from all the other processes with MPI_Recv(), and they send with MPI_Send(). One error from the MPI operations can be the error class MPI_ERR_RANK which tells that the specified rank is not defined.

When a process exits unexpectedly, the communicator does not know that the process has crashed. A way to test if a process has crashed is let rank 0 call the MPI_Iprobe() routine and check if flag is true and the return value equals MPI_SUCCESS. If not, it may assume that the process has crashed. The reduce operation can be replaced by a manual reduce which adds the same number from all the alive processes, and lets rank 0 print the sum. This can be done with intercommunicators, which means that rank 0 has a communicator for each other process, and the error handler MPI_ERRORS_RETURN is set for each intercommunicator. If an error value is returned from the MPI operations, the intercommunicator is marked as dead by freeing it. Then the process may exit gracefully with MPI_Finalize(). What happens in the simulator is that the processes exit unexpectedly, and never call MPI_Finalize(). When dealing with process crashes, a new approach is required when error handlers are in the picture.

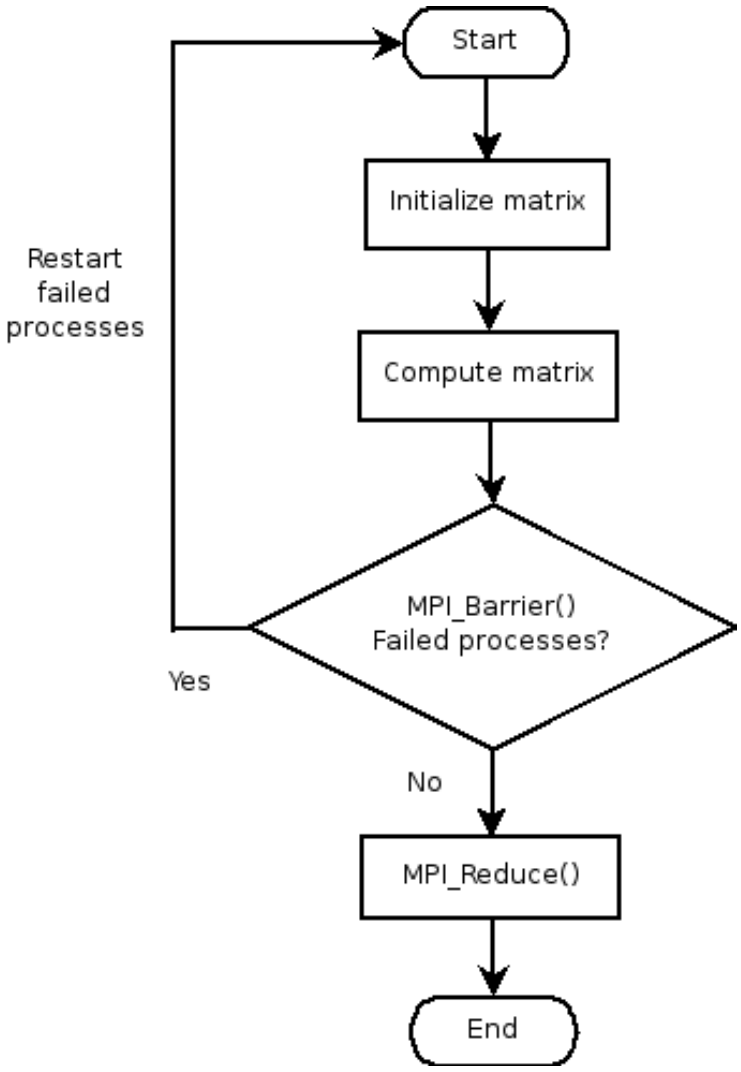

Figure 3.4: Illustration of the FT-MPI version of the simulator. The submatrices of the processes are initialised. Next some computation is performed on the submatrices. During this computation, a given number of random processes fail. When the computation is finished, MPI_Barrier() is called in a loop and failed processes are restarted. MPI_Reduce() is called in the end, and the simulator will print the result on rank 0 and exit gracefully.

A new approach is to run one instance of a master process. This master process uses the MPI-2 defined MPI_UNIVERSE_SIZE which tells how many nodes are available. Then the master process spawns slave processes with the universe size as the maximum count. Individual communicators are made for every slave process. Now, if one process dies, the communicator ceases to exist, which means that the application now has one slave process less, but the application still continues. This simulator version must be run with an implementation of the MPI-2 standard, and LAM/MPI (described in Section 2.4.2) suits well, because one is allowed to use a special application schema when spawning processes. With this schema, the programmer can decide how many processes to be spawned through the creation of a temporary application schema. The simulator also needed some major modifications to be converted to a master/slave application. An illustration of the master/slave-implementation can be found in Figure 3.5.

To run the master/slave-simulator, the following command line is used:

```
$ mpirun n0 ./simulator_errhandler_master #procfail #loops
```

Where #procs is the number of processes to be spawned, simulator_errhandler_master is the application name, #procfail is the number of random processes which will fail during the computation, and #loops is the number of times the rearranging operation is to be run on the submatrices. Both the simulator_errhandler_master and the simulator_errhandler_slave need to be compiled and run with LAM/MPI. Use the "-pthread" compiler flag when compiling the programs. The master/slave-implementation can be found in Appendix A.3.

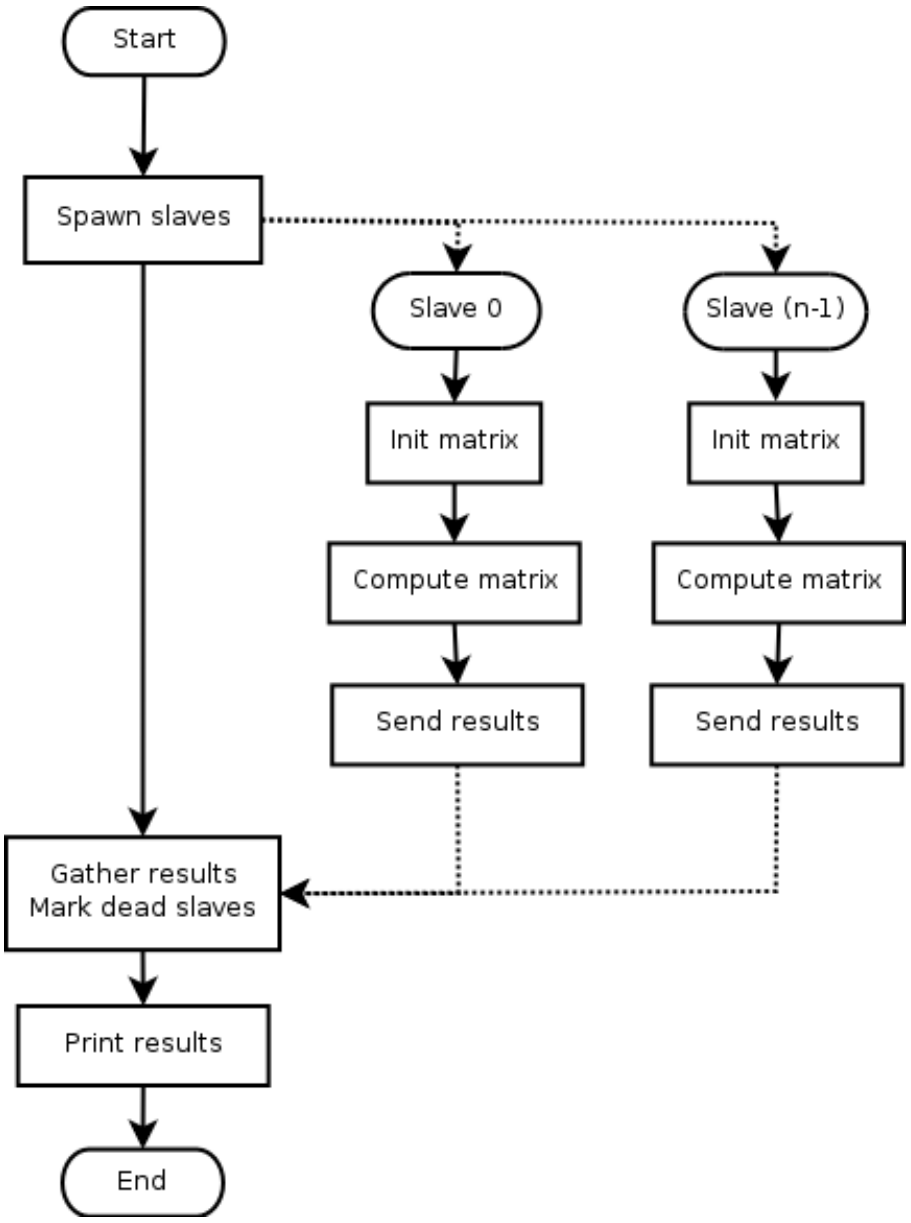

Figure 3.5: Illustration of the error handler version of the simulator. The master process is spawning slave processes which are given their own unique id. The submatrices of the processes are initialised based on this id. Next, some computation is performed on the submatrices. During this computation, a given number of random slave processes fail. When the computation is finished, the slave processes send their value to the master process. If the master process discovers that a slave process has failed, the intercommunicator is freed and marked as dead. The master process gathers the results from the slaves, prints the result and exits gracefully.

Chapter 4

Fault-tolerant techniques for MPI

In this chapter there is a collection and description of fault-tolerant techniques for MPI that are studied or created during this thesis work.

4.1 Checkpointing

Checkpointing is to enable states in an application from where it can be restarted. An example is in a matrix-matrix multiplication. The original matrices and the so far computed matrix can be saved in a file with a specific checkpoint identification. If a process crashes, it can be restarted and identify this last known checkpoint, which can be loaded into memory, and the multiplication can continue.

There are several ways to use checkpointing, which includes transparent checkpointing and manual checkpointing. Another is if the application is designed to pick a little amount of processing work and save this to a file. This is how the test application in this thesis works. Manual checkpointing should not be hard to implement, if the data structures are optimised to be saved to a file. Transparent checkpointing is a bit harder, because a middleware needs to analyse state variables and the data structures which are used in the program, and make automatic checkpoints which automatically should be loaded when a process is restarted.

If the program is parallel, computation in the program may be dependent on the rank number of the process. If several MPI operations are called more or less constantly, maybe the best solution will be to stop computation and message passing, take a checkpoint of every process, and continue. If one process crashes, the application can be restarted on every node and use the last known checkpoint. Another way is to log the MPI messages, and replay those who are sent to the failed process or processes. The checkpoints may also be stored at a checkpoint server which handles everything, and keeps track of which checkpoint to send to which process after a restart.

4.2 MPI Error handlers

The simulator is implemented with error handlers as a master/slave application. The prebuilt MPI_ERRORS_RETURN is used, so that error codes are returned after something has failed, unlike the default MPI_ERRORS_ARE_FATAL which just aborts every process upon a single process failure. An interesting approach is that an application can use a special constructed error handler which suits the problem, and maybe also rescue data structures. To use the matrix-matrix multiplication example again, a special constructed error handler can be made which saves the so far computed matrix to the disk if a process fails, and then the application can abort. When the application is started again, the matrix multiplication may continue from where it failed by reading what was flushed to disk. Such an approach induces that the failure was not a physical failure, like power loss, etc, but for example an error with the communicator.

The method with error handlers used in the simulator implementation, a master/slave implementation, shows that an application can be made fault-tolerant without introducing special frameworks built for fault-tolerance. A normal MPI implementation is then well suited for the problem. Anyway, if the master process dies, it is very hard, if not impossible to keep the application alive.

4.3 Extending MPI

The MPI specification can be extended, which means extending it with new functions or modifying the semantics of the existing functions. The purpose is to enable fault-tolerance, and let the programmer deal with it. New functions may be added, but these should also be consistent with existing MPI semantics. An idea here is for example a new implementation of MPI_Send(). If the function is not able to send a message to another process, and the fault is not caused by known errors, the function can notify every other process that the particular process is dead. Together they may make a new communicator and leave out this dead process.

A similar approach is implemented in FT-MPI, but here everything is known in states in the communicator, as the states are extended from the simple valid/invalid to a range of states. The most important is that if an MPI operation is called, the new implemented error code MPI_ERR_OTHER is returned. Every process that calls an MPI function will have returned this error code, and they know that a process has failed. Then they collectively may rebuild the communicator.

4.4 Replication

Replication can be used to gain a valuable level of fault-tolerance. This can be done with spawning two processes that do exactly the same computation. If one of these processes fail, the results from the other one can be sent to a master process which gathers all the data. Again, this introduces similarities with the master/slave paradigm where the master is a single point of failure.

The pair of slave processes can be run on different nodes, and these nodes may be in separate rooms, connected to separate switches and separate power supplies, etc. The master process should run on a "reliable" node.

A little less conventional, but a lot more easy, is to have two similar clusters where the same application is run on both of them simultaneously. This introduces more aspects like costs, space and most likely more manual interaction.

4.5 Fault detection

A very important matter concerning fault-tolerance is the ability to detect a failure. If the failure was not detected, the entire concept of fault-tolerance would vanish, because a system needs to know if a fault has occurred to respond gracefully to it. An application could have a dedicated thread for every MPI process which runs a function. This function sends a MPI message in a ring, where the ring is every process dedicated to the particular application. See Figure 4.1 for a simple illustration. If one process detects that the send operation to the next process in the ring never completes, then it may begin to assume that a failure has occurred. An approach here is to wait for a given amount of time, and then cancel the send operation, then restart it to see if it then completes. If the same error once again occurs, the process may assume that the next process in the ring has failed. The next step is to leave the failed process out of the communicator, and the process marking the failed process may continue the ring message to the second next process in the ring.

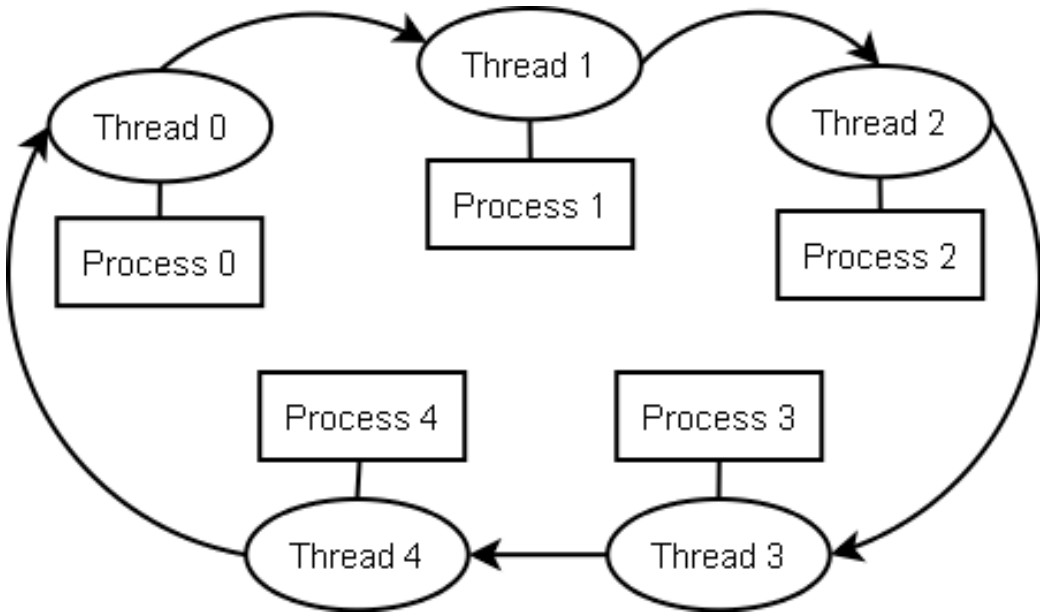

Figure 4.1: Illustration of the ring message fault detection. Processes have a spawned thread which sends MPI messages in a ring of dedicated processes of an application.

4.6 Atomic clocks

An interesting case is if every processor on a cluster were run with the exact same frequency, and the cycles were hit at the exact same time. This would have resulted in what is called atomic clocks on the processors. If an application is written in a way that every process will start at the exact same time and have exactly the same amount of work to do, then they will also finish simultaneously. This involves having an operating system that not lets other processes run on the processors during the computation. If a failure detection mechanism were used, which checks at a CPU cycle interval the instructions used on the processor, it could detect if a process were out of order, and failed in some way. An illustration can be found in Figure 4.2. What then is achieved is instant failure detection.

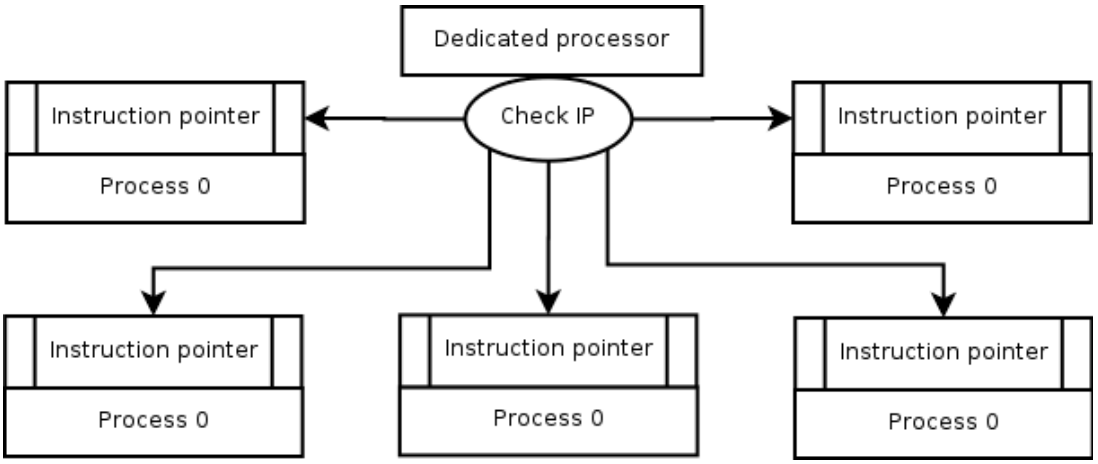

Figure 4.2: Illustration of the atomic fault detection. Every processor run with atomic clocks, and the dedicated processor's task is to check the instructions at every processor's instruction pointer (IP) for equality.

4.7 Multiple simultaneous failures

If one process fails, this could be a somewhat simple task to deal with. If several processes fail at the exact same time, there is a good chance that the techniques developed for a single process failure is not suitable. The failures could be detected with the atomic clock mechanism described in the previous section. However, the detection mechanism would have a problem in knowing which of the processors having the correct instruction, if half or more of the processes fail.

To approach a fault-tolerant technique, supporting multiple simultaneous failures, it can be elaborated with the usage of a message logging technique. Let us say that a Jacobi iteration is to be performed on a distributed matrix. Jacobi iteration is a series of computation rounds, and it computes, in each round, every cell in a matrix with the usage of the cell in the north, west, east and south. The formula of this computation is 4.1.

$$u_{i,j}^{n+1} = \frac{1}{4}(u_{i-1,j}^{n} + u_{i+1,j}^{n} + u_{i,j-1}^{n} + u_{i,j+1}^{n}), i = 1...m, j = 1...m \qquad (4.1)$$

When the cells at the rows and columns at the border of the processes' submatrices are being updated, they miss an element to compute the given cell. Either the north, west, east or the south cell. This cell is hosted on another process' submatrix. Because of this, MPI messages must be sent between these processes. These messages could be logged at a reliable server, or a cluster of these. If one failed process were detected, all the other processes could be paused, and the process could be restarted and do all the computation again with the replay of the messages from the reliable logging server. This actually supports multiple simultaneous failures, because all the other processes which have not failed, will be paused, and the messages which were sent before the failure are replayed. If all these messages were sent when the processes were restarted, the processes would not have to wait for the messages to arrive, just picking them from the memory.

Chapter 5

Results and discussion

5.1 Implementation of FT-MPI in Test application

This section is about making the test application from Statoil fault-tolerant. Although the simulator is in principle behaving the same as the test application, the simulator is way more simplified, letting out many of the problems in implementing fault-tolerance. The simulator makes all the nodes do the same on each computation round, and collective communications can run just fine. With the test application, this is not the case. The test application is also written in Fortran90, making the transition from the simulator a lot more difficult. The fault-tolerant method chosen to be implemented is FT-MPI. This is because FT-MPI is designed to handle failed processes in a neath way with marking communicators as failed, and rebuild them with a collective operation. With the usage of FT-MPI, the need of recoding the entire test application to become a master/slave application dissipates, and one can focus on the fault-tolerant part. Next, three approaches of fault-tolerant implementation are described.

5.1.1 MPI_Reduce()

Since rank 0 will act as an I/O server, doing collective communication is a bit harder. Let us take MPI_Reduce() as an example; all the processes must send a message to the I/O server to tell that they intend to call the collective function. The I/O server has to count all processes which have sent this message, and when this counted number is equal to the communicator size, the I/O server can enter MPI_Reduce() like all the other processes have done. If one process has failed before it sends the message to the I/O server, the server will wait forever, because the message will never arrive. However, if the communication was successful, the server will send acknowledgement messages to all the processes in the communicator after the MPI_Reduce() operation. One way to approach this problem in a fault-tolerant perspective is to observe whether the I/O server have received one message to be doing the collective communication, and instead of calling MPI_Recv(), it can begin to call MPI_Iprobe()

several times. If a new message does not arrive after some given time, then the I/O server can just go ahead and call the collective communication function. This will return the error MPI_ERR_OTHER on all the processes, and a new communicator can be built, so the processes which are alive may continue. The I/O server can now just send acknowledgement packages to all the processes in the communicator, because it has gotten a new communicator size, and the SHRINK option in FT-MPI has been used. Figure 5.1 illustrates this behaviour. Code snippet for this implementation can be found in Appendix B.1.

5.1.2 MPI_Barrier()

The method described above can also be used a bit different, and adapt the procedure from the fault-tolerant simulator. For every shot which is processed, MPI_Barrier() can be called to see if all processes are still alive. If they are not, the I/O server will never get the messages from these failed processes which are meant to notice the server that a MPI_Barrier() will be called. The I/O server must wait a given amount of time, and then just skip to the collective call, in the belief that there are failed processes. If the processes that are "marked as failed" just use more time than the other processes, and more time than the amount of time that the I/O server waits, these "failed processes" will eventually call MPI_Send(). Now, the I/O server is not performing a MPI_Recv(), it is doing a MPI_Barrier(), so this will lead to a deadlock. This is also the case for the MPI_Reduce() method above. Another problem with this approach is that the number of shots are seldom even divided between the processes. This means that in the end, one or more processes may have finished their amount of shots, but the rest are still computing their last shot. The processes which are finished will call MPI_Barrier() one time less, and the application will hang on that function forever. Figure 5.2 illustrates this behaviour. Code snippet for this implementation can be found in Appendix B.2. This implementation works both when the application uses the MPI_Reduce() call in the end, and just gathering the modeling of the shots into a file, which means that the Reduce call is not used.

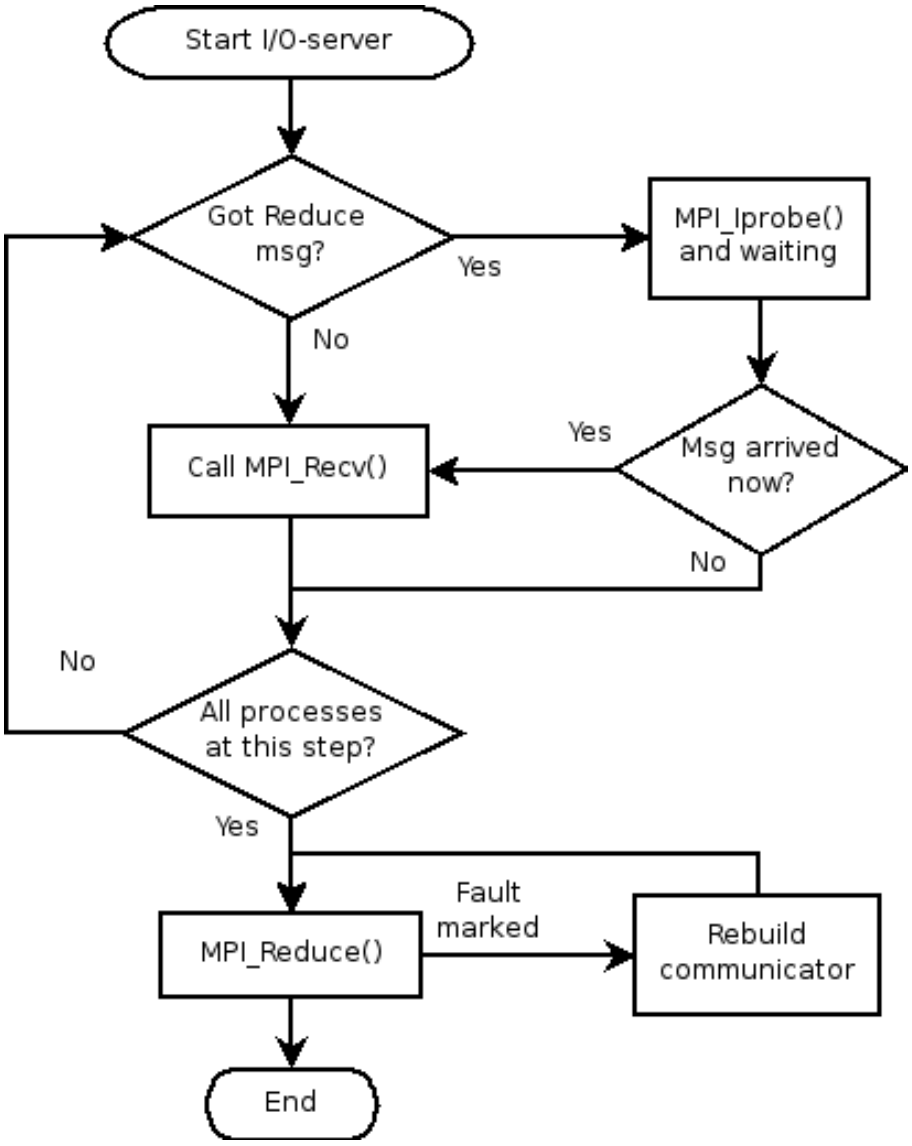

Figure 5.1: Illustration of the I/O server when getting reduce messages from the clients. The server listens for messages with MPI_Recv(), but after the first reduce message, it enters a state calling MPI_Iprobe() and actual waiting for a given amount of seconds. If a message has arrived after some waiting, it calls MPI_Recv(), otherwise it assumes that there are failed processes, and marking a fault. If all processes has sent the reduce message, or a fault is marked, the MPI_Reduce() operation can be performed by all processes. If there is a communicator fault, the communicator is rebuilt, and the MPI_Reduce() is performed again.

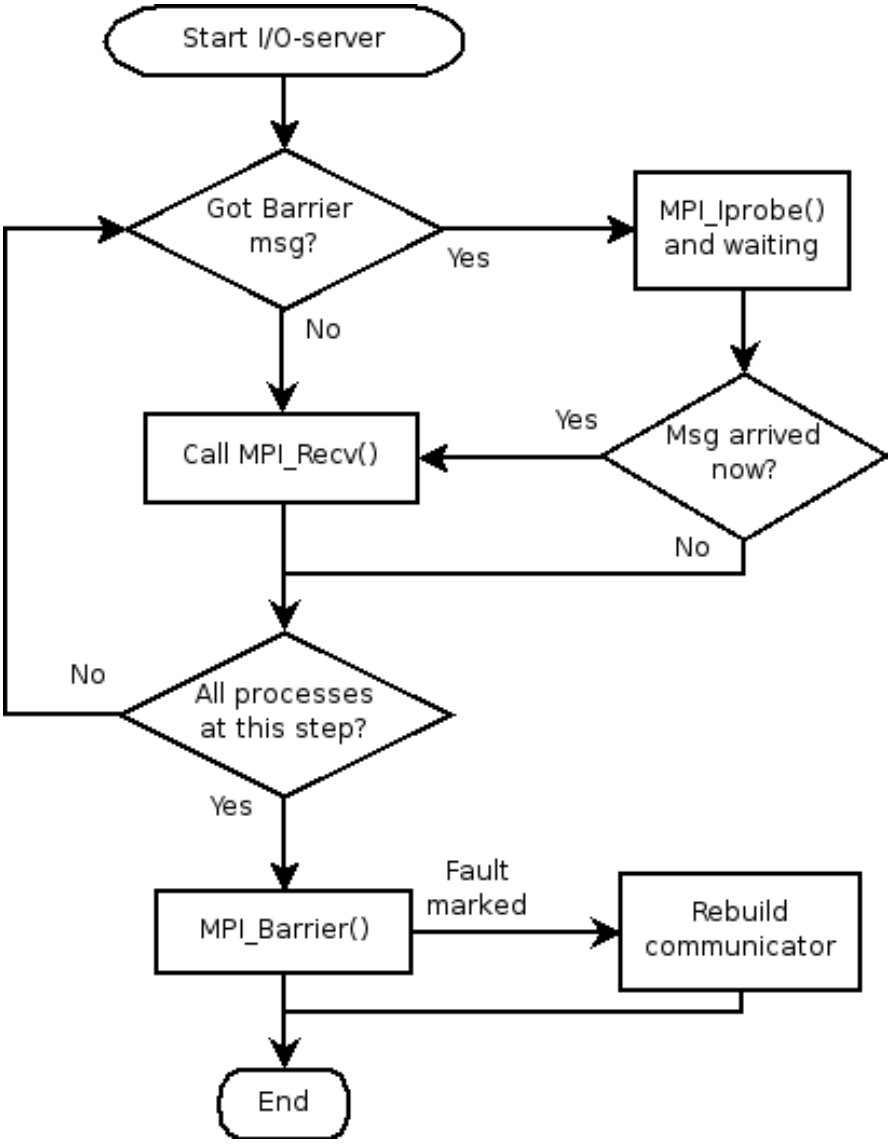

Figure 5.2: Illustration of the I/O server when getting barrier messages from the clients. This is meant to run for every round of shot processing. The server listens for messages with MPI_Recv(), but after the first barrier message, it enters a state calling MPI_Iprobe() and actual waiting for a given amount of seconds. If a message has arrived after some waiting, it calls MPI_Recv(), otherwise it assumes that there are failed processes, and marking a fault. If all processes has sent the barrier message, or a fault is marked, the MPI_Barrier() operation can be performed by all processes. If there is a communicator fault, the communicator is rebuilt, and the processes may continue on processing the next shot.

5.1.3 MPI Loop

To solve the above problems, there is another approach to implementing fault-tolerance on the test application. The FIO library, described in Section 3.1 uses three MPI functions; MPI_Send(), MPI_Recv() and MPI_Reduce(). Every place these functions are called, they can are encapsulated in a loop, and the return value is checked from the function call. If the value equals MPI_ERR_OTHER, there is a problem with the communicator, meaning there is a failed process, and the communicator needs to be rebuilt. All the other processes will eventually do a call to one of the MPI functions, and the error value will be returned, so the collective call to rebuild the communicator will happen on every alive process. After the communicator has been rebuilt, the loop will continue another time, so that the MPI operation actually will be performed, now in the new and smaller sized communicator, because the SHRINK option in FT-MPI has been used. The I/O server on rank 0 will be idle in calling MPI_Recv() during the computation of the shots. If rank 0 crashes, the whole application crashes, because the communicator is shrunk, and hence rank 1 in the previous communicator will become rank 0, but continue on processing shots. It will, like the other alive processes, call MPI_Send() with destination set to rank 0, and deadlock. Another issue is somewhat the same as with the MPI_Barrier() approach when the amount of shots is not evenly divided among the processes. If one or more of the processes fail on their last shot computation, when some processes have finished their amount, the application will hang on the communicator rebuild, because the finished processes are not calling a MPI function, thus not aware of that there is a failed process. If one can exclude the small potential that the I/O server will crash, or that some processes fail on their last computation of a shot, then this fault-tolerant implementation works well. Figure 5.3 illustrates this behaviour. Code snippet for this implementation can be found in Appendix B.3.

There are a few workarounds that needed to be done, though. The FT-MPI specification says that the return value from MPI functions are either MPI_SUCCESS or MPI_ERR_OTHER. If a communicator is marked as failed, and MPI_Send() is called, another value is returned, which is not defined in the FT-MPI system. It does not work for the process to rebuild the communicator either. If it calls MPI_Iprobe() two times with MPI_ANY_SOURCE and MPI_ANY_TAG and checks the return value for MPI_ERR_OTHER before the call to MPI_Send(), then it gets the right return value. The process may also rebuild the communicator together with all the other processes before the call to MPI_Send(). Code snippet for this implementation can be found in Appendix B.4.

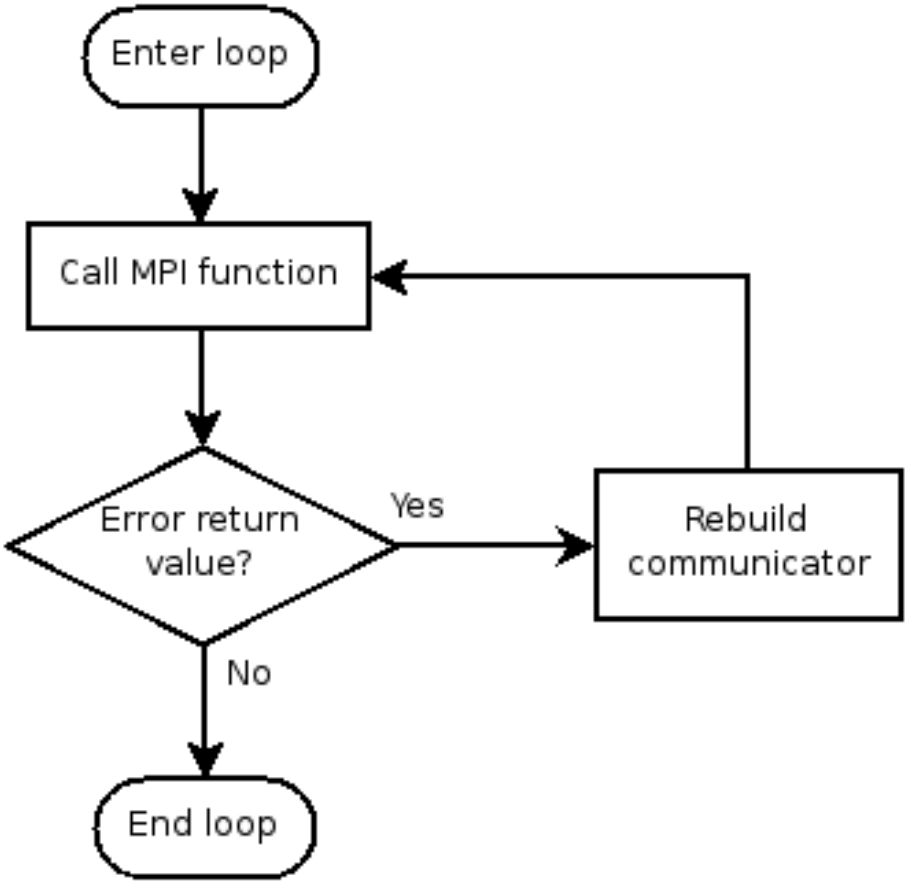

Figure 5.3: Illustration of every call to a MPI function, both the I/O server and the clients. All the processes call a MPI function before and after the shot processing. If the communicator has failed, the MPI function will return an error message, and the process must enter the rebuild communicator state, and wait for every process to do so, excluding the failed processes. After the communicator rebuild, the MPI function is called once again, so the application will go on and function properly.

5.2 Testing

5.2.1 Implementation in test application

The final implementation, described in Section 5.1.3, must be tested to approve that it is working. The idea of the testing is as follows:

1. Run on a cluster with nine nodes
   (a) Kill one process after the processing of two shots
   (b) Let the application run until it is finished - approx. 16 hours
   (c) List the output from the application, including output from FT-MPI
2. Run on a cluster with 100 nodes
   (a) Kill four processes - two during the first round of shot processing and two during the second round
   (b) Let the application run until it is finished - approx. 90 minutes
   (c) List the output from the application, including output from FT-MPI

The first test is meant to show that the fault-tolerant implementation works for a small scale cluster, and the second test for a large scale cluster. The two tests are run on two different clusters, both homogeneous.

The output from these tests show that a process is killed. The I/O server, which is idle, listening for a message from any process, first notices the failed communicator and enters the rebuilding communicator state. The other processes notice that the communicator has failed when they have finished the processing of their current shot. When every process has entered the rebuilding communicator state, the rebuilding can be performed and the processes which are alive may continue to process the next shot. FT-MPI will output some information during the exiting of the application, where every process tells that they have performed a recovery of the communicator. On the second test much of this information is cut, because it produces a lot of output, and it is practically the same output for every process. The places where the output is cut are clearly marked. The listings for the two tests can be found in Appendix C.

5.2.2 Measurement

This test is meant to show how much time is spent for FT-MPI in rebuilding the communicator. There are two communication modes that are interesting to test, and that is the SHRINK mode and the REBUILD mode, because ABORT just aborts every process, and the BLANK mode leaves open rank identifications unused. These are described in Section 2.5.1. The given numbers are the

walltime used in rebuilding the communicator, and the tests are done on a homogeneous cluster with 50 nodes and the processor Intel Xeon 2.8 GHz. It is tested with 5 to 50 nodes in the two communication modes. Table 5.1 shows the results from the tests, and Figure 5.4 shows the graph of the numbers.

The results show that the SHRINK mode compared to the REBUILD mode scales much better, because the values are somewhat the same with SHRINK, but with REBUILD the values are merely linear. The implementation of FT-MPI in the test application uses the SHRINK mode, and these test results show that it is suitable for scaling to a large number of nodes. A theori behind this is that with the REBUILD mode, every alive process must be informed that processes are rebuilt, but with the SHRINK mode, the failed processes are maybe left out with a O(1) algorithm, implying a O(n) algorithm for the REBUILD mode.

n	$REBUILD$	$SHRINK$
5	1.253082	0.345140
10	1.596352	0.339093
15	1.753929	0.354073
20	1.986104	0.352737
25	2.507687	0.356026
30	2.546442	0.363433
35	2.758086	0.364055
40	2.780654	0.357690
45	3.279314	0.364302
50	3.517437	0.362042

Table 5.1: Test of the wall time of rebuilding the communicator. The values show how much time is spent for FT-MPI to rebuild the communicator from 5 to 50 nodes.

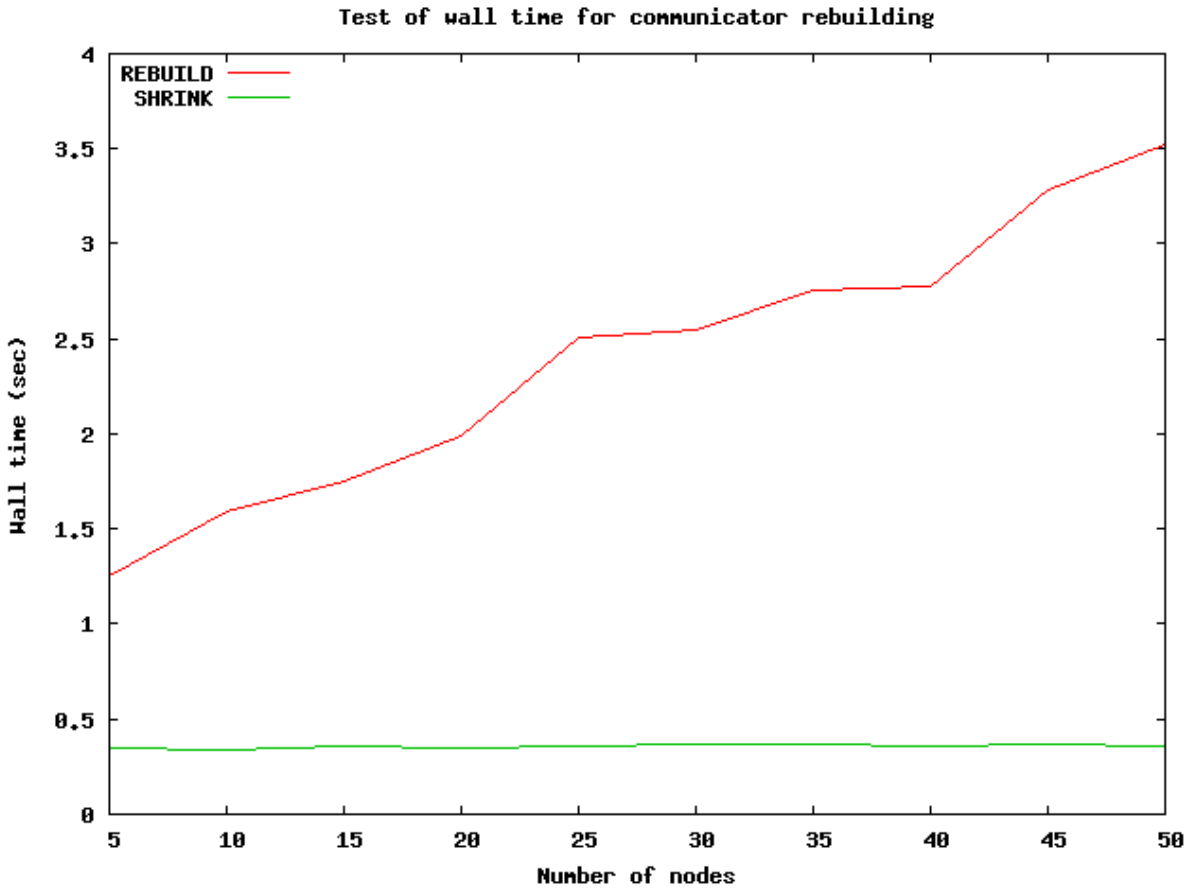

Figure 5.4: Test of wall time for communicator rebuilding. The values from Table 5.1 are shown in this graph. It shows the wall time in seconds for 5 to 50 nodes when FT-MPI is rebuilding the communicator.

5.2.3 Fault-tolerant model

The fault-tolerant implementation of the test application can be described with a model of the overall execution time based on a set of variables. These variables are described next.

- Overall execution time in seconds of application (E)
- Amount of shots in the dataset (D)
  The initial dataset may contain a large or small number of shots which are picked and modeled by the processes. The overall computation is finished when every shot is modeled.
- Amount of processors available (P)
  A cluster contains a number of available nodes, where each node has a given number of processes. One process run on one processor, which yields that the amount of processors is mapped directly to the number of processes.
- Probability of an overall failure (F)
  The probability that there will be process failures during the computation of the entire application, including the shot dataset. This variable must be written as a decimal number from 0 to 1.
- Average execution time of one shot (S)
  This is the time spent for one process to model one shot. On a Pentium 4 3.4 GHz processor the time spent is around 1800 seconds.
- Average time of communicator rebuilding (R)
  Section 5.2.2 describes the scaling of the FT-MPI communicator rebuilding. For the SHRINK option, the time spent is constant, and the average time is about 0.35 seconds.

The overall execution time in seconds is then modeled like this:

$$E(D,P,F,S,R) = \frac{D}{(P-(F\cdot P))}\cdot S + F\cdot P\cdot S + F\cdot P\cdot R \tag{5.1}$$

In this equation, the first term is the time of shot modeling with the alive processes, the second term is time spent in recomputing the shots in the end, and the third term describes total time spent in communicator rebuilding. The second term yields that the shots that are not modeled are run with a single process, and not in parallel, which means that this model is resilient to a low fault frequency.

To compensate for this, and do the recomputing of the shots in parallel when all nodes are alive, shots may be divided among the processors. It then comes down to that term two is replaced by only one S, since $F \cdot P < P$, meaning that the number of failed processes is less than the total number of processes. To

let this model support that if the probability of a failure equals zero, a need to introduce a new variable appears. This variable is called X and must be a boolean which is either 0 or 1. The meaning of the variable is
if (F > 0) *then* (X = 1) *else* (X = 0)
The new model with parallel execution of failed shots is then modeled like this:

$$E(D, P, F, S, R, X) = \frac{D}{(P - (F \cdot P))} \cdot S + S \cdot X + F \cdot P \cdot R \tag{5.2}$$

Figure 5.5 shows the two functions plotted in graphs. The free variable is F - the probability of an occurring failure, and the values are spread from 0 to 1. D = 231 because this is the amount of shots in the dataset received from Statoil. P = 9 because this is the number of processors in the small cluster from the testing of the test application. S = 1800 because this is the time of modeling one shot on the small cluster. R = 0.35 because this is the time measured in rebuilding the communicator with the SHRINK option. The graphs show that as the probability of an occurring failure during running of the application increases, the execution time increases approximately linearly from 0 to about 0.6, and after this point the graphs become exponential and will never converge. Time saved in running the failed shots in parallel decreases as the probability increases.

Figure 5.6 shows the graphs of the functions when the amount of processors in the cluster is 100, as in the testing of the large cluster. The graphs show that with more processors available, the execution time decreases dramatically, as expected. What is also expected is that the function 5.2 scales better than 5.1, because shots that have not been processed are run in parallel in the end. When the probability of an overall failure approaches 1, none of the graphs converge, and that is the same as with the small cluster.

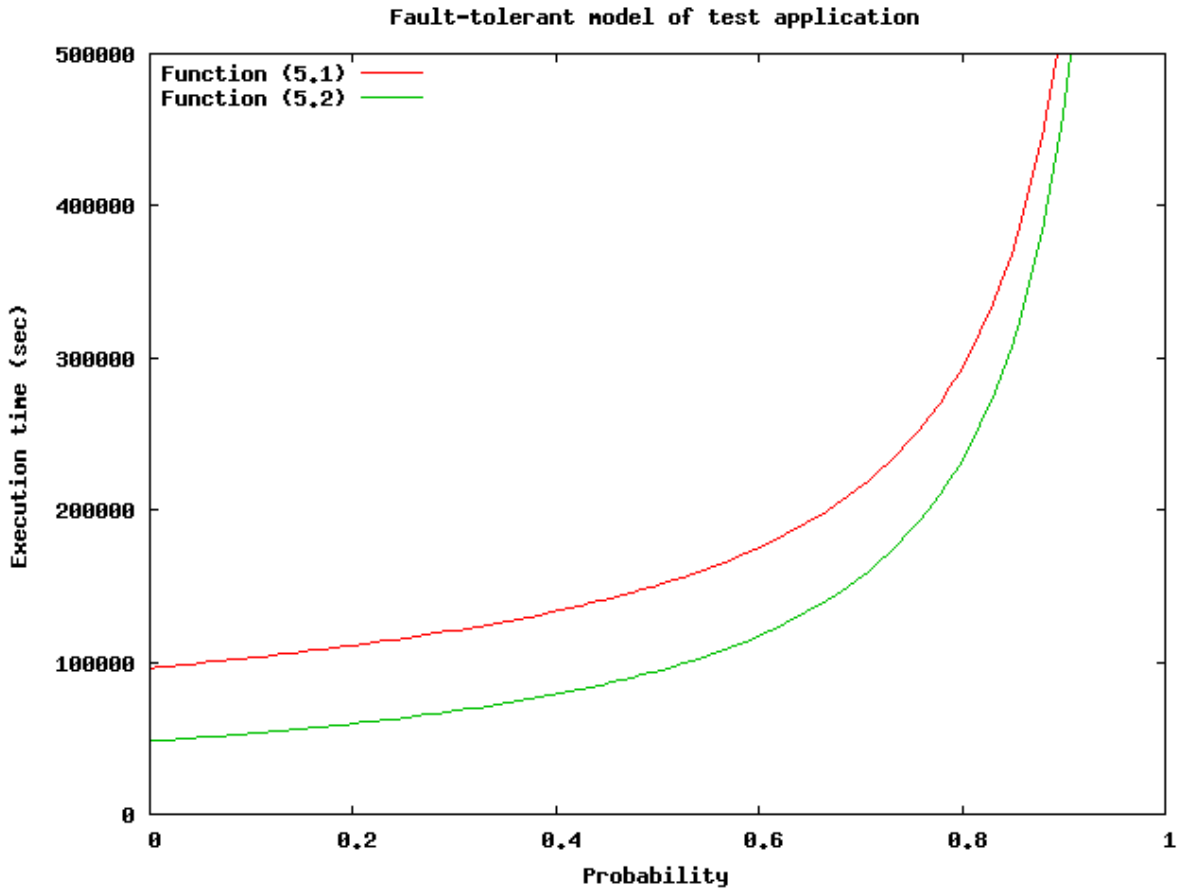

Figure 5.5: Graph of fault-tolerant model of test application, small cluster. The graphs show the execution time when the probability of an occurring failure of a process increases. Number of processors are 9.

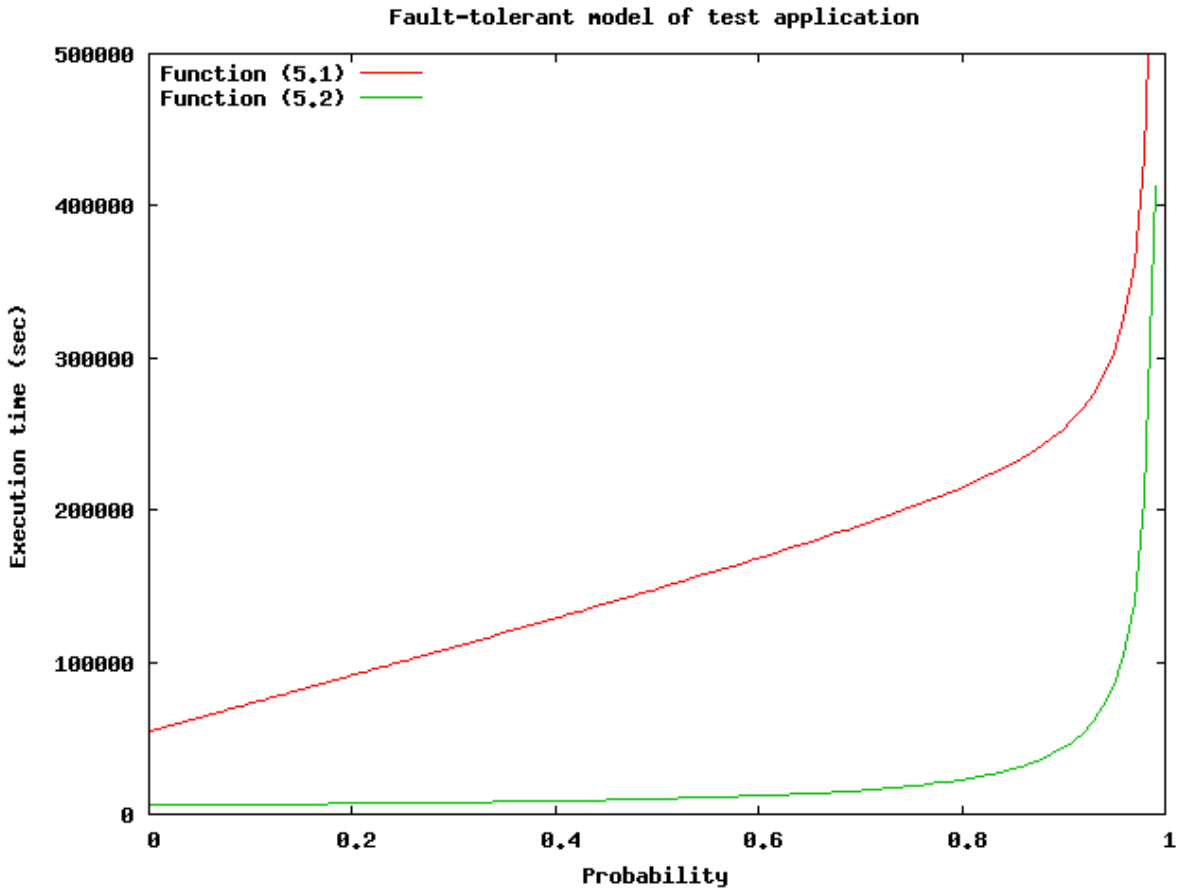

Figure 5.6: Graph of fault-tolerant model of test application, large cluster. The graphs show the execution time when the probability of an occurring failure of a process increases. Number of processors are 100.

Chapter 6

Conclusion

This thesis has focused on developing and analysing fault-tolerant techniques for MPI codes on computational clusters. Several MPI implementations are described, like MPICH1, MPICH2, LAM/MPI and Open MPI. Fault-tolerant MPI implementations and techniques were also described, such as FT-MPI, FEMPI, MPICH-V including its five protocols, the fault-tolerant functionality of Open MPI, and MPI Error handlers.

Our test application, a larger seismic simulation from Statoil Research Center Rotvoll, had no degree of fault-tolerance implemented when it was assigned to this thesis. If one process or one node crashed, the entire application also crashed. Now, their application is fault-tolerant at the degree of surviving a crash of n-2 nodes/processes, as process number 0 acts as an I/O server and there must be at least one process left to compute data. This last process and the I/O server must stay alive. Processes can also be restarted, if the test application were modified to support this. This is discussed further in the next section.

Three approaches to implement fault-tolerance on the test application were described. These were called MPI_Reduce(), MPI_Barrier(), and the final and current implementation MPI Loop. The first two are included to show the methods used on the way to the current implementation. Tests of the MPI Loop implementation are run on a small and a large cluster to show the fault-tolerant behaviour. A fault-tolerant model of this implementation was described with mathematical functions and graphs.

It is shown that the FT-MPI communication mode SHRINK scaled better than the REBUILD mode. This was because the wall time of communicator rebuild with SHRINK mode was constant and the REBUILD was linear.

Fault-tolerant techniques that were studied during this thesis work included checkpointing, MPI Error handlers, extending MPI, replication, fault detection, atomic clocks and multiple simultaneous failures. FT-MPI and MPI Error handlers were implemented in a fault-tolerant simulator which simulated the behaviour of the test application. The simulator was developed because the

test application was meant to run for a long time (e.g. several weeks), where the simulator was designed to simulate similar failures, but in a shorter period, and forcing processes to fail. Fault-tolerant techniques used in the simulator could then be adapted to the test application.

6.1 Future work

The techniques in Open MPI, described in Section 2.5.4, can be used to do a different approach of implementing fault-tolerance on the test application. This cannot be done until fault-tolerance is totally implemented in Open MPI. The need of running FT-MPI in combination with the fault-tolerant test application on the cluster will then vanish. This means that the ordinary "mpicc" and "mpirun" commands can be run, instead of the FT-MPI functions "ftmpicc" and "ftmpirun". To enable fault-tolerance with Open MPI in applications, variances of the parameter to "mpirun" can be used: "–mca ft-enable".

When processes have failed and the test application is finished, there exist shots that have not been processed which were in progress when the processes failed. These shots have to be processed afterwards to complete the overall computation. This could happen automatically, if this was integrated in the application. The user will then not notice, or does not have to bother, if some processes failed during the processing of the shots.

Processes could be restarted after a process crash, and these could run on the alive nodes. With FT-MPI this is done with the communication mode REBUILD. Then, processes which have failed will be restarted with the same rank number. What must be altered on the test application is the startup methods in the splfd2dmod program, so that the restarted process jumps gracefully to the inner loop. States of important variables must be checkpointed, so the restarted process can continue processing a new shot.

Bibliography

[1] University of Mannheim, University of Tennesse, and NERSC/LBNL. TOP500 Supercomputing Sites, 2007. Available from World Wide Web: http://www.top500.org/. This is a website. Date last visited: May 10, 2007.

[2] Argonne National Laboratory. Message passing interface, 2007. Available from World Wide Web: http://www-unix.mcs.anl.gov/mpi/. This is a website. Date last visited: January 23, 2007. Including links.

[3] Wikipedia. Message passing interface, 2007. Available from World Wide Web: http://en.wikipedia.org/w/index.php?title=Message_Passing_Interface&oldid=100474709. This is an electronic document. Date retrieved: January 23, 2007.

[4] Hubert Zimmermann. OSI reference model - the ISO model of architecture for open systems interconnection. *IEEE Transactions on communications*, COM-28(4), 1980. Available from World Wide Web: http://www.comsoc.org/livepubs/50_journals/pdf/RightsManagement_eid=136833.pdf.

[5] Wikipedia. Non-uniform memory access, 2007. Available from World Wide Web: http://en.wikipedia.org/w/index.php?title=Non-Uniform_Memory_Access&oldid=101909975. This is an electronic document. Date retrieved: January 24, 2007.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[7] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994. Available from World Wide Web: http://www.lam-mpi.org/download/files/lam-papers.tar.gz.

[8] Indiana University. LAM/MPI parallel computing, 2007. Available from World Wide Web: http://www.lam-mpi.org/. This is a website. Date last visited: January 25, 2007.

[9] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. *Proceedings, 11th European PVM/MPI Users' Group Meeting*, September 2004. Available from World Wide Web: http://www.open-mpi.org/papers/euro-pvmmpi-2004-overview/euro-pvmmpi-2004-overview.pdf.

[10] Graham E. Fagg and Jack Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. *Lecture Notes in Computer Science: Proceedings of EuroPVM-MPI 2000*, 1908(6):346–353, 2000. Available from World Wide Web: http://icl.cs.utk.edu/publications/pub-papers/2000/ft-mpi.pdf.

[11] Jack Dongarra, Al Geist, James Arthur Kohl, Philip M. Papadopoulos, and Vaidy Sunderam. HARNESS: Heterogeneous Adaptable Reconfigurable NEtworked SystemS. March 3 1998. Available from World Wide Web: http://www.csm.ornl.gov/harness/hpdc.ps.

[12] Rajagopal Subramaniyan, Vikas Aggarwal, Adam Jacobs, and Alan D. George. FEMPI: A Lightweight Fault-tolerant MPI for Embedded Cluster Systems. July 2006. Available from World Wide Web: http://www.hcs.ufl.edu/pubs/ESA2006a.pdf.

[13] Aurélien Bouteiller, Thomas Herault, Géraud Krawezik, Pierre Lemarinier, and Franck Cappello. MPICH-V: a multiprotocol fault tolerant MPI. *International Journal of High Performance Computing and Applications*, 2005. Available from World Wide Web: http://mpich-v.lri.fr/papers/ijhpca_mpichv.pdf.

[14] Laboratoire de Recherche en Informatique. MPICH-V MPI implementation for volatile resources, 2007. Available from World Wide Web: http://mpich-v.lri.fr/. This is a website. Date last visited: May 21, 2007.

[15] Wikipedia. Snapshot algorithm, 2006. Available from World Wide Web: http://en.wikipedia.org/w/index.php?title=Snapshot_algorithm&oldid=82477272. This is an electronic document. Date retrieved: January 31, 2007.

[16] The Open MPI Project. OPEN MPI Open Source High Performance Computing, 2007. Available from World Wide Web: http://www.open-mpi.org/. This is a website. Date last visited: May 25, 2007. Including links.

[17] William Gropp and Ewing Lusk. Fault Tolerance in MPI Programs. *Proceedings of the Cluster Computing and Grid Systems Conference, December 2002.*, 2002. Available from World Wide Web: http://www-unix.mcs.anl.gov/~gropp/bib/papers/2002/mpi-fault.pdf.

[18] Anne C. Elster, M. Ümit Uyar, and Anthony P. Reeves. Fault-tolerant Matrix Operations On Hypercube Multiprocessors. *IEEE International Conference on Parallel Processing, pp. 169-176*, August 1989.

Appendix A

Source Code of Simulator

A.1 Simulator Original

```
1  /*
2   * Simulator for fault-tolerance
3   * A number of random processes fail during computation
4   * This is the original version, simulating the test application
5   * Author: Knut Imar Hagen
6   */
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <unistd.h>
11 #include <string.h>
12 #include <time.h>
13 #include "mpi.h"
14
15 int totalsize = 2000; //total size of the square matrix, length & width
16
17 int help(int rank) {
18   if (rank == 0) {
19     printf("usage: simulator <procfail> <numloops>\n");
20     printf("    procfail    - Number of randomly failing ");
21     printf("processes during the computation\n");
22     printf("    numloops    - Number of loops of computation, ");
23     printf("to adjust computation time\n");
24     printf("    NOTE: procfail must be less or equal to numloops\n");
25   }
26   MPI_Finalize();
27   exit(0);
28 }
29
30 int main(int argc, char **argv) {
31   int rank,p,procfail,numloop,*failingprocs;
32   double *submat;
33
34   MPI_Init(&argc, &argv);
35   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
36   MPI_Comm_size(MPI_COMM_WORLD, &p);
37
```

```
38    //If there are not 2 arguments, print help and quit
39    if (argc != 3)
40      help(rank);
41
42    procfail = atoi(argv[1]);
43    numloop = atoi(argv[2]);
44
45    //number of failing processes must be less or equal to number of loops
46    if (procfail > numloop || procfail < 0 || numloop < 0) {
47      help(rank);
48    }
49
50    /* If the simulator shall fail some processes, let rank 0 make a random
51     * list of the failing processes and send the list to every process
52     */
53    if (procfail > 0) {
54      failingprocs = malloc(sizeof(int) * procfail);
55      memset(failingprocs,-1,sizeof(int) * procfail);
56      if (rank == 0) {
57        srand(time(NULL));
58        int k;
59        for (k=0;k<procfail;k++) {
60          int tmp = 0;
61          int ok = 1;
62          do {
63            ok = 1;
64            tmp = rand()%p;
65            int l;
66            for (l=0;l<k;l++) {
67              if (failingprocs[l] == tmp)
68                ok = 0;
69            }
70          }while(!ok);
71          failingprocs[k] = tmp;
72        }
73      }
74      MPI_Bcast(failingprocs,procfail,MPI_INT,0,MPI_COMM_WORLD);
75    }
76
77    //Allocate the submatrix
78    submat = malloc(sizeof(double) * totalsize * totalsize / p);
79    memset(submat, 0, sizeof(double) * totalsize * totalsize / p);
80
81    /* Initialize the submatrix
82     * the first element will always be rank number + 1
83     */
84    int i,j;
85    for (i=0;i<totalsize/p;i++) {
86      for (j=0;j<totalsize;j++) {
87        submat[i*totalsize + j] = (rank+1)*(j+i+1);
88      }
89    }
90
91    /* If the number of loops is set higher than 0, the computation will run
92     * the given number of times, and the processes listed to be failed
93     * will fail in a sequentially order in a regular manner
94     */
95    if (numloop > 0 ) {
```

```
96      int countfail = 0;
97      int upcountfail = 0;
98      if (procfail == 0)
99        procfail = 1;
100     else
101       upcountfail = numloop/procfail;
102     for (i=0;i<numloop;i++) {
103       if (numloop/procfail == upcountfail) {
104         if (countfail < procfail
105             && failingprocs[countfail++] == rank) {
106           printf("I am process %d and I am failing\n",rank);
107           fflush(stdout);
108           exit(0);
109         }
110         upcountfail = 0;
111       }
112       upcountfail++;
113       for (j=0;j<(totalsize*totalsize/p)/2;j++) {
114         double swp = submat[j];
115         submat[j] = submat[totalsize*totalsize/p-j];
116         submat[totalsize*totalsize/p-j] = swp;
117       }
118       for (j=0;j<(totalsize*totalsize/p)/2;j++) {
119         double swp = submat[totalsize*totalsize/p-j];
120         submat[totalsize*totalsize/p-j] = submat[j];
121         submat[j] = swp;
122       }
123     }
124   }
125
126   //Processes that have not failed will say that they are alive
127   printf("Process #%d of %d: submat[0]=%f\n", rank,p,submat[0]);
128   fflush(stdout);
129   double sum = 0;
130   //The first element in the matrix will be reduced
131   MPI_Reduce(submat,&sum,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
132   if (rank==0) {
133     printf("The result from the MPI_Reduce() operation is %f\n",sum);
134     fflush(stdout);
135   }
136   MPI_Finalize();
137 }
```

A.2 Simulator with FT-MPI implementation

```
1  /*
2   * Simulator for fault-tolerance
3   * For use with Harness/FT-MPI
4   * A number of random processes fail during computation
5   * and the processes are restarted
6   * This is the FT-MPI version - fault-tolerance is implemented
7   * Author: Knut Imar Hagen
8   */
9
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <unistd.h>
13 #include <string.h>
14 #include <time.h>
15 #include "mpi.h"
16
17 int totalsize = 2000; //total size of the square matrix, length & width
18 MPI_Comm comm;
19
20 void help(int rank) {
21   if (rank == 0) {
22     printf("usage: simulator <procfail> <numloops>\n");
23     printf("    procfail    - Number of randomly failing ");
24     printf("processes during the computation\n");
25     printf("    numloops    - Number of loops of computation, ");
26     printf("to adjust computation time\n");
27     printf("    NOTE: procfail must be less or equal to numloops\n");
28   }
29   MPI_Finalize();
30   exit(0);
31 }
32
33 int main(int argc, char **argv) {
34   int rank,p,rc,rs,procfail,numloop,*failingprocs;
35   double *submat;
36   comm = MPI_COMM_WORLD;
37
38   rc = MPI_Init(&argc, &argv);
39   if (rc==MPI_INIT_RESTARTED_NODE) {
40     printf("MPI_Init says I am a restarted process\n"); fflush(stdout);
41     rs = 1;
42   }
43   else {
44     printf("MPI_Init [%d]\n", rc); fflush(stdout);
45     rs = 0;
46   }
47
48   rc = MPI_Comm_rank(comm, &rank);
49   rc = MPI_Comm_size(comm, &p);
50
51   //If there are not 2 arguments, print help and quit
52   if (argc != 3)
53     help(rank);
54
55   procfail = atoi(argv[1]);
```

```
56    numloop = atoi(argv[2]);
57
58    //number of failing processes must be less or equal to number of loops
59    if (procfail > numloop || procfail < 0 || numloop < 0) {
60      help(rank);
61    }
62
63    /* If the simulator shall fail some processes, let rank 0 make a random
64     * list of the failing processes and send the list to every process
65     */
66    if (procfail > 0 && rs == 0) {
67      failingprocs = malloc(sizeof(int) * procfail);
68      memset(failingprocs,-1,sizeof(int) * procfail);
69      if (rank == 0 && rs == 0) {
70        srand(time(NULL));
71        int k;
72        for (k=0;k<procfail;k++) {
73          int tmp = 0;
74          int ok = 1;
75          do {
76            ok = 1;
77            tmp = rand()%p;
78            int l;
79            for (l=0;l<k;l++) {
80              if (failingprocs[l] == tmp)
81                ok = 0;
82            }
83          }while(!ok);
84          failingprocs[k] = tmp;
85        }
86      }
87      MPI_Bcast(failingprocs,procfail,MPI_INT,0,comm);
88    }
89
90    //Allocate the submatrix
91    submat = malloc(sizeof(double) * totalsize * totalsize / p);
92    memset(submat, 0, sizeof(double) * totalsize * totalsize / p);
93
94    /* Initialize the submatrix
95     * the first element will always be rank number + 1
96     */
97    int i,j;
98    for (i=0;i<totalsize/p;i++) {
99      for (j=0;j<totalsize;j++) {
100       submat[i*totalsize + j] = (rank+1)*(j+i+1);
101     }
102   }
103
104   /* If the number of loops is set higher than 0, the computation will run
105    * the given number of times, and the processes listed to be failed
106    * will fail in a sequentially order in a regular manner
107    */
108   if (numloop > 0 ) {
109     int countfail = 0;
110     int upcountfail = 0;
111     if (procfail == 0)
112       procfail = 1;
113     else
```

```
114         upcountfail = numloop/procfail;
115       for (i=0;i<numloop;i++) {
116         if (numloop/procfail == upcountfail && rs == 0) {
117           if (countfail < procfail
118               && failingprocs[countfail++] == rank) {
119             printf("I am process %d and I am failing\n",rank);
120             fflush(stdout);
121             exit(15);
122           }
123           upcountfail = 0;
124         }
125         upcountfail++;
126         for (j=0;j<(totalsize*totalsize/p)/2;j++) {
127           double swp = submat[j];
128           submat[j] = submat[totalsize*totalsize/p-j];
129           submat[totalsize*totalsize/p-j] = swp;
130         }
131         for (j=0;j<(totalsize*totalsize/p)/2;j++) {
132           double swp = submat[totalsize*totalsize/p-j];
133           submat[totalsize*totalsize/p-j] = submat[j];
134           submat[j] = swp;
135         }
136       }
137     }
138
139     do {
140       rc = MPI_Barrier(comm);
141       if (rc == MPI_ERR_OTHER) {
142         MPI_Comm newcomm;
143         newcomm = FT_MPI_CHECK_RECOVER;
144         MPI_Comm_dup(comm,&newcomm);
145         MPI_Comm_free(&comm);
146         comm = newcomm;
147       }
148     }while(rc == MPI_ERR_OTHER);
149
150     //Processes that have not failed will say that they are alive
151     printf("Process #%d of %d: submat[0]=%f\n", rank,p,submat[0]);
152     fflush(stdout);
153     double sum = 0;
154     //The first element in the matrix will be reduced
155     rc = MPI_Reduce(submat,&sum,1,MPI_DOUBLE,MPI_SUM,0,comm);
156     if (rank==0) {
157       printf("The result from the MPI_Reduce() operation is %f\n",sum);
158       fflush(stdout);
159     }
160     MPI_Finalize();
161   }
```

A.3 Simulator with implementation of error handlers

A.3.1 Master program

```
1  /*
2   * Simulator for fault-tolerance
3   * A number of random processes fail during computation
4   * This is the master/slave-version with fault-tolerance
5   * Author: Knut Imar Hagen
6   */
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <unistd.h>
11 #include <string.h>
12 #include <time.h>
13 #include <errno.h>
14 #include "mpi.h"
15
16 #define IC_TAG 1
17
18 static MPI_Comm *slave_comm;
19 static MPI_Request *recvrequest;
20 static MPI_Status *status;
21 static int totalsize = 2000; //total size of the square matrix, length & width
22
23 int help(int rank) {
24   if (rank == 0) {
25     printf("usage: simulator <procfail> <numloops>\n");
26     printf("    procfail    - Number of randomly failing ");
27     printf("processes during the computation\n");
28     printf("    numloops    - Number of loops of computation, ");
29     printf("to adjust computation time\n");
30     printf("    NOTE: procfail must be less or equal to numloops\n");
31   }
32   MPI_Finalize();
33   exit(0);
34 }
35
36 int main(int argc, char **argv) {
37   int i,j,rank,icrank,icsize,origsize;
38   int procfail,numloop,*failingprocs,ierr,flag,universe;
39   double *submat,*recvval;
40   void **universeattr;
41   char filename[100];
42   FILE *fp;
43   MPI_Info simulatorinfo;
44
45   MPI_Init(&argc, &argv);
46   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
47   MPI_Comm_size(MPI_COMM_WORLD, &origsize);
48
49   //If there are not 2 arguments, print help and quit
50   if (argc != 3)
51     help(rank);
```

```
52
53    procfail = atoi(argv[1]);
54    numloop = atoi(argv[2]);
55
56    //number of failing processes must be less or equal to number of loops
57    if (procfail > numloop || procfail < 0 || numloop < 0) {
58      help(rank);
59    }
60
61    /* If the simulator shall fail some processes, let rank 0 make a random
62     * list of the failing processes and send the list to every process.
63     * Rank 0 will not fail, because this will be the manager process
64     */
65    if (procfail > 0) {
66      failingprocs = malloc(sizeof(int) * procfail);
67      memset(failingprocs,0,sizeof(int) * procfail);
68      if (rank == 0) {
69        srand(time(NULL));
70        int k;
71        for (k=0;k<procfail;k++) {
72          int tmp = 0;
73          int ok = 1;
74          do {
75            ok = 1;
76            tmp = rand()%origsize;
77            int l;
78            for (l=0;l<k;l++) {
79              if (failingprocs[l] == tmp)
80                ok = 0;
81            }
82            if (tmp == 0)
83              ok = 0;
84          }while(!ok);
85          failingprocs[k] = tmp;
86        }
87      }
88      //MPI_Bcast(failingprocs,procfail,MPI_INT,0,MPI_COMM_WORLD);
89    }
90
91    //Getting the universe size
92    MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, &universeattr, &flag);
93    if(flag==1)
94      universe = (int) *universeattr;
95
96    //Initializing communicators, requests and receive values
97    slave_comm = (MPI_Comm *) malloc((unsigned) (universe*sizeof(MPI_Comm)));
98    recvrequest = (MPI_Request *) malloc((unsigned) (universe*sizeof(MPI_Comm)));
99    recvval = (double *) malloc((unsigned) (universe*sizeof(double)));
100   for (i=0;i<universe;++i) {
101     slave_comm[i] = MPI_COMM_NULL;
102     recvrequest[i] = MPI_REQUEST_NULL;
103   }
104
105   //Spawn the slaves
106   sprintf(filename, "./simulator_fault_schema");
107   MPI_Info_create(&simulatorinfo);
108   MPI_Info_set(simulatorinfo, "file", filename);
109   for (i = 0; i < universe; ++i) {
```

```
110       //Create a temporary application schema file
111       fp = fopen(filename, "w");
112       if (fp == NULL) {
113         printf("Could not open file %s\n", filename);
114         MPI_Abort(MPI_COMM_WORLD, -15);
115       }
116       fprintf(fp, "c%d ./simulator_errhandler_slave\n",i);
117       fclose(fp);
118
119       //Spawn
120       MPI_Comm_spawn(0, MPI_ARGV_NULL, 0,simulatorinfo, 0, MPI_COMM_SELF, &(slave_co
121       if (ierr != MPI_SUCCESS) {
122         printf("Spawn Error %d\n", ierr);
123         MPI_Abort(MPI_COMM_WORLD, -15);
124       }
125     }
126
127     MPI_Info_free(&simulatorinfo);
128     unlink(filename);
129
130     //Set error handlers on every communicator
131     for (i = 0; i < universe; ++i)
132       MPI_Errhandler_set(slave_comm[i], MPI_ERRORS_RETURN);
133     MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
134     MPI_Errhandler_set(MPI_COMM_SELF,MPI_ERRORS_RETURN);
135
136     //Sending slaveid and list of failing processes
137     for (i=0;i<universe;++i) {
138       MPI_Send(&i,1,MPI_INT,0,IC_TAG,slave_comm[i]);
139             MPI_Send(&numloop,1,MPI_INT,0,IC_TAG,slave_comm[i]);
140       MPI_Send(&procfail,1,MPI_INT,0,IC_TAG,slave_comm[i]);
141             MPI_Send(failingprocs,procfail,MPI_INT,0,IC_TAG,slave_comm[i]);
142     }
143
144     //Checking for failed processes and leaves them out while gathering data
145     double sum = 0.0;
146     int waitrank = 0;
147     for (i=0;i<universe;++i) {
148       MPI_Irecv(&(recvval[i]),1,MPI_DOUBLE,0,IC_TAG,slave_comm[i],&(recvrequest[i])
149     }
150     for (i=0;i<universe;++i) {
151       ierr = MPI_Waitany(universe,recvrequest,&waitrank,status);
152       if (ierr != MPI_SUCCESS && waitrank >=0) {
153         printf("Not success in receiving from %d\n",waitrank);
154         recvrequest[waitrank] = MPI_REQUEST_NULL;
155         MPI_Comm_free(&(slave_comm[waitrank]));
156       }else{
157         sum+=recvval[i];
158       }
159     }
160
161     printf("The result from the sum operation is %f\n",sum);
162     fflush(stdout);
163
164     MPI_Finalize();
165     exit(0);
166   }
```

A.3.2 Slave program

```
1  /*
2   * Simulator for fault-tolerance
3   * A number of random processes fail during computation
4   * This is the master/slave-version with fault-tolerance
5   * Author: Knut Imar Hagen
6   */
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <unistd.h>
11 #include <string.h>
12 #include <time.h>
13 #include <errno.h>
14 #include "mpi.h"
15
16 #define IC_TAG 1
17
18 int totalsize = 2000; //total size of the square matrix, length & width
19
20 int main(int argc, char **argv) {
21   int i,j,slaveid,origsize,currsize;
22   int procfail,numloop,*failingprocs,ierr,flag,universe;
23   double *submat,val;
24   void **universeattr;
25
26   MPI_Init(&argc, &argv);
27   MPI_Comm_size(MPI_COMM_WORLD, &origsize);
28   currsize = origsize;
29   MPI_Comm master_comm;
30   MPI_Status *status;
31   MPI_Comm_get_parent(&master_comm);
32   if (master_comm == MPI_COMM_NULL)
33     printf("NULL-COMMUNICATOR");
34   MPI_Recv(&slaveid,1,MPI_INT,0,IC_TAG,master_comm,MPI_STATUS_IGNORE);
35   MPI_Recv(&numloop,1,MPI_INT,0,IC_TAG,master_comm,MPI_STATUS_IGNORE);
36   MPI_Recv(&procfail,1,MPI_INT,0,IC_TAG,master_comm,MPI_STATUS_IGNORE);
37   failingprocs = malloc((unsigned)sizeof(int) * procfail);
38   memset(failingprocs,0,sizeof(int)*procfail);
39   MPI_Recv(failingprocs,procfail,MPI_INT,0,IC_TAG,master_comm,MPI_STATUS_IGNORE);
40
41   //Allocate the submatrix
42   submat = malloc(sizeof(double) * totalsize * totalsize / origsize);
43   memset(submat, 0, sizeof(double) * totalsize * totalsize / origsize);
44
45   /* Initialize the submatrix
46    * the first element will always be rank number + 1
47    */
48   for (i=0;i<totalsize/origsize;i++) {
49     for (j=0;j<totalsize;j++) {
50       submat[i*totalsize + j] = (slaveid+1)*(j+i+1);
51     }
52   }
53
54   /* If the number of loops is set higher than 0, the computation will run
55    * the given number of times, and the processes listed to be failed
56    * will fail in a sequentially order in a regular manner
```

```
57      */
58    if (numloop > 0 ) {
59      int countfail = 0;
60      int upcountfail = 0;
61      if (procfail == 0)
62        procfail = 1;
63      else
64        upcountfail = numloop/procfail;
65      for (i=0;i<numloop;i++) {
66        if (numloop/procfail == upcountfail) {
67          if (countfail < procfail
68              && failingprocs[countfail++] == slaveid) {
69            printf("I am slave %d and I am failing\n",slaveid);
70            fflush(stdout);
71            exit(10);
72          }
73          upcountfail = 0;
74        }
75        upcountfail++;
76        for (j=0;j<(totalsize*totalsize/origsize)/2;j++) {
77          double swp = submat[j];
78          submat[j] = submat[totalsize*totalsize/origsize-j];
79          submat[totalsize*totalsize/origsize-j] = swp;
80        }
81        for (j=0;j<(totalsize*totalsize/origsize)/2;j++) {
82          double swp = submat[totalsize*totalsize/origsize-j];
83          submat[totalsize*totalsize/origsize-j] = submat[j];
84          submat[j] = swp;
85        }
86      }
87    }
88
89
90    //Processes that have not failed will say that they are alive
91    printf("Slave #%d: submat[0]=%f\n", slaveid,submat[0]);
92    fflush(stdout);
93    MPI_Send(submat,1,MPI_DOUBLE,0,IC_TAG,master_comm);
94    MPI_Finalize();
95    exit(0);
96  }
```

Appendix B

Code snippets from the Test application

B.1 MPI_Reduce()

Described in Section 5.1.1. Added code to test application:
Line numbers 16-41,47,54,56,59,69-80,145-156

```
1  ! I/O server running the PioFm() function
2
3  integer function PioFm()
4    .
5    .
6    .
7
8  !-------------------------------------------------
9  !   Main loop until all clients are finished
10 !-------------------------------------------------
11
12   clients: do
13
14     if (pcount >= np-1) exit
15
16 !---  Listen for a message from any of the clients
17     if (reducing > 0)then
18 !---  Has entered reduce routine
19       flag = 0
20       waited = 0
21       faultoccur = 0
22       numprobes = 0
23       do
24 !---    Checking if a message has arrived with the flag
25         call MPI_IPROBE(msgsrc,ntag,MPI_COMM_WORLD,flag,status,ierr)
26         numprobes=numprobes+1
27         if (flag==1) exit
28         if (numprobes>=1000000) then
29 !---      Waiting with FioWait if more than 1 million Iprobes
30           waittime=FioWait(waittime) ! Waiting for #waittime seconds
```

```
31            waited=waited+1
32          endif
33          if (waited >= 10)then
34  !——      Has waited 10 times - communicator has failed
35            faultoccur = 1
36            msg(1) = PIOSTACK
37            exit
38          endif
39        enddo
40      endif
41      if (faultoccur==0) then
42        call MPI_RECV(msg,2,MPI_INTEGER,msgsrc,ntag,MPI_COMM_WORLD,status,ierr)
43        who = status(MPI_SOURCE) ! sender of the message
44        lu  = msg(2)    ! Get file descriptor
45                        ! In some cases this is not the file descriptor,
46                        ! but in those cases lu won't be used.
47      endif
48
49  !——Decode message
50
51      select case (msg(1))
52
53        case (PIOSTACK)
54          if (faultoccur==0) then
55            call MPI_RECV(msg,2,MPI_INTEGER,who,ntag,MPI_COMM_WORLD,status,ierr)
56          endif
57          reduce=reduce+1     ! Keep track of no of processors
58
59          if(faultoccur==1 .or. reduce == np-1)then
60            if(PioLus == -1)then
61              PioLus = lu
62            endif
63            ndata = msg(2)
64            allocate(stack(ndata))
65            allocate(rdata(ndata))
66            stack=0.0
67            rdata=0.0
68            call MPI_REDUCE(rdata,stack,ndata,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD,ierr)
69            if ( ierr .eq. MPI_ERR_OTHER ) then
70  !——          Rebuild the communicator
71              oldcomm = MPI_COMM_WORLD
72              newcomm = FT_MPI_CHECK_RECOVER
73              call MPI_Comm_dup (oldcomm, newcomm, ierr)
74              call MPI_Comm_rank (MPI_COMM_WORLD, PioRank, ierr)
75              call MPI_Comm_size (MPI_COMM_WORLD, np, ierr)
76              oldcomm = newcomm
77              np = PioNp
78  !——          Calling MPI_REDUCE one more time
79          call MPI_REDUCE(rdata,stack,ndata,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD,ierr)
80            endif
81            istat = FioWrite(PioLus,stack,ndata)
82            do i=1,np-1
83              msg(1) = PIOSTACKA
84              msg(2) = istat
85              who=i
86              call MPI_SEND(msg,2,MPI_INTEGER,who,ntag,MPI_COMM_WORLD,status,ierr)
87            enddo
88            deallocate(stack)
```

```
89              deallocate(rdata)
90              reduce=0
91              faultoccur=0
92              reducing=0
93              numprobes=0
94            endif
95            msgsrc=MPI_ANY_SOURCE
96
97          case (...
98          .
99          .
100         .
101         .
102       end select
103     end do clients
104     return
105   end function PioFm
106
107
108
109
110   ! Clients running the following function PioStack()
111
112   integer function PioFstack(fd,data,nreal)
113     implicit none
114     integer,intent(in)             :: fd     ! File descriptor
115     real,dimension(:),intent(in)   :: data   ! Array of floats
116     integer,intent(in)             :: nreal  ! No of floats to write
117     !-------------------------------------------------------------
118     integer                                 :: who      ! Process id
119     integer,dimension(2)                    :: msg      ! Message
120     integer                                 :: ntag     ! Tag
121     integer,dimension(MPI_STATUS_SIZE)      :: status   ! Mpi status
122     integer                                 :: ierr     ! Return flag
123     integer                                 :: nb       ! No of bytes to write
124     real,dimension(:),allocatable           :: dummy    ! receive buffer
125
126     allocate(dummy(nreal))
127
128   !-Send a stack request
129     msg(1) = PIOSTACK
130     msg(2) = fd
131     who = 0
132     ntag = 1
133     call MPI_SEND(msg,2,MPI_INTEGER,who,ntag,MPI_COMM_WORLD,status,ierr)
134
135   !--- Send the number of reals to stack
136     msg(1) = PIOSTACK
137     nb = nreal
138     msg(2) = nb
139     who = 0
140     ntag = 1
141     call MPI_SEND(msg,2,MPI_INTEGER,who,ntag,MPI_COMM_WORLD,status,ierr)
142
143   !-Do the reduce
144     call MPI_REDUCE(data,dummy,nreal,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD,ierr)
145     if ( ierr .eq. MPI_ERR_OTHER ) then
146   !---Rebuild the communicator
```

```
147       oldcomm = MPI_COMM_WORLD
148       newcomm = FT_MPI_CHECK_RECOVER
149       call MPI_Comm_dup (oldcomm, newcomm, ierr)
150       call MPI_Comm_rank (MPI_COMM_WORLD, PioRank, ierr)
151       call MPI_Comm_size (MPI_COMM_WORLD, np, ierr)
152       oldcomm = newcomm
153       np = PioNp
154  !---Calling MPI_REDUCE one more time
155       call MPI_REDUCE(data,dummy,nreal,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD,ierr)
156     endif
157
158  !-Wait until operation is completed
159     call MPI_RECV(msg,2,MPI_INTEGER,who,ntag,MPI_COMM_WORLD,status,ierr)
160
161  !--- Return status
162     PioFstack = msg(2)
163     deallocate(dummy)
164     return
165  end function PioFstack
```

B.2 MPI_Barrier()

Described in Section 5.1.2. Added code to test application:
Line numbers 16-41,53-73,90-123

```
1  ! I/O server running the PioFm() function
2
3  integer function PioFm()
4    .
5    .
6    .
7
8  !-------------------------------------------------
9  !   Main loop until all clients are finished
10 !-------------------------------------------------
11
12   clients: do
13
14     if (pcount >= np-1) exit
15
16 !---  Listen for a message from any of the clients
17     if (faultt > 0)then
18 !---  Has entered fault routine
19       flag = 0
20       waited = 0
21       faultoccur = 0
22       numprobes = 0
23       do
24 !---    Checking if a message has arrived with the flag
25         call MPI_IPROBE(msgsrc,ntag,MPI_COMM_WORLD,flag,status,ierr)
26         numprobes=numprobes+1
27         if (flag==1) exit
28         if (numprobes>=1000000) then
29 !---      Waiting with FioWait if more than 1 million Iprobes
30           waittime=FioWait(waittime) ! Waiting for #waittime seconds
31           waited=waited+1
32         endif
33         if (waited >= 10)then
34 !---      Has waited 10 times - communicator has failed
35           faultoccur = 1
36           msg(1) = PIOSTACKFAULT
37           exit
38         endif
39       enddo
40     endif
41     if (faultoccur==0) then
42       call MPI_RECV(msg,2,MPI_INTEGER,msgsrc,ntag,MPI_COMM_WORLD,status,ierr)
43       who = status(MPI_SOURCE) ! sender of the message
44       lu  = msg(2)    ! Get file descriptor
45                       ! In some cases this is not the file descriptor,
46                       ! but in those cases lu won't be used.
47     endif
48
49 !---Decode message
50
51     select case (msg(1))
```

```
52
53          case (PIOSTACKFAULT)
54            faultt=faultt+1     ! Keep track of no of processors
55  !——       I/O server has received a PIOSTACKFAULT message
56            if(faultoccur==1 .or. faultt == np-1)then
57  !——        Every process has sent the message or a fault has occured
58  !——        Calling MPI_BARRIER to get a return value and check for fail
59              call MPI_BARRIER(MPI_COMM_WORLD,ierr)
60              if ( ierr .eq. MPI_ERR_OTHER ) then
61  !——            Rebuild the communicator
62                oldcomm = MPI_COMM_WORLD
63                newcomm = FT_MPI_CHECK_RECOVER
64                call MPI_Comm_dup (oldcomm, newcomm, ierr)
65                call MPI_Comm_rank (MPI_COMM_WORLD, PioRank, ierr)
66                call MPI_Comm_size (MPI_COMM_WORLD, np, ierr)
67                oldcomm = newcomm
68                np = PioNp
69              endif
70              faultt=0
71              numprobes=0
72            endif
73            msgsrc=MPI_ANY_SOURCE
74
75          case (...
76          .
77          .
78          .
79          .
80        end select
81      end do clients
82      return
83  end function PioFm
84
85
86
87
88  ! Clients running the following function PioStackFault()
89
90      integer function PioFstackFault(rank,np)
91        implicit none
92        integer, intent(out) :: rank ! node no
93        integer, intent(out) :: np   ! the number of nodes
94      !-----------------------------------------------------------------
95        integer                                 :: who      ! Process id
96        integer,dimension(2)                    :: msg      ! Message
97        integer                                 :: ntag     ! Tag
98        integer,dimension(MPI_STATUS_SIZE)      :: status   ! Mpi status
99        integer                                 :: ierr     ! Return flag
100       integer                                 :: nb       ! No of bytes to write
101
102     !-Send a stackfault request
103       msg(1) = PIOSTACKFAULT
104       msg(2) = -1
105       who = 0
106       ntag = 1
107       call MPI_SEND(msg,2,MPI_INTEGER,who,ntag,MPI_COMM_WORLD,status,ierr)
108
109     !-Check for failed processes and rebuild the communicator
```

```
110       call MPI_BARRIER(MPI_COMM_WORLD, ierr)
111       if ( ierr .eq. MPI_ERR_OTHER ) then
112         oldcomm = MPI_COMM_WORLD
113         newcomm = FT_MPI_CHECK_RECOVER
114         call MPI_Comm_dup (oldcomm, newcomm, ierr)
115         call MPI_Comm_rank (MPI_COMM_WORLD, rank, ierr)
116         call MPI_Comm_size (MPI_COMM_WORLD, np, ierr)
117         oldcomm = newcomm
118       endif
119
120     !--- Return status
121       PioFstackFault = msg(2)
122       return
123     end function PioFstackFault
```

B.3 MPI Loop

Described in Section 5.1.3. Added code to test application:
Everything, except the actual MPI calls

```
1  ! Both the I/O server and the clients call MPI functions in a loop
2  ! and checks the return value with the function PioFault()
3  ! which tells if the loop shall continue
4
5  !-Loop for MPI_SEND
6    do
7      call PioPresend()
8      call MPI_SEND(msg,2,MPI_INTEGER,who,ntag,MPI_COMM_WORLD,status,ierr)
9      mpierr = PioFault(ierr)
10     if (mpierr == 0) exit
11   enddo
12
13 !-Loop for MPI_RECV
14   do
15     call MPI_RECV(msg,2,MPI_INTEGER,who,ntag,MPI_COMM_WORLD,status,ierr)
16     mpierr = PioFault(ierr)
17     if (mpierr == 0) exit
18   enddo
19
20 !-Loop for MPI_REDUCE
21   do
22     call MPI_REDUCE(data,dummy,nreal,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD,ierr)
23     mpierr = PioFault(ierr)
24     if (mpierr == 0) exit
25   enddo
26
27
28
29
30 ! Function checks the return value from a MPI function and
31 ! rebuilds the communicator if there is an error
32
33 integer function PioFault(ierr)
34   implicit none
35   integer, intent(in)  :: ierr ! mpi error
36 !-------------------------------------------------------------------
37   integer              :: oldcomm    ! old communicator
38   integer              :: newcomm    ! new communicator
39   integer              :: localerr   ! error handle
40   integer              :: waitsec    ! waiting seconds
41   integer              :: wasfault   ! was there a fault?
42   integer              :: debugfault ! output debug info?
43
44   wasfault = 0
45   debugfault = 1
46
47 !-Check for failed processes and fix the communicator
48   if ( ierr == MPI_ERR_OTHER) then
49     if(debugfault>=1)then
50       print *,"Process#",PioRank," is rebuilding the communicator"
51     endif
```

```
52        oldcomm = MPI_COMM_WORLD
53        newcomm = FT_MPI_CHECK_RECOVER
54        call MPI_Comm_dup (oldcomm, newcomm, localerr)
55        call MPI_Comm_rank (MPI_COMM_WORLD, PioRank, localerr)
56        call MPI_Comm_size (MPI_COMM_WORLD, PioNp, localerr)
57        oldcomm = newcomm
58        wasfault = 1
59        if (debugfault>=1)then
60          print *, "Process#",PioRank," has rebuilt the communicator"
61        endif
62      endif
63
64      PioFault = wasfault
65      return
66    end function PioFault
```

B.4 PioSend()

Described in Section 5.1.3. Added code to test application:
Every line

```
1  ! The subroutine calls MPI_IPROBE two times as a workaround for
2  ! erroneous values from MPI_SEND when there is a failed communicator
3
4  subroutine PioPresend()
5    implicit none
6    integer                                 :: flag     ! Process id
7    integer,dimension(MPI_STATUS_SIZE)      :: status   ! Mpi status
8    integer                                 :: ierr     ! Return flag
9
10   call MPI_IPROBE(MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,flag,status,ierr)
11   mpierr = PioFault(ierr)
12   call MPI_IPROBE(MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,flag,status,ierr)
13   mpierr = PioFault(ierr)
14 end subroutine PioPresend
```

Appendix C

Output from testing of test application

C.1 Small cluster

```
 ** This is splfd2dmod version 1.0 multi processor mode node no:
0
 --- No of nodes:             9
* Server running!
 ** This is splfd2dmod version 1.0 multi processor mode node no:
1
 ** This is splfd2dmod version 1.0 multi processor mode node no:
2
 ** This is splfd2dmod version 1.0 multi processor mode node no:
3
 ** This is splfd2dmod version 1.0 multi processor mode node no:
5
 ** This is splfd2dmod version 1.0 multi processor mode node no:
6
 ** This is splfd2dmod version 1.0 multi processor mode node no:
7
 ** This is splfd2dmod version 1.0 multi processor mode node no:
8
 ** This is splfd2dmod version 1.0 multi processor mode node no:
4
forrtl: error (78): process killed (SIGTERM)
[0] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#           0  is rebuilding the communicator
 Process#           2  is rebuilding the communicator
[2] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#           5  is rebuilding the communicator
[5] FTMPI Error with communicator 0, error code -19 Known error not in list
[3] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#           3  is rebuilding the communicator
 Process#           7  is rebuilding the communicator
[7] FTMPI Error with communicator 0, error code -19 Known error not in list
[6] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#           6  is rebuilding the communicator
[4] FTMPI Error with communicator 0, error code -19 Known error not in list
```

```
 Process#           4  is rebuilding the communicator
[8] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#           8  is rebuilding the communicator
 Process#           3  has rebuilt the communicator
 Process#           2  has rebuilt the communicator
 Process#           7  has rebuilt the communicator
 Process#           6  has rebuilt the communicator
 Process#           5  has rebuilt the communicator
 Process#           4  has rebuilt the communicator
 Process#           1  has rebuilt the communicator
 Process#           0  has rebuilt the communicator
Error event set for channel 3
Error event set for channel 3
Error event set for channel 3
Error event set for channel 4
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Callback on CON ERROR chan 3 con 6 Sock = 0
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Callback on CON ERROR chan 3 con 4 Sock = 0
Callback on CON ERROR chan 4 con 6 Sock = 19
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Callback on CON ERROR chan 3 con 7 Sock = 0
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Error event set for channel 4
Error event set for channel 2
Error event set for channel 5
Error event set for channel 6
```

```
Error event set for channel 9
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Callback on CON ERROR chan 4 con 4 Sock = 18
Callback on CON ERROR chan 2 con 3 Sock = 0
Callback on CON ERROR chan 5 con 6 Sock = 23
Callback on CON ERROR chan 6 con 7 Sock = 19
Callback on CON ERROR chan 9 con 5 Sock = 22
```

C.2 Large cluster

```
 ** This is splfd2dmod version 1.0 multi processor mode node no:
12
 ** This is splfd2dmod version 1.0 multi processor mode node no:
13
 ** This is splfd2dmod version 1.0 multi processor mode node no:
23
 ** This is splfd2dmod version 1.0 multi processor mode node no:
14
 ** This is splfd2dmod version 1.0 multi processor mode node no:
15
 ** This is splfd2dmod version 1.0 multi processor mode node no:
17
 ** This is splfd2dmod version 1.0 multi processor mode node no:
24
 .
 <<<CUT>>>
 .
 ** This is splfd2dmod version 1.0 multi processor mode node no:
0
 ** This is splfd2dmod version 1.0 multi processor mode node no:
1
 ** This is splfd2dmod version 1.0 multi processor mode node no:
2
 ** This is splfd2dmod version 1.0 multi processor mode node no:
3
 ** This is splfd2dmod version 1.0 multi processor mode node no:
4
 ** This is splfd2dmod version 1.0 multi processor mode node no:
5
 ** This is splfd2dmod version 1.0 multi processor mode node no:
6
 ** This is splfd2dmod version 1.0 multi processor mode node no:
7
 ** This is splfd2dmod version 1.0 multi processor mode node no:
8
 ** This is splfd2dmod version 1.0 multi processor mode node no:
9
 ** This is splfd2dmod version 1.0 multi processor mode node no:
10
 ** This is splfd2dmod version 1.0 multi processor mode node no:
11
 --- No of nodes:           100
* Server running!
forrtl: error (78): process killed (SIGTERM)
 Process#           0  is rebuilding the communicator
[0] FTMPI Error with communicator 0, error code -19 Known error not in list
forrtl: error (78): process killed (SIGTERM)
Error event set for channel 22
[15] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#          15  is rebuilding the communicator
 Process#          17  is rebuilding the communicator
[17] FTMPI Error with communicator 0, error code -19 Known error not in list
[21] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#          21  is rebuilding the communicator
[19] FTMPI Error with communicator 0, error code -19 Known error not in list
```

```
 Process#          19  is rebuilding the communicator
[14] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#          14  is rebuilding the communicator
 .
 <<<CUT>>>
 .
[4] FTMPI Error with communicator 0, error code -19 Known error not in list
[3] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#           3  is rebuilding the communicator
[11] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#          11  is rebuilding the communicator
[2] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#           2  is rebuilding the communicator
 Process#           5  is rebuilding the communicator
[5] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#          47  has rebuilt the communicator
 Process#          49  has rebuilt the communicator
 Process#          68  has rebuilt the communicator
 Process#          50  has rebuilt the communicator
 Process#          46  has rebuilt the communicator
 Process#          48  has rebuilt the communicator
 Process#          52  has rebuilt the communicator
 Process#          45  has rebuilt the communicator
 Process#          82  has rebuilt the communicator
 Process#          69  has rebuilt the communicator
 Process#          72  has rebuilt the communicator
 Process#          51  has rebuilt the communicator
 Process#          16  has rebuilt the communicator
 Process#          71  has rebuilt the communicator
 Process#          53  has rebuilt the communicator
 .
 <<<CUT>>>
 .
 Process#          56  has rebuilt the communicator
 Process#          57  has rebuilt the communicator
 Process#          60  has rebuilt the communicator
 Process#          61  has rebuilt the communicator
 Process#          62  has rebuilt the communicator
 Process#          87  has rebuilt the communicator
 Process#          64  has rebuilt the communicator
 Process#          63  has rebuilt the communicator
 Process#          21  has rebuilt the communicator
 Process#          22  has rebuilt the communicator
 Process#          55  has rebuilt the communicator
 Process#          54  has rebuilt the communicator
forrtl: error (78): process killed (SIGTERM)
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
 Process#           0  is rebuilding the communicator
[0] FTMPI Error with communicator 0, error code -19 Known error not in list
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
```

```
forrtl: error (78): process killed (SIGTERM)
Error event set for channel 40
 Process#          11  is rebuilding the communicator
[11] FTMPI Error with communicator 0, error code -19 Known error not in list
[41] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#          41  is rebuilding the communicator
[13] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#          13  is rebuilding the communicator
 Process#          16  is rebuilding the communicator
[16] FTMPI Error with communicator 0, error code -19 Known error not in list
[15] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#          15  is rebuilding the communicator
 Process#          30  is rebuilding the communicator
[30] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#          19  is rebuilding the communicator
[19] FTMPI Error with communicator 0, error code -19 Known error not in list
[18] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#          18  is rebuilding the communicator
 Process#          17  is rebuilding the communicator
 .
 <<<CUT>>>
 .
[5] FTMPI Error with communicator 0, error code -19 Known error not in list
[7] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#           7  is rebuilding the communicator
[4] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#           4  is rebuilding the communicator
[2] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#           2  is rebuilding the communicator
 Process#           9  is rebuilding the communicator
[9] FTMPI Error with communicator 0, error code -19 Known error not in list
[3] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#           3  is rebuilding the communicator
[1] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#           1  is rebuilding the communicator
[10] FTMPI Error with communicator 0, error code -19 Known error not in list
 Process#          10  is rebuilding the communicator
 Process#          35  has rebuilt the communicator
 Process#          13  has rebuilt the communicator
 Process#          68  has rebuilt the communicator
 Process#          57  has rebuilt the communicator
 Process#          67  has rebuilt the communicator
 Process#          66  has rebuilt the communicator
 Process#          90  has rebuilt the communicator
 Process#          44  has rebuilt the communicator
 Process#          47  has rebuilt the communicator
 Process#          70  has rebuilt the communicator
 Process#          25  has rebuilt the communicator
 Process#          72  has rebuilt the communicator
 Process#          14  has rebuilt the communicator
 .
 <<<CUT>>>
 .
 Process#          86  has rebuilt the communicator
 Process#          85  has rebuilt the communicator
 Process#          52  has rebuilt the communicator
 Process#          53  has rebuilt the communicator
 Process#          61  has rebuilt the communicator
```

```
 Process#           62  has rebuilt the communicator
 Process#           74  has rebuilt the communicator
 Process#           75  has rebuilt the communicator
 Process#           76  has rebuilt the communicator
 Process#           84  has rebuilt the communicator
Error event set for channel 3
Error event set for channel 4
Error event set for channel 3
Error event set for channel 3
Error event set for channel 3
Error event set for channel 3
Error event set for channel 3
Error event set for channel 4
Error event set for channel 3
Error event set for channel 3
Error event set for channel 4
Error event set for channel 3
Error event set for channel 3
Error event set for channel 4
Error event set for channel 3
Error event set for channel 4
Error event set for channel 3
Error event set for channel 3
Error event set for channel 4
Error event set for channel 3
Error event set for channel 3
Error event set for channel 4
Error event set for channel 3
Error event set for channel 3
Error event set for channel 4
Checking for a failure and doing a recovery if found.
Error event set for channel 4
Error event set for channel 4
Error event set for channel 4
SHRINK
Error event set for channel 3
Error event set for channel 3
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
.
<<<CUT>>>
.
```

```
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Callback on CON ERROR chan 3 con 20 Sock = 0
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Error event set for channel 3
Error event set for channel 3
Error event set for channel 3
Error event set for channel 3
Error event set for channel 3
Checking for a failure and doing a recovery if found.
Error event set for channel 3
Checking for a failure and doing a recovery if found.
SHRINK
SHRINK
SHRINK
Error event set for channel 4
SHRINK
SHRINK
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
```

```
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Callback on CON ERROR chan 3 con 80 Sock = 0
Callback on CON ERROR chan 4 con 82 Sock = 0
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Error event set for channel 3
Checking for a failure and doing a recovery if found.
Error event set for channel 4
Error event set for channel 4
Checking for a failure and doing a recovery if found.
Error event set for channel 3
Checking for a failure and doing a recovery if found.
Error event set for channel 3
Error event set for channel 4
SHRINK
SHRINK
Error event set for channel 4
SHRINK
NOP/RESET
NOP/RESET
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Performed a recovery
Performed a recovery
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
SHRINK
SHRINK
NOP/RESET
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Performed a recovery
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
.
<<<CUT>>>
.
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
```

```
Performed a recovery
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Error event set for channel 3
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
Error event set for channel 4
SHRINK
SHRINK
SHRINK
SHRINK
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
```

```
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
```

```
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Callback on CON ERROR chan 3 con 82 Sock = 0
Callback on CON ERROR chan 3 con 89 Sock = 0
Callback on CON ERROR chan 3 con 90 Sock = 0
Callback on CON ERROR chan 3 con 91 Sock = 0
Callback on CON ERROR chan 3 con 93 Sock = 0
Callback on CON ERROR chan 3 con 94 Sock = 0
Callback on CON ERROR chan 3 con 95 Sock = 0
Callback on CON ERROR chan 4 con 99 Sock = 0
Callback on CON ERROR chan 3 con 99 Sock = 0
Callback on CON ERROR chan 3 con 70 Sock = 0
Callback on CON ERROR chan 4 con 84 Sock = 0
Error event set for channel 3
Error event set for channel 3
Callback on CON ERROR chan 4 con 93 Sock = 0
Error event set for channel 3
Error event set for channel 3
Error event set for channel 3
Error event set for channel 4
Error event set for channel 3
Error event set for channel 3
Error event set for channel 4
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Checking for a failure and doing a recovery if found.
SHRINK
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Callback on CON ERROR chan 3 con 71 Sock = 0
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
SHRINK
SHRINK
NOP/RESET
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Performed a recovery
.
<<<CUT>>>
.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
```

```
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
Checking for a failure and doing a recovery if found.
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
SHRINK
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
NOP/RESET
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
DONE Checking for a failure and doing a recovery if found.
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Performed a recovery
Callback on CON ERROR chan 3 con 34 Sock = 0
Callback on CON ERROR chan 4 con 38 Sock = 0
Callback on CON ERROR chan 4 con 42 Sock = 0
Callback on CON ERROR chan 3 con 42 Sock = 0
Callback on CON ERROR chan 4 con 46 Sock = 0
Error event set for channel 3
Error event set for channel 4
Error event set for channel 4
Error event set for channel 3
Error event set for channel 4
 Sock = 44
Callback on CON ERROR chan 32 con 23 Sock = 48
Callback on CON ERROR chan 63 con 54 Sock = 78
Callback on CON ERROR chan 55 con 44 Sock = 70
```

```
Callback on CON ERROR chan 48 con 41 Sock = 56
Callback on CON ERROR chan 49 con 52 Sock = 58
Callback on CON ERROR chan 52 con 51 Sock = 62
Callback on CON ERROR chan 54 con 53 Sock = 76
Callback on CON ERROR chan 38 con 38 Sock = 51
Callback on CON ERROR chan 51 con 45 Sock = 65
Callback on CON ERROR chan 46 con 43 Sock = 49
Callback on CON ERROR chan 50 con 57 Sock = 60
Callback on CON ERROR chan 57 con 55 Sock = 63
Callback on CON ERROR chan 59 con 56 Sock = 69
```

Appendix D

Description of bibliography citations

[1]: The Top 500 list gave me the answer of how many processors there are on the largest cluster in the world as of November 2006.

[2]: General information on MPI. Gave me information on the different versions of MPICH. It also provides the MPI Standard with the MPI Forum. Here I also have a list of all the MPI functions available.

[3]: Provides a detailed description of MPI, and better explained than any other sources I found. The article is continuously in development, and makes more issues come clear, at least for me.

[4]: This article describes the OSI model, which I knew in detail from before, so this article was not so important to me. It is cited because I wanted to explain what the OSI model is, since I used the abbreviation in my thesis.

[5]: This article describes the Non-Uniform Memory Architecture (NUMA). I used this abbreviation in my thesis, and wanted to show what this means, even though it is not directly important to my thesis.

[6]: This article has a good description of the MPICH library. Even though I knew something about this from before, I had to cite this, because it shows what MPICH is. It also involves several topics as describing how it is a high-performance portable implementation.

[7]: This article gives me a detailed description on what LAM/MPI really is, and more than I needed to know, to give a description of it in my thesis.

[8]: This website gave me the useful information that LAM/MPI is not being developed anymore, the team has migrated to the new project, Open MPI.

[9]: This article gave me general and detailed knowledge on Open MPI, which I used to describe the project in my thesis. I only needed general knowledge, so this was good and enough for my thesis.

[10]: An excellent article describing FT-MPI. Actually, the entire article was useful in letting me know how the system works, and how to write a fault-tolerant program, using the FT-MPI system. This article is widely used when describing FT-MPI in my thesis.

[11]: FT-MPI is built open Harness, so a description is provided by this citation. The article is not so important for my thesis, because I do not use it explicitly. Anyway, because Harness is on top of FT-MPI, it must be shown what it is.

[12]: My thesis describes FEMPI, and this article has been a good inspiration for providing this description. I mostly used chapter 3, Design overview of FEMPI, because it gave the most relevant information.

[13]: This article describes MPICH-V very detailed and gives a good insight in what the system is capable of. For me, it was important to notice all the different qualities of the different implementations they have developed.

[14]: The MPICH-V website mostly gave me a good and short description on the different implementations of MPICH-V, which was inspiring for writing about them in the description of MPICH-V in my thesis.

[15]: This is cited because I had to show what The Chandy Lamport algorithm is, because I mentioned it in my thesis regarding the VCL implementation of MPICH-V.

[16]: The Open MPI website has a Frequently Asked Questions page, where I found that Fault-tolerance not yet is implemented in Open MPI. I described what was being stated here. I also found some general information here to use with my description of Open MPI.

[17]: This article describes how to make a non-transparent fault-tolerant MPI application. I studied every aspect of this article to mainly understand what they wanted to tell. It is also well described in my thesis, where I provide their four approaches.

[18]: This article describes techniques used for fault-tolerant matrix operations on hypercube multiprocessors. It gave me ideas for going deeper in the processor's hardware and understand that there is more to the fault-tolerance topic than only software implementations.

Appendix E

Electronic appendix

Included in this thesis, there is an electronic appendix as a zip file. Contents of this file are the following:

- Source code of Simulator original: simulator_original.c
- Source code of Simulator FT-MPI: simulator_ftmpi.c
- Source code of Simulator Error handlers - master: simulator_errhandler_master.c
- Source code of Simulator Error handlers - slave: simulator_errhandler_slave.c
- Source code of Simulator used for measuring wall time of communicator rebuilding: simulator_ftmpi_time.c
- Output listings from the test of test application, small cluster: testappoutput_small.log
- Output listings from the test of test application, large cluster: testappoutput_large.log
- Example screen shot from the test application: example_output_shotgather_su.png
- Example zoom screen shot from the test application: example_zoom_output_shotgather_su.png