Investigation of Fault Tolerance in MPI Applications

Paul Woodhams

August 16, 2008

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2008
Abstract

Standard MPI implementations are not fault tolerant and the default behaviour of MPI when a fault is encountered is to crash. Fault tolerance in MPI applications is becoming more and more important as the world’s supercomputers get bigger. As codes run on more processes for longer, the chances of them encountering a fault and the cost of that fault both become greater. This project aims to investigate the development of fault tolerant MPI applications. The project succeeded in designing an interface for a fault tolerant library. This involved informing the user of a fault via the return calls from the MPI communications functions as well as supplying a communicator containing the processes which were still available. After this a fault simulation library which provided this interface was designed and implemented. The implementation involved the generation of random errors and notification of errors to other processes via a unique global state variable (stored on rank 0) accessed using MPI single-sided communications. In addition, it was necessary to ensure that all outstanding sends to and receives from a dead process were matched to avoid deadlock. This was achieved by having each process record its current messaging state in a local memory window that could be accessed remotely by other processes. Finally the use of this fault simulation library was demonstrated with disk checkpointing versions of taskfarm and image processing applications. Fault tolerance was added to these applications using the interface as provided by the fault simulation library to detect an error, and disk checkpointing as the method of recovery. This resulted in parallel programs that produced the same results regardless of how many processes failed during execution. The taskfarm works consistently but the image processing code does encounter subtle errors that result in deadlock: these errors are documented as part of this report. The interface designed for the library is clearly easy to use and functional as shown by the completion of these fault tolerant applications. Despite the minor problems with the image processing code, the project was a success.
# Contents

Chapter 1 Introduction ................................................................................................... 1

Chapter 2 Background .................................................................................................... 3
  2.1 Fault Tolerant MPI Implementations ..................................................................... 3
     2.1.1 Fault Detection .............................................................................................. 3
     2.1.2 Recovery ....................................................................................................... 4
     2.1.3 Message Completion ...................................................................................... 5
     2.1.4 Communicators ............................................................................................. 5
     2.1.5 Collective Communications .......................................................................... 6
  2.2 Schemes for Fault Tolerant Applications ............................................................. 6
     2.2.1 Disk Checkpoint ........................................................................................... 6
     2.2.2 Data Redundancy ........................................................................................ 7
     2.2.3 Task List ....................................................................................................... 7
     2.2.4 Intercommunicators ..................................................................................... 8

Chapter 3 MPI-FS Library Implementation ................................................................ 9
  3.1 Aims ..................................................................................................................... 9
  3.2 Implementation Synopsis ..................................................................................... 10
  3.3 Design Questions ................................................................................................. 11
  3.4 Design Criteria ................................................................................................... 15
     3.4.1 Criteria ......................................................................................................... 16
     3.4.2 Discussion .................................................................................................... 16
  3.5 Implementation ................................................................................................... 17
     3.5.1 Working Communicator .............................................................................. 17
Chapter 6 Regular Domain Decompositions .............................................................. 54
  6.1 Motivation .............................................................................................................. 54
  6.2 Framework Application ......................................................................................... 55
    6.2.1 Synchronous Communications ...................................................................... 55
    6.2.2 Dynamic .......................................................................................................... 56
    6.2.3 Derived Datatypes .......................................................................................... 59
    6.2.4 Collective Communications ........................................................................... 60
    6.2.5 Modularisation ............................................................................................... 60
  6.3 Disk Checkpointing ............................................................................................... 61
    6.3.1 Start-Up ........................................................................................................... 63
    6.3.2 Creating Checkpoints ..................................................................................... 64
    6.3.3 Loading Checkpoints ...................................................................................... 64
    6.3.4 Catching Errors ............................................................................................... 65
    6.3.5 Restarting ........................................................................................................ 65
  6.4 Results .................................................................................................................... 66
  6.5 Conclusion ............................................................................................................. 71
Chapter 7 Conclusions and Further Work ................................................................. 73
References ....................................................................................................................... 75
List of Tables

Table 1: Status Window Contents ................................................................................... 24
List of Figures

Figure 1: Intercommunicators................................................................. 8
Figure 2: Proposed Library Functionality ........................................... 10
Figure 3: MPIFS Intercepts Selected MPI Routine Calls.................... 12
Figure 4: Profiling Interface Code Example ....................................... 12
Figure 5: Split Communicators.......................................................... 13
Figure 6: Locking and Unlocking RMA Windows.............................. 15
Figure 7: Send Flow Diagram............................................................. 19
Figure 8: Receive Flow Diagram........................................................ 21
Figure 9: Global Error Flag ................................................................. 22
Figure 10: Standard Status Window Usage – Local Windows .......... 24
Figure 11: Clean-Up Routine Flow Diagram....................................... 25
Figure 12: Clean-Up Status Window Usage – Remote Windows........ 26
Figure 13: Split Communicator between Alive and Dead................... 28
Figure 14: Fault Propagation Implementation .................................... 30
Figure 15: Message Round a Ring ...................................................... 33
Figure 16: Synchronous Message Round a Ring.................................. 33
Figure 17: Generation of a Single Fault ............................................. 34
Figure 18: Generation of Multiple Simultaneous Faults.................... 35
Figure 19: Manual Reduction ............................................................. 35
Figure 20: Manual Reduction with and without Clean-Up

Figure 21: Split Communicator Changing Ranks............................... 39
Chapter 1

Introduction

Parallel processing machines are getting bigger and bigger. As the scale of these machines increases the importance of fault tolerance becomes much greater. The number of CPUs in these machines can be anywhere from the thousands to tens of thousands. With so many CPUs the probability of one or more of them failing during a computation, even with the probability of a single process failing being very small, becomes significant. For example, if the lifetime of a processor is five years and you are running your application on 10,000 processors then you can expect a fault due to a processor failure on average around every 4 hours. The trend towards larger machines looks set to continue and, with the advent of multi-core, fault tolerance will only become ever more critical.

A normal implementation of MPI does not handle faults in a very effective manor. The default behaviour of an MPI application when a fault is encountered is for the application to simply terminate. Normally no clean exit is possible and the user is left with little or no idea as to what the problem was. Without the possibility of a safe and clean closing down of the application then any results from the application cannot be trusted. This results in the application having to be run again from the beginning. If the application were small then this would not be too much of a problem as the application would not take long to run again and the chance of it encountering a fault during the run is small in the first place.

However, most applications being run on the most powerful machines run on many processes for many hours, if not days. The fact that more processes are being used coupled with the fact that they are being used for longer means they are more likely to encounter a process failure during the computation than smaller programs. A process failure on one of these large applications has a high cost in terms of lost machine time. In these applications fault tolerance is a key issue and this project will aim to investigate possible fault tolerance methods.

The project will begin by setting itself in context. Various attempts at making a fault tolerant implementation of MPI have been attempted. These will be reviewed here and there impact on the possible direction of the project assessed. In this background section details of possible fault tolerance methods which could be applied to user applications will also be looked at.
With the project context established the design and implementation of a fault simulation library will be looked at. It was not feasible to develop a genuinely fault tolerant version of MPI for this project and there is not one present on any of the target machines. As such it was decided that a library to simulate random hardware faults would be developed. By designing and implementing this as part of the project it allowed for the interface between any fault tolerant MPI implementation and a user application to be investigated. What a fault tolerant MPI could provide to the user and what the user may actually want for building fault tolerant applications could be very different. This project aims for the fault simulation library to provide the ideal interface for a user application to become fault tolerant while still being feasible for applications developers. Also, by developing a fault simulation library rather than an actual fault tolerant implementation of MPI it allows control over when faults occur. This can be very useful for looking at specific problems and extreme cases.

Finally, two sample applications will be looked at and fault tolerance methods for them investigated. The two target applications are a simple taskfarm and an image processing application. It is very easy to find implementations of these applications and the fault tolerance methods applied to both of them, especially the image processing code, have significance for many other applications. By linking these with the fault simulation library developed earlier in the project and using the interface provided it was hoped to have these applications find the simulated fault and respond in an appropriate way to actually complete the computation. The details of the framework applications, the fault tolerance methods applied to them as well as some results from them are included in this report.
Chapter 2

Background

Following from the literature review carried out as part of the project preparation module some more specific details on some areas are given here. Firstly some discussions on current implementations of MPI which include fault tolerance are presented. Secondly, some details of possible fault tolerance schemes are presented.

2.1 Fault Tolerant MPI Implementations

There are several papers available which document the development of a fault tolerant version of MPI, most notably [1] and [2]. By looking at these and how they achieve fault tolerance within an actual MPI implementation it should give some insight into what the fault simulation library to be developed as part of this project should produce.

2.1.1 Fault Detection

In [1] failures are detected using a UNIX script. This script periodically checks for the existence of all the processes. This script passes its results to an “observer” process which is responsible triggering the recovery process. Should a failure be detected then the observer is responsible for informing all other processes that a fault has occurred. The observer is not involved with the normal computation, only the detection and response to faults.

The other processes are notified of a fault by having set up a non-blocking receive at the beginning of their computation. When the observer discovers a fault it sends a message indicating that a fault has occurred to all the remaining processes. By having each process regularly test the initial non-blocking receive, processes become aware that a fault has occurred.

The method proposed in [1] does however require the assumption that the process which is the observer will not fail. Were the observer process to fail, then any application using this MPI implementation would also fail.

Unfortunately few details on how errors are detected are given as part of [2]. There is a “failure handler” which receives notifications of errors from the communications libraries as well as the operating system. It is this failure handler which is then
responsible for notifying all other processes of the error by “injecting notify messages into the send message queues ahead of user level messages”.

It seems that both implementations use a routine or process which is responsible for dealing with the detection of a fault (observer in [1] and failure handler in [2]). Once this process has found the fault it is then responsible for informing all other processes that a fault has occurred.

To apply this to the idea of simulating faults in a library is not straightforward. Clearly, detecting an error by polling the operating system will not be necessary as the faults will be generated in the library. A method for having all processes become aware of this fault is however clearly necessary. The method used for [1] where a non-blocking receive is setup and all processes are sent a message by the observer when a fault is detected could work. However, there is the problem of how would the observer become aware that a process has failed.

To overcome this, a method where the responsibility of notifying when a fault occurs is taken away from an observer and given to the faulting process could be used. In this method a global error flag would replace the observer as the point of reference for all processes. When a process generates a fault it would then set the error flag. To go with this, instead of each process regularly testing an initial non-blocking send they could regularly test the value of this flag and respond appropriately.

2.1.2 Recovery

The recovery attempted in [1] is done by spawning a new process to replace the one that died. Once this process has been spawned all the other processes then re-send all the appropriate messages to the replacement process. Basically, the replacement process will run the application from the start until it has caught up with all of the other processes. In such a way the application doesn’t ever become aware of a fault and the fault tolerance is all within the MPI implementation. This seems the ideal solution but for long programs running one process all the way from the start to catch up with the others will be time consuming and a large overhead to fault tolerance.

This method does not seem very appropriate for this project. The main aim of the project is to look at fault tolerance methods for MPI applications. As such a library which takes all fault tolerance responsibility away from the applications programmer would not be useful. The issues covered in the following sections detailing the implementation used in [1] are however relevant for the practicalities of certain elements of the library.

The recovery implementation of [2] is of great interest for this project. The user of the implementation becomes aware of a fault through the return values of MPI calls. If the return value is MPI_ERR_OTHER this indicates that a new fault has occurred. More information on the nature and specifics of the error are available through the cached attributes interface in MPI. This method of using MPI call return values for informing an application of the fact a fault has occurred shall be used as part of the library developed for this project.
Another interesting feature of [2] is the actual fault recovery carried out. In [2] the errors are associated with communicators. Any fault in a communicator leads to all processes in that communicator being made aware of it. If the error is a process failing then all the processes in all the communicators which the failed process is part of are made aware of the fault. Any communicators affected by the fault (communication failed in it or failed process was part of it) must be updated.

The updating of a communicator is done by rebuilding the communicator using a modified version of MPI_Comm_{create, split or dup}. The idea of rebuilding a communicator to be made up of only available processes is very useful for the library. The method of most interest is the idea of “shrinking” a communicator. This method removes the failed process from the communicator and in the process reduces the communicator’s size. Having the fault simulation library return a communicator which contains only valid processes after a fault would be very useful for applications programmers.

2.1.3 Message Completion

In [1] there are two methods detailed for dealing with the resending of messages. As this project will not take the approach of resending every message after a fault to a replacement process this would not seem immediately relevant. However, the fact that all the messages will not be resent does lead to an issue.

The issue comes from the case where a process issues a blocking communication for which the target/source is the now failed process. With the method of having a replacement process resend all communications of the dead process this would not be an issue as the communication would eventually be matched. However, with this removed the process in the blocking communication will stay there forever if no matching communication is issued. Clearly some method for having a faulting process clear all outstanding communications is necessary for the fault simulation library to work correctly. This will be possible as part of the fault simulation library due to the fact a process is not actually unavailable and hence could clear the blocking message before “dying”.

2.1.4 Communicators

Both [1] and [2] deal with the idea of a process failing leaving a “hole” in the communicator and hence making it invalid. An invalid communicator cannot be used and hence this is a major issue. Several methods for dealing with this “hole” in the communicator to allow the successful recovery are described in the papers.

Fortunately, for this project the details are not relevant. When a fault is generated in the fault simulation library the “dead” process will still actually be available. This means that the communicator it is part of will not actually be invalid. This allows for the fault recovery step described at the end of section 2.1.2 where a split communicator is used to shrink the communicator to be called on the original communicator despite it being a global routine. As such the failed process could be split off and the communicator containing only the valid processes returned the users application.
2.1.5 Collective Communications

Both [1] and [2] deal with the idea of implementing collective operations in a fault tolerant implementation of MPI. In [1] the collectives are implemented via explicit calls to the appropriate point-to-point communications with dynamic routing tables. This was done as collective operations are called over a whole communicator and with the communicator being invalid they would not complete successfully without being implemented differently.

In [2] limited details on what was done to implement collective communications is given but it does state that “care has been taken in implementing the collective communications so that if an error occurs during an operation, the result of the operation will be the same as if there were no error, else the operation is aborted”.

The idea expressed in [2] of ensuring collective communications complete exactly even with an error or abort is vital. This will be included in the library should it reach the stage of developing collective communications. It will most likely be accomplished by returning an error in the same way as for point-to-point communications were an error detected. In such a way applications programmers can check the return calls of collective operations in the same way as they would point-to-point to find faults.

The issues raised in [1] are not immediately relevant, but as the routine is collective it may well suffer from the same issue of deadlock as described in 2.1.3. Say one process has reached and called the collective operation. If any other process fails before calling that routine the first process is now blocked in the collective operation. To overcome this some method of clearing collective operations by having all processes issue them even after “dying” would be needed. A simpler method may be to overwrite the collective communications with point-to-point equivalents and hence any system put in place to match outstanding point-to-point communications would also clear the collective operations.

2.2 Schemes for Fault Tolerant Applications

Many different schemes for making fault tolerant applications are available. The details of many of these schemes were looked at in the project preparation report. In this section selected details for fault tolerance methods to be looked at as part of this project are repeated with minor modifications from those previously presented in the project preparation report. They are included here again for completeness.

2.2.1 Disk Checkpoint

The simplest and most common method for making a fault tolerant application is to periodically create a checkpoint. Disk checkpointing can be used for most applications and as part of this project it will be used with a task farm and domain decomposition application. Disk checkpointing involves each process writing out its section of the dataset to disk. If a fault is ever detected then the latest checkpoint is loaded. The application redistributes the data between the surviving processes and the computation continues. All of this code must be written by the application developer.
When this method is being used the applications programmer must consider how often to checkpoint. This depends on many factors but most important is the time it takes to create a checkpoint. For a detailed description of this method and some of the mathematics behind it please see [3] or [4]. Creating a checkpoint is a time consuming process as generally all of the data is being written to main memory and the amount of data can be substantial. It is critical to implement this routine as efficiently as possible using things such as MPI-IO so that checkpointing is a feasible scheme.

Redistributing the data when a fault is found can be a serious overhead for checkpointing. It is a similar routine to that which must be used to distribute the data at the start of the program. Setup is generally a slow and often serial process for many MPI applications and the problems apply when redistributing data. Again MPI-IO should be used to make this as fast as possible. Redistributing data when a fault occurs does however ensure that there are no load balance issues.

2.2.2 Data Redundancy

Data redundancy is a scheme similar to disk checkpointing. It attempts to reduce the time it takes to make a copy of the data. To do this the scheme does not create a disk checkpoint in main memory, but instead each process periodically swaps data with its neighbouring processes. This is similar to a halo swap except that all of the data from a process is sent to and then stored on its neighbours.

As there is no full checkpoint created it makes restarting the application a more complicated and time consuming process than disk checkpointing. There are two options when a fault is detected. First the processes could create a disk checkpoint as previously described but with one process filling the hole left by the “dead” process. The data would then be redistributed as before and the computation continues. The second option is for the process which has the swapped data from the “dead” process to simply take up the work. Clearly this leads to the problem being less well load balanced. If however the application is nearing termination then this could be better than incurring the overhead of creating a checkpoint and fully redistributing the data.

2.2.3 Task List

A task list approach is only suitable for task farm applications. In a task farm the master simply provides jobs to each of a pool of worker processes. When a worker completes a job it submits the result back to the master and a new job is provided. In this scheme the master simply keeps a list of which task each worker currently has. If a fault is detected then the job which that worker had can be recovered from the list and added back into the list of jobs yet to be completed.

Clearly this scheme is dependant on the master process always being available. If a fault were to occur on the master then the application would fail just like an application with no fault tolerance scheme. The scheme is still relevant and useful though as the application only fails if the master faults while the original application without fault tolerance would fail if any process faults. Thus the probability of the application failing has been reduced significantly.
2.2.4 Intercommunicators

Intercommunicators are provided as part of the MPI-2 standard. They aim to allow the developer to manage the processes being used within an application dynamically, at runtime. As part of this a process is able to spawn other processes. In doing so an intercommunicator is formed. This is the communicator which allows communication between the spawning process and those processes which it has spawned. The spawned processes also have their own communicator to communicate with each other. In general intercommunicators are aimed at allowing a master to spawn a pool of processes which can then do a section of work. In this scheme however multiple intercommunicators which link the master with just a single spawned process will be formed. This is shown diagrammatically below.

![Figure 1: Intercommunicators](image)

A scheme involving intercommunicators is again best suited to a task farm application. At the start of the application the master spawns all of the workers with each worker in a separate intercommunicator. The application then runs like any other task farm with the master sending out jobs and receiving the results from the workers.

Fault tolerance is handled as for the task list approach. The master keeps a list of which jobs are with which worker. Now if a process becomes unavailable though then the master marks the relevant intercommunicator as invalid as well as putting its job back on the stack of waiting jobs. The advantage of this approach over the task list approach is that it gives MPI a better chance of catching a process failing and there are no invalid communicators being used. Details of this scheme can be found in [3].
Chapter 3

MPI-FS Library Implementation

In this section the details of the MPI fault simulation library, MPI-FS, are provided. Included are the aims of the library, the design criteria set at the start of the development, a discussion of the pros and cons of an alternative solution and details of the actual implementation of the library. This stage was necessary to investigate the best interface to a fault tolerant library for applications programmers and also to allow the investigation of fault tolerance methods in applications.

3.1 Aims

The main aim of the library was to generate random faults which simulated a process failing during the running of an application code. The faults to be simulated were to be of a processor becoming unavailable during the computation due to some form of hardware fault.

The library also needed to provide a useful interface to the applications programmer. As such the library was also emulating the behaviour of a real fault-tolerant library. This is needed so that when a fault is generated the user’s application can do some form of fault recovery, either to continue the computation or exit cleanly. There are two main things that are needed for an application to become fault tolerant. The first of these is that all surviving processes must become aware of the fault so that they can respond in the correct manner. Secondly, the library must provide a communicator which only contains the “alive” processes after a fault has occurred. This is needed because MPI applications rely on the use of a valid communicator which contains only processes which are available for communications calls.

The diagram below shows the general operation and interface that the library was aimed to have.
Figure 2: Proposed Library Functionality

3.2 Implementation Synopsis

This section aims to give a brief outline of the implementation which will be used to develop the fault simulation library. More details on how this implementation was decided on and how the actual implementation was carried out are shown in the following sections.

The profiling interface will be used to intercept communications calls to the MPI library from the user’s application. Interception will allow for the generation of faults and the handling of faults when they occur to be dealt with in the library. This means that the details of how the interface is realised are kept away from the user.

Notification to all processes of when an error has occurred will be accomplished through use of a global error flag. All processes will be able to read and write the value
of this error flag. To handle the tracking of what communications each process is currently undertaking a local state table will also be used. This will only be written to locally but can be read by anyone. Both of the global error flag and local state table along with access to them will be implemented using MPI-2 single-sided communications.

3.3 Design Questions

The aims and diagram presented in section 3.1, although appearing simple, do raise some complicated issues. These issues stem from the fact that the process is not actually unavailable but it is just acting as if it is “dead”. The three key issues which must be addressed are:

- Deciding when and if a process is to simulate being “dead”
- A “dead” process must be split off and not interact anymore
- Some method to create a new valid communicator will be needed

These issues lead to a series of questions which then must be asked, and these questions in turn generate more questions. Below these questions and the answers to them are presented.

- How are the faults going to be generated?

The generating of faults would occur during the sending or receiving of messages. It was decided early on that faults should only be generated when sending messages. Generating faults on both sends and receives within the library would have made the implementation more complex. An extension to the project could be to add fault generation on receives as well or possibly on any MPI call.

To generate the fault the communications calls from the user’s application would need to be intercepted. The method used to intercept communication calls was the MPI profiling interface. The diagram below shows how the library works with the user’s application as well as the original MPI library.
Selected calls to MPI communications routines are intercepted in the fault simulation library, MPI-FS. From here some work to generate and handle faults is done and then the original MPI routine can be called, this is done by issuing the communication call as normal but with a ‘P’ placed at the start of the routine name. The extract below shows a simple example of how the profiling interface works.

```c
int MPI_Ssend(void* buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm) {
    int ret,
    //do something extra
    printf("Within the MPI_Ssend routine!\n");
    //call the actual MPI routine
    ret = PMPI_Ssend(buf, count, datatype, dest, tag, comm);
    return ret;
}
```

Selected calls to MPI communications routines are intercepted in the fault simulation library, MPI-FS. From here some work to generate and handle faults is done and then the original MPI routine can be called, this is done by issuing the communication call as normal but with a ‘P’ placed at the start of the routine name. The extract below shows a simple example of how the profiling interface works.

```
int MPI_Ssend(void* buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm) {
    int ret,
    //do something extra
    printf("Within the MPI_Ssend routine!\n");
    //call the actual MPI routine
    ret = PMPI_Ssend(buf, count, datatype, dest, tag, comm);
    return ret;
}
```

Figure 3: MPIFS Intercepts Selected MPI Routine Calls

In this example, when the user application calls MPI_Ssend this call will be intercepted by the profiling interface. From here a message to screen is printed out and then the actual communication call from the MPI library is called.

By doing this within the fault simulation library it allows for faults to be generated and the intricacies involved with informing the user application plus providing a useful interface for fault tolerance to be hidden from the user. The user is made aware of the fault by the communication call return value and then knows to switch to the new valid communicator provided. The exact method for generating a fault used in the library is described in the next chapter.

- What does a process do when it is “dead”?

Once a process is “dead” it should not be actively involved with the computation anymore. Due to the fact this project is actually only simulating faults and hence the
process is still actually available it can be used to call collective operations which may be necessary to provide a suitable interface for applications programmers. Once this has been completed the “dead” processes must finalize and then exit. To ensure a clean exit to the application all processes must call MPI_Finalize. For this reason it is important that even processes that are simulating being “dead” call it.

- After a fault can MPI_COMM_WORLD be used or is a new communicator needed?

With one of the processors “dead” MPI_COMM_WORLD becomes invalid. If the project were to implement a fault tolerant version of MPI then we would like to redefine MPI_COMM_WORLD so that it contains only alive processors and then continue to use it after the fault. Unfortunately the project will only be using the MPI profiling interface and hence redefining MPI_COMM_WORLD is not possible. In this case a new communicator will be needed.

- How can the new valid communicator be created after a fault?

To create a new communicator the routine MPI_Comm_split will be used. This works by using a variable called the color. Each process has its color set and will be put into a group with all other processes with their color set to the same. This is best illustrated with a diagram as shown below.

![Figure 5: Split Communicators](image)

By using this on the original communicator with the color set to whether the process is “alive” or “dead” the processes will be grouped correctly. This will provide the valid communicator of “alive” processes as needed. In the diagram above the rank shown is that which the process had in the original communicator and it remains the same even after the split. In reality processes would be re-ordered within their group of the split communicator.
- How would all of the processes become aware of when one process is “dead”?

As split communicators are collective operations all processes will have to become aware of the fault and call the appropriate routines. This is also necessary so as to provide the interface to applications programmers as described earlier. For all processes to become aware of when a fault appears on any of the other processes, an area of global memory will be used. This space would be used to store the current status of the system. Access to this area would be needed by all processes and they may want to access it at anytime. However, only a single process should be able to gain access to this global memory at a time.

An alternative possibility for informing all processes that a fault has occurred is to use communication time-outs. In this method faults propagate through the system until all the processes are aware of the error. This method was not employed for this project but details of how it would work are given in section 3.6.

- How could this global memory area be implemented?

As the applications being looked at are all MPI then it must be an MPI feature which is used to implement the global memory area. The solution is to use the single-sided communications features of MPI-2. This involves declaring an area of memory as being in an RMA (remote memory access) window. Once this is done it allows the other processes to access the data in the window using the appropriate RMA calls.

- What sort of access will be needed for these RMA windows?

The access pattern to the data stored in the RMA windows is not known at the start. This is due to the random nature of the faults and also the fact that a range of different applications which all have different communications patterns will be targeted by the library. This lends itself to using the lock and unlock routines combined with put and get operations to retrieve or modify the data. The diagram below shows how lock and unlock would work.
In the diagram above process 0 first calls the lock routine for the RMA window on process 1. When process 0 has obtained the lock it can then carry out any RMA calls that it needs to on the RMA window declared on process 1. Once process 0 has finished all of its RMA calls it then releases the lock for the RMA window. Process 2 can only obtain the lock for the RMA window of process 1 after process 0 has finished its RMA calls and released the lock. Once the lock is available process 2 can obtain it and then carry out its own RMA calls on the window of process 1 before releasing the lock so that other processes can gain access to the RMA window of process 1. If a process calls the lock routine while another process has obtained the lock then that process will block and wait for its turn to obtain the lock before carrying out any RMA calls.

### 3.4 Design Criteria

When work began on the project the criteria for the library were fairly vague, simply to “use the MPI profiling interface to implement a version of the MPI library that simulates random hardware faults.” This reflected the fact that it was thought of as a simple step which was necessary before the development of fault tolerant applications could begin.

Upon beginning to investigate possible methods for implementing the library it was soon realised that this step would be far more time consuming and involved than previously expected. The issues raised during this phase have been described in the previous section. It would be simple to simulate a fault on a send or receive, but much more difficult to let all other processes know about it. With these issues looked at the first step in developing the library was to define some more stringent design criteria. These are presented below and are followed by a brief discussion on some of the points to make it clear why they were included.
3.4.1 Criteria
- Simulate single or multiple processes failing during computation
- Work with any applications using synchronous point-to-point communications
- Easy to modify the method for determining when a fault is generated
- Any “dead” processes cannot be involved with further computation
- All processes must be informed of a fault as soon as possible after it occurs
- Return an error to calling application if a fault has been generated
- Provide a valid communicator of the alive processes to the application
- Limit the additional communication costs associated with the library
- Use a limited amount of computational resources such as memory
- Library structure allows for easy extension or improvement

3.4.2 Discussion
The library aims to provide all of the things that an application programmer would need to develop a fault tolerant application. As such the library creating a communicator containing all of the processes which are still alive was a key feature. Without a valid communicator to work with the user’s application has very few fault tolerance methods to choose from. In a similar vein returning an error to the application when a communication call fails or an error is found was vital. Without such a return value the user application could not respond to and recover from a fault using the returned communicator.

The decision that it was necessary for all processes to become aware of a fault and synchronise in some way to deal with the fault was also to make the library easier to use by applications developers. If some processes were able to continue computing while others had discovered the fault this would make recovering from the fault by loading a checkpoint or another such method more difficult within the application code. The alternative implementation to that used which is presented in section 3.6 suffers from this disadvantage.

The criteria also attempt to limit the functionality required in the library. To make the library suitable for use with every possible MPI call would have involved far too much work for a master’s project. As such it was decided early on to use only synchronous point-to-point communications. Synchronous operations were chosen as the basis of the library as they ensure that a process is only ever involved in one operation at once. If time allowed then the addition of simple collective communications such as a reduction or broadcast was planned to increase the range of applications which could be targeted. However, if there was not time to develop this functionality into the library then hand-coded versions using synchronous point-to-point communications would be used.
instead. Therefore they would also automatically inherit the implemented fault
tolerance mechanisms.

3.5 Implementation

3.5.1 Working Communicator

The working communicator is a vital element of the MPI-FS library. It is the
communicator which must be used for all communications when using the MPI-FS
library. The user should set it to the communicator they intend to use for their
application. This can be MPI_COMM_WORLD if this is what the user requires.

The working communicator must be set so that MPI-FS can create the RMA windows
correctly. An RMA window must be assigned to a communicator. This declaration and
assignment is done as part of the setup done in MPI_Init described in the next section.
This means that the working communicator must be set before the user’s application
calls MPI_Init.

After a fault the library does not automatically set the working communicator to the
split communicator of still “alive” processes. This is left to the user and must be done
on all still “alive” processes once a fault has been discovered. The creating of new
windows after a fault is handled in the error handler routine of the MPI-FS library as
described later so once a fault has been recovered from it is safe to use the new
communicator of “alive” processes immediately.

3.5.2 Setup

When MPI_Init is called from the user’s application it is intercepted by the MPI-FS
library. As well as calling the genuine MPI version of MPI_Init the library also carries
out some other tasks to initialise the library for use. Details of the steps taken within the
intercepted MPI_Init routine are given in this section.

To allow for the library to be run with different settings but without the user needing to
recompile their code the settings for the library are loaded from a text file. This file can
be edited by the user and the settings which can be set from it are the maximum number
of restarts to happen during the computation and some debugging levels which decide
what messages, if any, the library sends to the user. For more details of these please see
the relevant sections of the submitted code.

The implementation will be using RMA windows to allow for processes to have access
to a global error flag as well as a status window which contains information on what
communication a process is currently attempting to complete. The use of these
windows will be discussed later in sections 3.5.5 and 3.5.8. Within MPI_Init however
these windows need to be declared. RMA windows need to be attached to a
communicator and it is for this reason that the working communicator as described in
the previous section must have been set appropriately before calling MPI_Init. With
this done memory is assigned for each of the windows using MPI_Alloc_mem and then
the windows published using MPI_Win_create so that all processes become aware of them.

The final step carried out in the MPI_Init routine is to seed the random number generator that will be used to determine whether to simulate a fault or not. To ensure that each process is seeded with a different number the random number generator is passed the processes rank in MPI_COMM_WORLD multiplied by current system time converted into an integer. In this way each run of the library will get different seed values and also each processor should also have a different seed value. This helps to ensure that several processes do not get the same seed value and hence generate a fault at a similar point within the application. However, this method can be easily changed should a different implementation produce the user’s desired functionality.

3.5.3 Synchronous Sends

When the routine MPI_Ssend is called within the application it is intercepted by the MPI-FS library and the following scheme is used instead of the MPI_Ssend just completing normally. The flowchart below shows the stages that are gone through within the MPI_Ssend routine in the MPI-FS library. Some of these stages may look simple but some, including use of the error flag, clean-up and error handler, are quite complex. Each of these stages is described in greater detail within the following sections.
The first stage when issuing an MPI_Ssend is to check the global error flag. This involves locking the window containing it so that no other process can change its value while the current process is reading it. If the flag is set it indicates that an error has occurred on a process and the clean-up routine (to match outstanding communications) and the error handler (to create the communicator of “alive” processes) are called. More details on these routines are given in the relevant section of this report.

If no error is found then the next step is to check whether or not the calling process itself should fault and generate an error. The exact method for deciding this is described in detail later. If the process is to “die” then it first sets the error flag, locking it while doing so, to inform other processes that an error has occurred. The process will then update its own status to indicate that it is dead and enter the clean-up and error handler routines as previously mentioned.
Should the process neither find a fault nor decide to simulate a fault itself then the communication will actually be issued. Before the MPI_Ssend routine is actually issued though the process sets its own status to show what communication it is doing, to who as well as other details to do with the message. This information is stored in the status window as later described and it is this information that is used as part of the clean-up to determine if there are any outstanding communications to be completed. When setting the status information the processes status window will have been locked. With the status set the send is actually issued. Once this completes the process then resets its status window to show that it is not currently carrying out any communications.

Finally the process will check the global error flag once more, again locking it while doing so. If it is set then the clean-up and error handler routines will be called as before. If it is not set then the communication has completed successfully and MPI_SUCCESS is returned to the application to indicate the successful completion. Should the communication have to call clean-up and the error handler then it will return an error to the application. An error is indicated by a return value of MPI_FAULT.

MPI_FAULT is a defined value in the MPI-FS library that is available to the applications programmer. By having the value of MPI_FAULT available to the applications programmer it allows them to test the return value of a communications call and if it is equal to MPI_FAULT they know that a recoverable fault has occurred and that a valid communicator to continue computation with has been created. This allows them to go through an appropriate recovery routine. The user’s application must now check the return code of every communication.

### 3.5.4 Synchronous Receive

When the MPI_Recv function is called by the user’s application the call is intercepted as with the MPI_Ssend. The process which is gone through within the intercepted receive is very similar to that shown within the section above except that there is no checking whether to simulate a fault and hence no simulating of deaths within the receive call. The flowchart below shows the stages and order of routines called within the intercepted receive. Although each of these elements looks simple many of them are actually much more complex and greater detail on each of them can be found in the following sections.
The receive starts by first checking the global error flag, locking it while doing so. If the flag is set then the clean-up and error handler routines, as described in detail later, will be called. If the error flag is not set then the calling process will update its status. This involves locking its own status window and setting the variables appropriately to indicate it is issuing a receive, who the source should be and other information associated with the message. When accessing its own status window to set the status the process will lock the window. Full details of all the status information contained in the status window is given in the relevant section.

Once the processes status has been set the receive call will actually be issued. When this completes the status window is again locked and the status information reset to indicate that no communication is currently being carried out. Finally the error flag is checked again to see if a fault was generated while the receive was being completed. If the flag is set then the clean-up and error handler routines will be called as before. If the flag is not set then the communication was successful and MPI_SUCCESS is returned to the calling application to indicate this.

Should the call ever enter the clean-up and error handler routines then this means that the communication has not been successful. To indicate this to the calling application MPI_FAULT, as described in the MPI_Ssend section, is returned.
3.5.5 Error Flag

A global error flag is used to indicate when one or more process is simulating being “dead”. An area of memory is declared on the master process and then published so as to be available for RMA calls. The memory allocation and publishing of the window occurs on all processes, but the window of only one of these processes is used by all the others so as to treat the flag as global. This value is protected by a lock such that only one process can access it at a time, for both reads and writes. Every time a communication call is issued the RMA call *get* is used to obtain the value of the error flag. If the error flag is set then the process will not issue the send or receive, instead it will enter the clean-up routine followed by the error handler routine. If the flag is not set then if the call was a receive it is issued as normal. If the call was a send then the fault simulation testing routine as described in the next section is entered. As part of this routine if a fault is to be simulated the RMA call *put* will be used to set the error flag.

3.5.6 Simulation Test

All faults are simulated within the send routine. This was to make development of the library simpler. Within the send routine the error flag is first checked as described above. If this is not set then the next stage is to check whether to simulate a fault or not. This check is contained within a separate routine. This means that different methods for determining whether to simulate a fault or not can tested with just the modification of one routine.

The default method for determining whether to generate a fault or not begins by generating a random number between 0 and 100. This value is then compared to the desired fault percentage which can be set by the user. If the value is below the fault percentage then a fault is to be simulated. If the value is above or equal to the fault percentage then a fault is not simulated. Whether to simulate a fault or not is indicated to the calling process through use of the return value. If a fault is to be simulated, a 1 is returned; otherwise, a 0 is returned.

If the routine returns that no fault is to be simulated then the send will be issued as normal after the process has set its status. This is described in full detail in section 3.5.8.
below. If a fault is to be simulated then the process for simulating a fault as described in
the following section is instead entered.

The method of fault simulation here uses MPI calls as a sort of software interrupt to
randomly generate hardware faults. It does not imply that the fault was actually a
communications fault. This method is contained within its own routine so could be
easily modified to base the fault generation on another method, such as at a constant
rate in time rather than a constant rate per communication.

3.5.7 Fault Simulation

When it has been determined that a process is to simulate a fault it must first call the
clean-up and error handler routines as any process discovering an error would do. This
is so as it can match any outstanding communications and be involved in the forming of
the valid communicator through use of MPI_Comm_split, which is a collective routine.
More detail on the matching of outstanding communications is contained in the clean-
up section.

After these routines have been completed the process calls MPI_Finalize and then exits.
The “dead” process must call MPI_Finalize before closing so as to ensure that the
application will close cleanly at the end of computation even after a fault. By having
the “dead” process act in this way, after having created the split communicator the
“dead” process is no longer involved and closes. This means that the library is
guaranteed to meet the design criterion set out earlier, that a “dead” process cannot be
involved in any further computation. Rather than have this be carried out in a separate
routine the “killing” of a process is included as part of the error handler routine.

3.5.8 Status Window

The status window is an area of memory declared on each process within an RMA
window such that other processes can access it through using the relevant RMA calls.
This window is needed to allow the successful clean-up of outstanding messages when
a fault occurs. The window contains general information about the process, such as
whether it is “alive” or “dead” as well as details of any communication it is currently
undertaking. The exact data contained in the window is shown in the table below as
long with the values each element can take.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
</table>
| 0       | Dead/Alive           | 0 – Dead
          |                      | 1 – Alive       |
| 1       | Communication Type   | -1 – Default
          |                      | 0 – Send
          |                      | 1 – Receive       |
### Table 1: Status Window Contents

Table 1 highlights the fact that datatype usage within the library will be limited to the types defined in MPI. The status information is being stored in an RMA window. Data elements within that window are accessed by giving the offset of the start of that variable in bytes from the beginning of the RMA window. All of the elements are hence being stored as integers so as to make calculation of these offsets easier. This means that the datatype being used must be converted to an integer value representing it before being stored in the status window. This is done within the MPI-FS library so the user does not need to be concerned about it. However, there is no provision for user defined datatypes and hence user’s applications are limited to the standard datatypes defined by MPI.

Below is a diagram showing the normal operation of the status window. Each process is in control of its own status window, setting and then clearing its status as appropriate. A diagram showing the use of the status window for dealing with outstanding messages is shown in the clean-up section.

Each time a process is to issue a communication call it will first set all of the status window information to the appropriate value. This is done by first locking its own status using the MPI_Win_lock routine to obtain the window lock, then setting the information based on the communications type, destination and other relevant information before finally unlocking the window. With this done the process can then
call the actual MPI communication routine. After the communication is complete a similar process of locking the window, resetting all of the values to default and then releasing the lock is gone through to show that the process is not currently in a communication routine.

3.5.9 Clean-Up

The clean-up routine is the first stage in dealing with an error. It is aimed at handling the situation where a process has issued its communication call while another process was setting the error flag. In this situation the communication would be left hanging unless the intended target issues the matching call. This is necessary as all of the communications being used are blocking. If the matching communication call for a blocking communication was not issued then the process calling the original communication routine would remain blocked and the whole application would deadlock.

![Figure 11: Clean-Up Routine Flow Diagram](image-url)
As shown in Figure 11, the clean up routine works by having the calling process cycle through using the RMA call `get` to obtain the status information of every other process in turn. The diagram below shows which status windows would be checked when the clean-up routine is run on process 1. Process 1 would first check the status of process 0, then process 2 and finally process 3.

![Diagram showing status window usage](image)

**Figure 12: Clean-Up Status Window Usage – Remote Windows**

The information retrieved from each processes status window is checked to see if the source/destination in the status window matches the rank of the process calling the clean-up routine. If they match then the correct communication call, determined so as to be opposite the communication type in the status window, is issued. The information contained in the status window, as previously described, is used to make sure this communication matches the original issued call exactly. This is done by first declaring a temporary buffer using the count and datatype status elements. The call is then issued with the buffer that was just assigned, the source/destination and the tag. The communicator is the working communicator as described earlier.

Without this routine if a process entered a blocking point-to-point communication routine just before another process generated an error then the application would deadlock due to the first process never having its communication completed and hence never discovering the error flag has been set. With the clean-up routine, when a process discovers a fault it will first match any communications calls which are directed at it from other processes. This ensures that deadlock is avoided. With this complete the process can continue with the error handling routine. This situation comes about due to the use of synchronous communications. Much more care must be taken to match the communications because they are blocking and the clean-up is a stringent test of the library.

If a process has its communication completed by a matching clean-up call then the process will discover the error after the communication is complete. Once the communication is complete it will check the error flag before returning to the calling application. If the process was completed by a clean-up message then the error flag must have been set otherwise clean-up would not have been called on any process. Thus when the process checks the error flag after having its communication completed by a clean-up message it will always find an error and return `MPI_FAULT` to the
application. In such a way it ensures that the application using the library does not try and use the data which was sent as part of the clean-up routine as if there was no error and the communication had completed as normal.

This problem was highlighted during the testing of the taskfarm application. The first thing a worker process would do upon receiving a task was to call the `sleep` routine and “sleep” for the number of seconds indicated by the task value. When the communication was completed via clean-up the data received by the worker was “junk” data. This often led to the worker sleeping for incredibly long periods of time and hence caused the code to timeout or to crash.

### 3.5.10 Error Handler

After a fault has been discovered within the library and the clean-up routine has been run to deal with any outstanding messages the error handler routine is then called. This routine completes the recovery process and provides the user with a communicator containing only the still “living” processes to continue their application on. The details of each of the stages carried out within this routine are described here.

The first step within the error handler routine is to create the split communicator. The routine `MPI_Comm_split` is called on the current working communicator with the color set to the processes status (alive or dead) and the key set to the processes rank in the working communicator. This returns a split communicator where the “dead” processes are within one communicator and the “alive” processes a separate one. The diagram below shows this process. It is this communicator which is then set as the active communicator which can be accessed and used by the application programmer. The ranks shown in the diagram indicate the ranks in the initial communicator so as to show which group a processor moves to after the split communicator call. In reality the ranks of processes in the new communicator may have changed from those they had in the original communicator.
Once the split communicator has been created each process then looks at its status and if it is dead follows the process of “dying” as described in the fault simulation section, 3.5.7. Before doing this however it must be involved in another collective operation to clear all the windows and the memory associated with them. The process must check if it is “dead” before calling this routine as the status information about being “alive” or “dead” is contained within the status window which is cleared as part of this routine. The “dead” process also prints out the time at which it simulated death. This allows users to check at what point in the computation a process generated a fault.

Processes which determine they are “alive” must also call the routine to clear the windows as described above. Once this is complete they must create a new set of error and status windows. These are created attached to the new split communicator. This step of clearing and then declaring new windows must be done so as to ensure the library will continue to work after a fault. The old windows are associated with a now invalid communicator and the processes may not have the same rank within the new communicator of “alive” processes as they did in the old. If the old windows were used then processes may end up setting or reading the wrong processes status window.

With this process complete an MPI_FAULT value is returned to the user from the communication routine they called. By checking this return value, doing some form of
fault recovery and setting the working communicator to equal the active communicator
the application can then recover from a fault and continue their computation. The use of
the library to build fault tolerant applications is described in much more detail in
chapters 5 and 6.

3.6 Alternative Implementation

An alternative implementation which was suggested involved the use of time-outs. This
would be the most likely method that an actual fault tolerant version of MPI would take
but has some drawbacks. The method along with its advantages and disadvantages will
be presented here along with a brief comparison with the chosen implementation.

In this case when a synchronous communication call is issued it would be replaced with
a non-blocking communication and a test. In this way it allows the communication to
be tested. The communication is tested regularly, most likely starting with a short delay
and then a longer delay. If it has not completed after a defined time then the
communication has timed-out. When this happens the process will set its status to an
error state and not complete any further communications.

In this way the fault will propagate through the system, from the original faulting
process, then to those who try to contact it, next to those trying to contact those who’ve
been in contact with the “dead” process and so on through the system until all of the
processes are aware of the fault.

This can be clearly seen in figure 14, shown below. The diagram shows a set of
processes which exchange data with their nearest neighbours on each iteration, much
like a standard image processing code. The first diagram shows the system after no
iterations, one process has decided to simulate a fault and none of the others are aware
of it. On the first iteration the communications from the neighbours of the faulted
process will fail. These processes will then enter the error state. On the next iteration
the neighbours of these processes will also have their communications time out and
they will enter the error state. By the third iteration all but one process are aware of the
fault and finally on the fourth iteration all of the processes will be aware of the fault.
Figure 14: Fault Propagation Implementation

It is this propagation of the fault through the system which is the greatest disadvantage of the approach. There is no global synchronisation of the processes when a fault is generated. From an application programmer’s point of view some form of synchronisation after a fault is desirable. This is to stop all of the processes and keep one from running ahead of the others so that the correct point in computation can be rolled back to when recovering from the fault.

The advantage of this method is the simplicity with which it could be implemented. When a process is to simulate being dead it simply has to enter MPI_Finalize and then exit. All of the other processes will gradually be informed of the death through the fault propagating through the system. In this way this method would not require any form of clean-up routine to handle unmatched communications. Some form of error handler which creates the valid communicator could still be used as the process which is to die could simply enter that routine before closing and other processes will also enter the routine when they discover the fault.

The chosen library implementation, using a global error flag, began to become more complex when it was realised that a clean-up routines would be required. This is because the clean-up routine would require local state tables to be included. At this point it was decided to consult an MPI implementer, Dr Stephen Booth. Dr Booth suggested the implementation given here which involves time-outs. However, he did also say that the implementation involving a global error flag was feasible but may have performance implications.
The time-outs implementation explained here would not suffer from any slowdown caused by the checking of a global variable for each communication and would hence also scale better to larger processor numbers. The time-out method could however suffer from issues to do with the time-outs themselves. The length of time to wait before saying a communication has failed would need to vary between applications. It could also have problems when there is load imbalance in the application. For example, when one process finishes its work and calls the communications routine but another is still working. In this situation the first process will decide the communication has failed if it is not matched within a set time and could generate a fault just because of the load imbalance if this length of time was chosen poorly.

As the project was mainly aimed at investigating fault tolerance methods for applications it was decided to continue with the initial implementation using a global error flag.
Chapter 4

MPI-FS Library Testing

The following sections detail the test cases which were used to check the functionality of the MPI-FS library. Each section contains a list of the key tests highlighted by the test case, a brief description of how the test case code works and finally a discussion of the issues raised by the test case. Finally some details on actual use of the library are presented and followed by some conclusions for the whole development and testing process carried out for MPI-FS.

It was planned to produce timelines for each of the test cases showing how the library functions. The obvious way to have done this would have been through using Vampir traces. Unfortunately Vampir employs the MPI profiling interface when producing its traces and hence is not compatible with the MPI-FS library as it too uses the profiling interface. Time stamps on messages produced by the library documenting the process it is going through have been included in the library with the aim of being able to produce timelines from them. When running even the simplest applications the library produces a large amount of information and hence it makes generating timelines a time consuming and complex process. For this reason only one has been included and that can be found in section 4.2.

4.1 Message Round a Ring

Key Tests:

- Fault generation within the library
- Validity of the communicator produced by the library after a fault
- Error values being returned to calling applications after a fault
- Multiple faults at the same point in time
This was the first and simplest of the test cases. The code is much like that developed in the message-passing programming course. The diagram above shows how the processes can be imagined to be arranged and the direction in which the messages are passed. Each process sends out its rank to the neighbour on its right (process with a higher rank unless you are the end process in which case it is sent to 0). When the value is received the receiving process will add this value to a running total and then the messages value will be passed on to the next process to the right and so on. As such after a complete circuit passing the message on the right up to the number of processes being used each process should have the sum of the ranks of all the processes involved in the computation.

The original code used asynchronous sends so that each process could send out a non-blocking send, issue the synchronous receive and then wait for the send to complete. This method meant that the application avoided deadlock. Unfortunately the library only allows for synchronous communications and hence the application needed some modifications. The sending and receiving of data was split into two phases. This process is shown in the diagram below.

The modification was fairly simple. In stage one if the processes rank is even then it issues the send while if the processes rank is odd then it issues the receive. In the second stage this is reversed such that even processes issue a receive while odd processes issue a send. The diagram shows at which stage each communication
completes. There is a slight oddity with the last process seen when the total number of processes being used is odd. In this example the last process is 4 and as it is even it sends during stage 1. However, it is sending to process 0 which is also treated as being even. In this case the receive will only be issued by process 0 on the second stage and hence the send from process 4 does not complete until the second stage. This means that both the send and receive from process 4 occur as part of stage 2, one after the other. This could cause a slight delay to the applications progress but this method does avoid deadlock.

When a fault was generated within the library all subsequent communication calls should return an error value to the calling application. Within the test application here the return value of each call is tested. If the call is an error then a special routine to test the validity of the returned communicator is called. In this way if the test of the new communicator is carried out then it is clear that the return calls are working as expected.

The routine for testing the new communicator simply works by having each process which reaches the routine print out its rank in the new communicator as well as in the original communicator. An output sample from when running on 6 processors with a fault percentage of 10% is shown below.

```
New Rank:0 New Size:5 Old Rank: 1 Error Value: 1000
New Rank:1 New Size:5 Old Rank: 2 Error Value: 1000
New Rank:2 New Size:5 Old Rank: 3 Error Value: 1000
New Rank:3 New Size:5 Old Rank: 4 Error Value: 1000
New Rank:4 New Size:5 Old Rank: 5 Error Value: 1000
DEAD: Process 0 died after 0.005038 s
```

Figure 17: Generation of a Single Fault

Figure 17 shows that process 0 generated a fault and “died” 0.005038s into the running of the application. The output from the other processes shows their ranks in both the new communicator of “alive” processes as created by the library as well as their rank in the original communicator. Every process has had its rank changed between the old and new communicator. This is because process 0 of the original communicator “died” and when creating the split communicator ordering of process ranks begins again at 0. This meant that all the surviving processes had their ranks shifted. The output also shows the error code which was returned to each of the surviving processes. This error value is 1000 which is the same value as MPI_FAULT is set to and hence shows the correct error code is being returned after a fault.

At this stage the generation of multiple faults at the same moment was also looked at. An output sample where two faults occur at the same time can be seen below. The application was started on 6 processors. At around 0.0062s faults occurred on both process 0 and 2 and after being involved with creating the split communicator they printed that they had failed to the screen and then exited. The other processes upon discovering the fault printed the same messages as described previously.
This clearly shows that when two faults are generated simultaneously the library still produces the active communicator as expected and returns the error code MPI_FAULT to the calling application.

### 4.2 Manual Reduction

Key Tests:

- Clean-up routine operating when there are many sends to a single process
- Fault discovery without issuing a point-to-point communication

![Diagram of Manual Reduction](image)

The communications involved in a manual reduction are shown in the diagram above. A reduction involves carrying out a global sum on one process, the root. In this test case this is done by each process sending its rank to the root. The root meanwhile issues the receive calls in incremental order and each time adds the received value to a running total. The computation is complete once the root has received from all of the other processes involved in the computation.

During the development of the library some concerns were raised as to how the library would handle the case where several processes all issue sends to one process which has already faulted or responded to the error flag being set. The manual reduction was a test case which stressed this issue as all processes send to one root process. The clean-up routine is run on the root process when it discovers a fault and this matches all of the outstanding messages from the other processes and hence allows them to all respond to the error flag correctly. The incorrect and then correct operation of the application in this situation is shown in the diagram below. This test highlighted the need for some form of clean-up of outstanding blocking messages when a fault occurs. When first running the code without clean-up it would often deadlock as communications that had been issued by one process would not be completed by the appropriate process as a fault had occurred.
In this diagram four processes are involved in the manual reduction. Each process checks the error flag before issuing its initial communication. After this those processes which are sending (1-3) also check whether to simulate. In this example processes 1 and 2 have a zero returned to them and hence do not simulate a fault but issue the send to the master. Process 3 however is returned a 1 and hence simulates a fault. Process 3 calls the routine to simulate its death and then sets the error flag to inform other processes that there is a fault. After completing this process 3 then enters the clean-up routine, finds no outstanding communications for it to complete and then calls the error handler routine.

Process 0 has issued a receive with the source set to 1. This is completed by the send from process 1. Both process 0 and 1 check the error flag again before returning from the intercepted MPI call. As process 3 has failed during the completion of this communication and set the error flag they both find the error and call the clean-up routine. Process 1 checks the status windows of all the other processes and finds no process with their target/source set to 1. This indicates there are no outstanding communications for process 1 to complete and hence it calls the error handler routine. Process 0 however when cycling through doing the clean-up operation finds that process 2 has set its target to 0. Process 0 then looks at the rest of the status window information and determines that process 2 has issued a send to process 0. Process 0 then uses the other status window information (tag, count and datatype) to issue a receive call to match this communication and thus allow process 2 to progress.

When the library was first developed faults were only tested for when entering a synchronous point-to-point communication call. This meant that if a process’s first communication completed correctly but another process developed a fault at the same moment then the process would only discover it on entering its next point-to-point communication call.

With this simple test case only the root issues more than one communication call. This meant that if a process missed another process faulting and its communication completed successfully then it would never discover the error and hence never enter the error handler meaning that the program ceased working. To get around this issue the MPI_Finalize routine also had to be intercepted and within it the error flag value tested. If the flag was set then the error handler routine would be called and MPI_Finalize would return an error to the calling application. If the error flag was not set then the MPI_Finalize call would go through as normal and no error would be returned to the calling application.

4.3 Reduction using MPI_ANY_SOURCE

Key Tests:

- Library functionality when using MPI_ANY_SOURCE receives

In this test case again a reduction is being carried out. In this case however the root process issues sequential MPI_ANY_SOURCE receives up to the number of processes which will be sending to the root. As such it means that the sends can be received in
any order. It is very common to use MPI_ANY_SOURCE receives for these reductions as it prevents all processes having to wait for one process which must issue its send before the matching receive can be posted. MPI_ANY_SOURCE is also vital when implementing a task farm as you do not know at development time which tasks will be assigned to which processes and how long they will take to complete.

When developing the library some additional features had to be added to cope with MPI_ANY_SOURCE receives. Previously in the clean-up routine a process would only send to a process waiting to receive if that processes target rank matched the clean-up process’s rank. This was modified so that a clean-up send was issued if the ranks matched or if the target rank of the receive call was MPI_ANY_SOURCE.

Unfortunately this solution brought in more issues. It was now possible for more than one process to be in the clean-up routine and try and match the MPI_ANY_SOURCE receive. There were two possible solutions to this. The first was to have the process issuing the clean-up send set its own status to indicate what it was doing. In this way if more than one process issued a clean-up send the first would match the MPI_ANY_SOURCE receive and the second would be caught and cleaned up by the receiver when the original receiving process found the error and entered the clean-up routine.

The second solution was the have only one process carrying out the clean-up routine at a time. In this way it would mean that only a single process could match the MPI_ANY_SOURCE receive.

It was decided that implementing both methods had some benefits. The locking method was used to avoid multiple clean-up messages being sent. However, a process setting its status when sending a clean-up message was still useful as it made tracking of what each process was doing easier to follow and the functionality of the library clearer.

### 4.4 Task Farm

**Key Test:**

- Library’s compatibility with MPI_ANY_TAG receives
- Recovering from more faults after a previous recovery

This was not designed as an initial test case. However, when doing the first development work combining the library with the task farm code an issue involving MPI_ANY_TAG was discovered. After a fault being generated the clean-up routine was called as normal. All of the receive calls on the master are issued with MPI_ANY_TAG. This means that when a worker attempts to issue any clean-up messages to match an outstanding receive it will be attempting to issue the send with a tag of MPI_ANY_TAG. This is an invalid tag for a send and hence the code had to be modified to test for this and correct it by sending a valid tag instead.

When running the code several times some interesting cases with multiple faults occurred. In these situations the code would run as normal with the first fault being
recovered from successfully. However, when a second fault was generated the code would cease working. The cause of this seemed to be the fact that the processes had new ranks within the communicator created after the first fault and hence all the RMA calls to windows specified earlier were accessing the incorrect processors memory. To fix this problem the library had to be adjusted to redefine the RMA windows after each fault.

The issue comes about from the fact that RMA windows when they are created are attached to a communicator. The windows were being created during MPI_Init and attached to the original working communicator. After a fault this communicator was set to the set of “alive” processes. The ordering of processes in this new communicator was aimed to be as close to how it was in the original communicator by using the processes rank as the key variable when calling MPI_Comm_split. However, a process may have a different rank in this new communicator if it was not the end process which died. The diagram below illustrates how this occurs.

The diagram shows the original communicator and the resulting communicators when a split is carried out over the processes status. As you can see processes 0 and 1 have the same rank in both the old and new communicator. However, process 3, 4, 5, 6 and 7 of the original communicator all have changed rank within the new communicator. As process 2 of the old communicator died all the processes after process 2 in the old communicator have had their rank shifted down a place. Thus if trying to access the windows of these processes using the new rank then the wrong area of memory would be looked at as the windows are attached to the processes positions in the old

![Figure 21: Split Communicator Changing Ranks](image)
communicator. By redefining the windows after a fault it meant that the RMA calls would then be looking at the appropriate window.

4.5 Library Usage

There are some key guidelines which must be adhered to for correct use of this library from within an application. Many of these issues came up while testing the library and to help applications programmers details of them are presented here.

(i) Only applications which use one communicator during computation and only communicate using synchronous point-to-point communications can use the library.

(ii) Only the defined datatypes in MPI can be used with the MPI-FS library. No accommodation for user defined datatypes has been included.

(iii) The working communicator as previously described must be set to the communicator which the application intends to use for the computation before any MPI routine is called.

(iv) Every communication call or call to MPI_Finalize must have its return value checked. This check must be carried out before any manipulation of data involved in the communication is done as the data returned after an error may be invalid.

(v) After a fault has been detected all surviving processes must set the working communicator to the active communicator provided by the library if they intend to continue the computation and issue communications calls.

4.6 Conclusions

The MPIFS library was completed such that it could simulate random hardware faults based on a fixed fault rate per communication. The library allows users to use point-to-point synchronous communications (MPI_Ssend and MPI_Recv). The library provides notification of when a fault is found via the return values of the communications called. It also provides for the applications programmer a communicator containing only “alive” processes once a fault has been discovered. The library ensures that there is no deadlock when a fault occurs as all communications calls are matched by the appropriate process. Some test codes which test the functionality of the library have also been developed and the results presented here.

The library has been written in such a way that it can be easily extended, such as seen later when collective communications were implemented as part of the domain decomposition section. Overall the development was successful. Future work might be to modify the library such that it uses the time-out implementation previously discussed. This would allow the library functionality to be increased by adding asynchronous communications and also remove the possible scalability problems associated with having a global error flag. The library could also be made better by
adding the possibility of using more than one communicator within a user’s application. This would be a fairly substantial amount of work and there was no time to carry it out as part of this project as fault tolerant applications needed to be investigated.

After having completed the development of the MPI-FS library on Ness the initial plan was to port the library to various other systems. After consultation with the project supervisor, Dr David Henty, it was decided to limit the development of the library and applications to Ness. Due to the complexity of single-sided communications, as used widely in the MPI-FS implementation, porting the library to other systems might not have been a simple process. The library was written to match the standards set in MPI-2 but this was no guarantee there would be no bugs during the porting process. Porting to other systems was planned and is vitally important but it was a choice between porting and working on developing fault tolerant applications. It was decided to work on fault tolerant applications on Ness as this was one of the key aims of the project.

The following section details the work carried out to make a fault tolerant version of a taskfarm application. The MPI-FS library was used as part of this to generate the faults and to provide the necessary interface to allow the taskfarm application to recover from the fault.
Chapter 5

Task Farms

5.1 Motivation

The first application which was to be made fault tolerant was a simple taskfarm. Although not widely used the taskfarm is possibly the simplest method of obtaining parallelism. Taskfarms are generally not used as they have too many constraints for most real world problems. The problem must be able to be broken down into many “tasks”. Ideally the number of tasks would be much greater than the number of processors to aid with making the application load balanced. These tasks need to require no communication with other tasks to be processed successfully. This is because in a taskfarm you do not know which task will be on which process and you do not know what order the tasks will be completed in.

With these constraints the most common use of a task farm is as a “harness”. In this role a task farm controls more complex tasks, which themselves are run on a group of processors. The task farm controls sending out these complex tasks to groups of processes and interpreting the results. The tasks can either be completely different application codes or independent copies of one particular application code.

There are several reasons why the first application to be looked at in this project is the task farm. The reasons for this are:

- Simplest possible application
- Includes sections which will test the fault simulation library thoroughly
- Widely discussed in the reviewed literature

It is the simplest application possible due to all the tasks being independent, the application using just synchronous point-to-point communication and also the lack of collective communication use. The application structure involves multiple sends to one processor from different sources as well as using any tag receives. These features should stress the fault simulation library and confirm its correct operation. Within the reviewed literature numerous methods such as checkpointing, keeping a list of tasks and their status as well as intercommunicators are all discussed.
At the outset of the project it was aimed to look at all the fault tolerance methods discussed in the literature. Unfortunately due to time constraints it was decided not to look at any other method than disk checkpointing. The other methods all have some drawbacks associated with them, whether that be the reliance on a root process surviving for the task list or the availability of dynamic process creation on the target machine for the intercommunicators. Domain decomposition codes are far more widely used and offer many more challenges for making fault tolerant due to their more complex structure and use of collective communications. For these reasons it was decided to complete the disk checkpointing version of the task farm and then spend the rest of the available time developing fault tolerant domain decomposition applications.

5.2 Framework Application

The framework application code was based on a simple MPI task farm code from the School of Informatics at the University of Edinburgh. The framework code is shown as [5]. However, modifications had to be made to the basic application before any fault tolerance schemes could be undertaken. These modifications are described in the following section.

5.2.1 Dynamic Processor Numbers

The original version of the task farm code required that there must be more tasks than processes. For normal operation of a task farm this is a reasonable requirement but this would not be suitable for a fault tolerant task farm. For example, if the task farm were being run on four processes and with one hundred tasks then a fault towards the beginning of the computation would cause no problems as there would still be more tasks than processes. However, were the fault to occur near the end of computation then it is most likely that there will be more processes than uncompleted tasks. This would lead to errors in the task farm application.

To remove this problem the task farm code needed to be dynamic such that it would work with any number of processes and any number of tasks. Tasks were sent out on start-up up to the number of processes available or up to the number of uncompleted tasks left, whichever was the lower. The number of tasks sent out in this start-up phase was recorded. This information was used in the last phase where the processes close down to ensure that the same number of receives are issued so as to match the number of tasks sent out at the beginning. This ensures that all of the task results are returned correctly.

5.3 Disk Checkpointing

In the following section a description of how the framework application above was modified so as to implement a disk checkpointing method of fault tolerance is described. To see a working version of this code, as implemented as part of the project, please see the submitted source code. Disk checkpointing is one of many possible methods for making an application fault tolerant and it is not always the best one. It is used in this situation to illustrate how fault tolerance could be achieved. Other methods
would follow a similar structure as to that used here so it is still a useful method to look at.

Below are diagrams showing the general operation of the fault tolerant taskfarm application using disk checkpointing. The first deals with processes starting the code, deciding if they are a master or worker and how faults are dealt with. The second and third diagrams show how the farmer and worker routines work in practice. In these diagrams points where the MPI-FS library may generate a fault are indicated by an ‘E’ on the diagram. At any point where a fault could be generated it is tested for and the routine passes MPI_FAULT to the calling function should a fault be detected.
Figure 22: Taskfarm Application Flowchart
Figure 23: Master/Farmer Routine Flowchart
These diagrams explain the general functionality of the disk checkpointing version of the task farm application. In the following sections some more detail on the more interesting points will be discussed.

5.3.1 Creating Checkpoints

The first thing to be decided when creating checkpoints was what data would be necessary for a restart to be carried out after a fault. For the taskfarm to restart it would first need all of the tasks, both those which have been completed and those yet to be completed. Secondly, the application would need to know the results of any tasks which have already been completed. Finally, the application would need to have some indication of what task the application had reached before encountering a fault.

All of the tasks are kept in one array and all of the results in another array. Clearly both of these arrays need to be written out in there entirety as part of the checkpoint. This satisfies the first two requirements for restarting and now only an indication of on which task to start is necessary. To provide this a new routine was added to the application. Each element in the results array is set to -1 until a result has been calculated for the matching task. The new routine cycles through from element 0 in the results array until it encounters a -1. The results array element which contains the first -1 is the first task which should be sent out when restarting the application. The result of this routine is also written out as part of the checkpoint.

As the arrays containing tasks and results are only on the master process it is only the master process required to create checkpoints. There is also no need for any communications with other processes as part of checkpointing. Checkpoints need to be created at two points. The first is an initial checkpoint after the tasks have been created. This is to ensure there is a checkpoint available to load should a fault occur before the second point at which checkpoints are created. The second point is during the sending out of tasks. The user can set how often they would like checkpoints to be created. This is set as the number of tasks to be sent out between creating checkpoints. Every time the master is to send out a new task they will check whether to create a checkpoint.
based on this value. Should they need to create a checkpoint then the correct routines are called and the checkpoint created.

In this taskfarm application the task is simply to square the integer task value and return this as the result. All the information stored is hence going to be an integer. To create a checkpoint a new checkpoint file is first opened, clearing any old ones in the process. `fprintf` is then used to write out the task number to restart at followed by loops over the two arrays, tasks and results, writing out every element of these arrays. With this completed the checkpoint file is then closed.

5.3.2 Loading Checkpoints

The method for loading the checkpoint data is very similar to that used when creating them. The checkpoint file is first opened for reading. Then `fscanf` is used to first read in the task to restart from and then the complete tasks and results arrays are read in. Finally the checkpoint file is closed.

Loading of a checkpoint occurs after a fault has been discovered. How the fault is discovered and the complete recovery process are described in detail in the following sections.

5.3.3 Fault Discovery

Every time a communication routine, MPI_Ssend or MPI_Recv, are called the return value is checked. If this value is not MPI_SUCCESS then the calling routine returns the error value to the routine which called it. By checking the return value of all routines which could have an error thrown to them in this way it allows for the error to be caught at the appropriate level and dealt with. The diagram below shows how the fault would be thrown from an error when was found by the master during the receiving of a task.

![Fault Passing Diagram](image)

**Figure 25: Taskfarm Fault Passing**

Once the error has been found then the recovery process must be gone through. This process is described in the following section.
5.3.4 Fault Recovery

When the taskfarm routine fails an error handling routine called error_taskfarm is called. This routine does not duplicate code from the taskfarm routine but co-ordinates the fault recovery process. When the error_taskfarm is called it first sets the working communicator of the MPI-FS library to the set of active processes it will have generated after the fault. The new master process in this communicator (rank 0) then loads the checkpoint data. With this done each process then recursively calls the taskfarm routine again and the application should continue as normal.

To catch future errors the taskfarm launched after the fault also has its return value checked. Should this be MPI_FAULT then a new error_taskfarm routine will be called and the same process gone through. In such a way recursive calls to error_taskfarm allow for multiple errors during the computation to be caught and dealt with. Should the number of alive processes contained in the returned communicator ever fall below 2 however then the application will exit as there are not enough processes left to complete the computation.

As errors could be caught within MPI_Finalize the error code from this is also checked. If MPI_Finalize returns MPI_FAULT then the error_taskfarm is called as normal. However, the argument finalize is set so as to indicate that the error was caught in MPI_Finalize and hence that MPI_Finalize must be called after the recovery process has been completed to ensure that the application terminates cleanly.

5.4 Results

In this section some results from sample runs of the fault tolerant task farm application will be presented. There will also be a discussion on the interesting features these results highlight as well as brief notes on interesting things found during the testing process.
Figure 26: Progression of “Alive” Processor Numbers during Task Farm

The diagram above shows how the number of “alive” processes progresses as tasks are completed in a sample run of the task farm application. These results were generated with the fault percentage set to 5.5%, a maximum number of deaths set to 6 and with the task farm to complete 100 tasks. The job was run on 8 processors of Ness.

When attempting to get these sample runs it was noticed that at a certain point the library generates too many faults for the task farm to really get going. If the fault percentage was too high then when sending out the initial tasks and subsequently returning the result a fault was often generated. The application would then restart on fewer processes, load the initial checkpoint and start from the beginning again. When this happened another fault was often generated in the initial set of tasks and a cycle of faulting and reloading continued with no tasks being completed until the number of active processes became lower. It was this issue which gives rise to the two processor deaths before any tasks have been completed seen in Figure 26. As faults are not generally based on number of communications and actually occur infrequently it would be unlikely to see this happen in practice when using a genuinely fault tolerant MPI implementation.

When running the application as it is, the fault percentage needs to be carefully matched with the number of processes being run on. For smaller numbers of processors a higher fault percentage can be set as there are fewer communications being sent. As the number of processors increases the fault percentage should be lowered so as to not get into the cycle of faults and reloads with no progress being seen.
The diagram above shows how the processes arrangement into master and workers was rearranged as the number of “alive” processes changed over the completion of the application on the sample run. As can be seen in the diagram above, in this particular run it was always the processor which is the master that died. The reason for this is that the master process issues far more sends than any other process. It must send out all the initial tasks and then respond by sending out a new task once a worker has completed their task. A worker on the other hand only issues a send when returning the result from their current task to the master. This means the master will issue at least as many sends as there are tasks while on average a worker will issue just the number of tasks divided by the number of workers.
As the faults are generated on the sends, the more sends a process issues the greater its chances of having to simulate on one of those calls. An alternative method for deciding whether or not to simulate a fault may give a more even spread of the deaths over the master and workers. This sample does however highlight the advantage of disk checkpointing that allows for any process, including the master, to die but for the task farm to still be recoverable. Examples where the worker died were also seen but it was thought this sample run highlighted the issues involved with having faults generated based on number of communications.

As the results being generated by the application are simply a square of the task value checking the results after the application had completed was a simple case of checking the results were correct. During the computation each process also writes to screen which task it is sending or receiving as well as if it finds a fault when carrying out one of these communications. When a checkpoint is loaded the full list of tasks and results as loaded from the checkpoint are printed to screen. By checking these as the program progresses and faults occur it was possible to confirm that the application was running correctly.

5.5 Conclusion

This chapter has described the implementation of a fault tolerant version of a taskfarm application using disk checkpointing. The application can now recover from a fault generated in any of the called communication routines. When a fault is discovered the application loads the most recent checkpoint and then continues from the last completed task. As shown in the results section the application can cope with multiple faults and will continue to run as long as there are at least two surviving processes.

Overall the development was very successful. The results section documents the code working effectively and also mentions an issue encountered where many faults are generated close to the start of the application. This is unrealistic in that it is unlikely for all faults on a real system to occur within a short space of time right at the beginning of the computation. This issue can be avoided by modifying the method for generating faults in the MPI-FS library. Currently faults are generated on a fixed rate per communication. If this were changed to be a fixed rate per hour then the faults could be more evenly spread over the applications runtime. It would be a good piece of further work to implement this alternate method for generating faults.

After having completed the disk checkpointing version of the task farm code it was decided to move on to making a fault tolerant domain decomposition code rather than look at other fault tolerance methods for taskfarms. There were several reasons for this.

The first reason not to look at other fault tolerance methods for taskfarms was time constraints. The library took much longer to develop than expected and it was felt that cutting the different taskfarm fault tolerance methods was the best way to move forward. Secondly, domain decomposition types of code are far more widely used than taskfarms and hence looking at practical fault tolerance methods for domain decomposition problems would be far more useful and interesting.
Finally, the other methods of fault tolerance proposed for task farms have inherent 
drawbacks as described in the project preparation report. The task list approach would 
not take advantage of the active communicator provided by the library and not be able 
to work if a fault occurred on the master. While an intercommunicator approach would 
involve dynamic process creation routines which were unavailable on Ness. With 
having decided to concentrate on just developing on Ness at the end of the library 
development this was clearly going to have to be cut.

After having completed the disk checkpointing method for making a fault tolerant 
taskfarm application the next step was to look at domain decomposition problems. 
Details of this application and the methods used to make it fault tolerant are described 
in the following chapter.
Chapter 6

Regular Domain Decompositions

6.1 Motivation

Many real applications such as image processing, cellular automata, computational fluid dynamics, ocean modelling [6] and quantum chromodynamics are best parallelised using a domain decomposition approach. All of these applications have the following features in common:

- Nearest neighbour interactions
- Equal amount of work associated with each element
- Occasional calculation on some value based on all the data elements

A domain decomposition solution involves breaking the complete data set into “chunks” and assigning each one of these to a separate processor. Each chunk is a contiguous section of the complete data such that as many data elements are kept on the same process as their neighbours. Each process works on the “chunk” of data it holds locally. Every iteration halo data is exchanged. This is done so that those data elements on the edge of the local “chunk” also have the nearest neighbour information necessary for the processing.

Domain decomposition works well with the applications described earlier for the following reasons:

- Communications costs are minimised
- Every process has a similar amount of work, avoiding load imbalance

Clearly with so many key application areas using domain decomposition as a method of parallelisation developing a sample fault tolerant domain decomposition code would be beneficial. It would provide information on the interface a fault tolerant version of MPI should provide as well as how to make the application itself fault tolerant.
Many interesting features such as collective communications which had not been tackled in previous versions of the fault simulation library would also need to be developed.

6.2 Framework Application

The framework application code was an image processing code as developed for the message-passing programming course. Several modifications were necessary before any work could be carried out to make the application fault tolerant. These modifications are discussed below. Full details on the original image processing code itself will not be presented here. For more information on this please see the previous report submitted as part of the MPP course or the handout associated with the coursework [7].

6.2.1 Synchronous Communications

The original application as developed for the message passing programming course used asynchronous sends with synchronous receives. The fault simulation library as previously described was developed to only allow for synchronous point-to-point communications. For this reason it was necessary to modify the applications code.

It was decided to use a parity method of carrying this out. In this method each process in the topology is assigned as being either a one or a zero. This assignment is based on the processes position in the processor topology. The processors are assigned such that the one and zero processes resemble the black and white pattern of a checkers board. The diagram below shows the desired allocation of ones and zeroes for a 4 x 4 array of processors.

![Figure 28: Parity Arrangement](image)

This is achieved by using the following formula with the processes position within the processor topology:

\[ \text{parity} = (\text{position}[0] + \text{position}[1]) \mod 2 \]

With each processor assigned to either the ones or zeroes group it is now possible to have the image processing code work with synchronous communications. This is
achieved through having one group first send each separate halo to the correct neighbour and then receive the halo data from its neighbours. The other parity group first receives the halo data from its neighbours and then sends out the halo data to the appropriate neighbours. The stages involved in this process are shown in the diagram below.

![Diagram of Parity Halo Swapping](image)

Figure 29: Parity Halo Swapping

6.2.2 Dynamic

The original version of the image processing code was static. This meant that the number of processors and the topology they are arranged in was set at compile time, not runtime. Clearly when the application encounters a processor fault the number of active processors will be reduced. These faults occur randomly during the applications runtime and hence a static declaration of processor numbers and processor topology would not be suitable. In this way the original application code needed to be modified so that it would work with any number of processors as well as generate a suitable processor topology.

To have the application now generate an appropriate processor topology itself rather than have it set at compile time the routine MPI_Dims_create was used. This takes the number of processes you want arranged in the topology (the size of the communicator being used in this case), the number of dimensions you want the topology to have (2)
and returns a suitable topology in the array dimensions. This array can then be used as before to create the processor topology to be used as part of the image processing code.

Previously the number of processes and the topology they were arranged in was chosen so that it matched with the image being processed. That is that the image could be evenly divided over the processes in the process topology with each process getting the same amount of pixels as all other processes. The diagram below shows how this used to work.

![Static Data Decomposition](image)

**Figure 30: Static Data Decomposition**

As the number of processes and the topology would now be changing at runtime it could not be guaranteed that all of these topologies would match the original codes criteria for working. As such the code needed to be changed to deal with ill-fitting images where each process may not get an even amount of the data array. An example of this is shown in the diagram below.
To make the image processing code be able to handle this, two new variables in each dimension were needed. These were called \( n_{\text{max}} \) and \( n_{\text{pmax}} \) for the N dimension, and \( m_{\text{max}} \) and \( m_{\text{pmax}} \) for the M dimension. Shown below are the functions used to calculate \( n_{\text{max}} \) and \( n_{\text{pmax}} \). Identical formulas but working with the M direction variables were used to determine \( m_{\text{max}} \) and \( m_{\text{pmax}} \). Here \( n \) is the size of the image in the N dimension and \( dN \) the number of processes in the N dimension of the processor topology being used.

\[
\begin{align*}
n_{\text{pmax}} &= (n + dN - 1)/dN \\
n_{\text{max}} &= n_{\text{pmax}} \times dN
\end{align*}
\]

\( n_{\text{pmax}} \) is equal to the maximum number of data elements in the N dimension that a process could have. \( n_{\text{max}} \) is the maximum total image size in the N dimension. These are used to define the sizes of the local data arrays (big enough for each processes section of the data) and the global data arrays (big enough for the total image data) respectively. Each process must only carry out the image processing on actual elements of the data array. Clearly the end processes in the N dimension may have less data elements than the other processes. This means that they must work over a different number of data elements in the N dimension. Each process works over \( n_p \) elements in the N dimension when doing the image processing. For most processes this value is set to \( n_{\text{pmax}} \). For the any process which is the end process of a row in the N dimension though \( n_p \) is set by the following equation.
\[ np = n - (position[0] \times n_{\text{max}}) \]

By defining the arrays in such a way and also determining the number of elements to be processed by each process it allowed for ill-fitting images to be accommodated by the code and hence allow the fault tolerance schemes to be implemented.

In the original image processing code arrays were declared statically. This ensured that the data elements were consecutive in memory allowing the use of derived datatypes for passing around the halo data. Unfortunately the normal function for declaring arrays dynamically, `malloc`, does not ensure that the data elements of the array are consecutive in memory when the array being declared is multidimensional. To make sure that the data elements were consecutive in memory a library provided by Dr David Henty was used. This library contains a routine called `arralloc` which allocates memory in a similar way to `malloc` but ensures that the resultant array has all the data elements consecutive in memory even when the array is multidimensional. Details of this routine can be found in the relevant documentation that comes with the `arralloc` library.

Now that `arralloc` is being used to declare the memory some more changes needed to be made to the code. To use `arralloc`, whenever you want to pass an element of the array or the array itself it must be the address of the element or the first element of the array which is passed. The other option is to pass around the pointer to a pointer which represents the start of the array. This method is used when wanting to pass the whole of an array to a function while passing the element addresses is used when calling MPI library routines.

6.2.3 Derived Datatypes

The image processing code used a derived datatype when swapping halos between neighbours in the vertical dimension. The fault simulation library as described earlier was developed to only be able to deal with the standard MPI datatypes. This situation left two choices, to alter the application so as it did not use derived datatypes or to modify the fault simulation library so that it could handle user derived datatypes. Both of these choices offered different advantages and disadvantages.

Modification of the library to handle user derived datatypes would have meant a large amount of work on the library. The status window would have needed modification so that it could store more detailed datatype information. It would also have been necessary to intercept the function calls declaring the derived datatype itself so that this information could be stored for later use in the library. Clearly this large amount of work would be time consuming and take time away from developing the fault tolerant applications. The advantage of this method would be that the functionality of the library is greatly increased. Many more applications could use the library without requiring modification if the library were able to handle user derived datatypes.

In most implementations when using a derived datatype it will involve the MPI library packing and unpacking the data into a temporary buffer itself before sending and receiving messages. By following a similar method it would be possible to remove the use of derived datatypes from the image processing application. Basically the packing
and unpacking of data before communications would be put in the hands of the applications developer rather than being handled within MPI.

This method requires a little work on the applications code to pack and unpack data when sending and receiving the halos from neighbours in the vertical dimension. For the send this involves a loop over the data array putting the correct elements into a temporary buffer and then sending this buffer. For the receive call the data would be received into a temporary buffer and unpacked in a similar way into the image processing array. The advantage of this method is its simplicity while the disadvantage would be that these modifications would be needed for any applications code wanting to use the fault simulation library.

When using temporary buffers for sending and receiving halos they must be zeroed before each use. Some bugs were seen during the running of the image processing code when this was not done. Those processes on the boundaries of the topology would be sending and receiving halo information to MPI_PROC_NULL. When receiving from an MPI_PROC_NULL the receive would not be issued and no data placed in the temporary buffer. This meant that any data from previous communications using that buffer was not overwritten and hence the wrong data was being used. Zeroing the buffers between uses fixed this problem.

6.2.4 Collective Communications

The image processing code being looked at involves several collective communications. These are broadcasts, reductions and all reductions. It was feared that if the collectives were used as normal then this could lead to deadlock. Deadlock would occur if one process reached and called the collective communication while another generated a fault in a point-to-point communication before reaching the collective operation. This would cause deadlock in a similar way to when there were unresolved blocking point-to-point communications which caused a process to never discover the fault as described in section 4.2.

In this situation the process which has called the collective operation would never discover the fault and hence the application would deadlock. To avoid this, the collective communication routines were intercepted within the MPI-FS library and implemented using point-to-point communications. These point-to-point communications themselves are intercepted as normal by the library such that faults can be generated, caught and dealt with as described in the earlier implementation section. By implementing the collective communications in this way it meant that deadlock would be avoided and also the situation of faults occurring within a collective communication could be looked at. This step improved the functionality of the MPI-FS library, allowing for users to use broadcasts, reductions and all reductions.

6.2.5 Modularisation

The original image processing application was written as one long program with no function calls other than to the MPI library. To make the application fault tolerant this would need to be changed. Previously the image processing code would only ever be
run once, now that fault tolerance was being looked at the code may be run several
times as it is restarted after a fault.

To avoid code duplication the image processing code itself was placed into a function.
This routine took a communicator, the size of the image to be processed and whether or
not the routine was being called after a process fault as input variables. The
communicator passed is of the group of processes to carry out the image processing. At
the beginning this will be MPI_COMM_WORLD and after a fault it will be the set of
“alive” processes provided by the fault simulation library. The image size is used as
normal to decide the array sizes and to loop over the array elements when carrying out
the image processing. Finally, the error variable tells the routine whether or not the
routine has been called after a process fault. This is used so that if the function is called
after a fault a checkpoint will be loaded and used as the start point for the image
processing rather than the data file.

As well as putting the main image processing code into a function many other common
tasks such as the sending and receiving of halos were put into there own functions. This
was to reduce code duplication and also to allow errors to be “thrown” to calling
functions as described in section 6.3.4.

6.3 Disk Checkpointing

This section describes the implementation of the disk checkpointing method of fault
tolerance as carried out on the image processing code described in the previous section.
How the image processing code is called and how errors returned from it are handled is
shown in figure 32 below. Figure 33 below shows the flowchart for the image
processing routine itself. The sections following this give more details about each of the
stages shown in the diagram.
Figure 32: Image Processing Application Flowchart
6.3.1 Start-Up

The image processing code itself starts up in a very similar way to as described in the MPP report submitted earlier this year and with the modifications described in the previous section of this report. To add disk checkpointing however some further modifications to the code were needed. No matter if the routine is running for the first time or has been restarted after a fault the original data file must be loaded. This is then placed into the *edge* array. The difference comes in how the *old* array is initialised. If the image processing routine is called with no error having occurred then the *old* array
is initialised from edge in the same way as it was in the original code. At this point an initial checkpoint of the old array is created. The creation of this checkpoint is described in the following section.

If however the image processing routine is called after an error then a checkpoint must be loaded. The loading of the checkpoint is described fully in the section 6.3.3 below. The checkpoint is loaded into checkpointbuf by process 0 and then a broadcast is used to distribute the checkpoint data to all the other processes. Once this is done each process then initialises its old array from the checkpointbuf array in the same way as it would from the edge array in the initial code.

6.3.2 Creating Checkpoints

To allow the image processing code to create checkpoints one routine was added to the application and this was called from two separate points within the code. The new routine took a pointer to the data array which was to be written out as the checkpoint, the size in each dimension of this array and the iteration number. The routine opens a new checkpoint file and in the process clears any old checkpoint files. The routine then first writes out the iteration number. The actual data array is then written out in one block. All of the writes output the raw binary to the checkpoint file using fwrite. fwrite is used instead of fprintf as it means the data from the data array will be written out bit identical to what is stored in the array rather than it being rounded as would be the case for fprintf.

It only took the addition of one routine to create checkpoints because during the modularisation phase gone through with the framework application a routine called collect_data was created. This routine takes three arrays, local_data, positioned_data and collected_data. The routine positions the data from the local_data array into the correct position in the positioned_data array based on the calling processors position in the process topology. A reduction is then carried out to gather all of the data into the collected_data array on the one root process. By using this with the root as process 0, all of the image data can be gathered onto process 0. Once this has been completed process 0 can then call the create_checkpoint routine to actually write the data to disk.

This process of gathering the data and then writing it to disk is carried out at two points within the code. Firstly, an initial checkpoint is created when the application is first run after the data has been read from the input file. The second is during the image processing code itself. The user can decide the frequency at which the checkpoints are created during the running of the application. Based on this, at the start of each iteration the iteration number is tested. If it is divisible by the frequency at which checkpoints are being created then the data is gathered together and then written to disk. Else, on that iteration a checkpoint is not created.

6.3.3 Loading Checkpoints

As with the creating of checkpoints, as described above, to add the loading of a checkpoint to the application only required the addition of one routine. This routine opens the checkpoint file for reading and then uses fread to read the data into the
appropriate data array. Using `fread` instead of `fscanf` ensures that the data is read in exactly, with no rounding errors. The reading of the checkpoint file should only ever be carried out by one process. This is to ensure that the file system is not overloaded with too many processes attempting to open the same file. Instead, one process loads the checkpoint file and then this is distributed using the appropriate communications routines within the application code.

### 6.3.4 Catching Errors

Every time an MPI communication routine (either collective or point-to-point) is called or when calling any routine within the application which itself uses MPI communications the return value is tested. If this value is not equal to `MPI_SUCCESS` then it means that some error has occurred. When this happens the routine will print a suitable message to the screen and then itself return the return value it just received.

![Image Processing Fault Passing](image)

**Figure 34: Image Processing Fault Passing**

The program has been written in a very modular fashion, as described in section 6.2.5. This makes the code more readable, easier to debug and means there is no code duplication. The added benefit of this structure is that when a routine discovers an error during a communication it can print a suitable message to the screen and return the error it found which will be passed up through all the routines until it reaches a point at which it can be dealt with. In this way errors are passed all the way up to `main` where the error handling is carried out. The diagram above shows this process after an error has been found during an `MPI_Ssend` while sending halo data. The process of restarting the image processing application once a fault has been found is described in the following section.

### 6.3.5 Restarting

Once an error has been detected and the return value passed all the way up to `main` the error value is checked. If the value is `MPI_FAULT`, as defined in the MPI-FS library, then the recovery process is gone through. The checking for `MPI_FAULT` is used so as to ensure that the application only attempts to recover from faults generated by the library. The MPI version being used is not fault tolerant and hence unless the fault was generated through the fault simulation library the valid communicator will not have
been produced and hence the framework for recovering from the fault would not be in place.

Once the error has been determined to have been generated by the fault simulation library the routine `error_image_processing` is called. This routine does not replicate any code from the `image_processing` routine. It is there to co-ordinate the calling of `image_processing` again once a fault has been generated. `error_image_processing` takes the communicator in which the fault has been generated, the size of the image to be processed and a variable called finalize as arguments. The finalize argument is used to tell the routine whether the error was found when calling MPI_Finalize. This allows the routine to call MPI_Finalize once the image processing code has been run again successfully only if the error was first found when calling MPI_Finalize. This ensures that MPI_Finalize will be called and hence that the application will finish cleanly.

`error_image_processing` first sets the working communicator to the communicator provided by the MPI-FS library after a fault. It then calls the image processing code with this communicator and the image size as before. When calling the image processing routine this time the error variable is also set so as to ensure that the checkpoint is loaded and the computation restarted from there. As part of this routine the return value from running the image processing code is again checked. If a fault is discovered then a new instance of `error_image_processing` will be called and this will progress in the same way.

### 6.4 Results

![Graph showing the progression of "Alive" Processor Numbers during Image Processing](image)

**Figure 35: Progression of “Alive” Processor Numbers during Image Processing**
The diagram above shows the number of “alive” processors against the iteration number from a sample run of the image processing code on 16 processors. These results were generated running on Ness, with the fault percentage set to 0.04% and the maximum number of deaths allowed during the computation set to 15. The image processing code was running to a stop criterion of 0.5 (finishes on 1300 iterations) with checkpointing occurring every 10 iterations. As you can see faults occur randomly throughout the computation and the number of “alive” processes decreases over the running of the program until 15 processes have “died” and only a single process is left.

The default method for generating faults in the library is based on testing whenever a synchronous send is issued. This means that if you have a higher number of communications occurring then it is more likely one of them will generate a fault than if you had a small number of communications. In this example, at the start of the computation when there are more processes all doing halo swapping and calling collective operations there are far more communications than towards the end of the run where the number of processes and hence necessary communications has been reduced. It is for this reason that you see in the above figure that the number of faults is much greater at the start than towards the end.

There is an interesting point shown in the figure between when running on 12 and 10 processes. Here two faults have occurred in very close succession. When the code is running on 12 processes a fault is generated and the code detects the error and loads the last checkpoint. This checkpoint will be anywhere between 1 and 10 iterations behind the current iteration and the code is hence “rolling back” to a previous state. After this recovery another fault is generated before the next checkpoint is created and also at an earlier iteration number than when the last fault to drop, from 12 to 11 processors, was generated. It is this that causes the graph to look like the code goes backwards in iteration number for a moment between when running on 12 and 10 processes.

The diagrams presented below show the processor decompositions that the image processing code went through as processes “died” and the number of active processors was reduced. They also show the ranks of the processors in MPI_COMM_WORLD and when each of them “dies” and is split off. The first diagram shows the decompositions used between 16 and 13 active processors while the second shows the decompositions used between 4 and 1 active process.
Figure 36: Changing Decomposition after early Errors
When first running the code it was noticed that several of the processor decompositions produced from MPI_Dims_create were not optimal. For example, when running on 15 processes the default topology is 15 x 1. As the library was already intercepting many MPI calls it was decided to override these particular problematic processor numbers within MPI_Dims_create and return better processor decompositions for them.

Performance of the MPIFS library was also looked at during the testing of the image processing application. Two versions of the image processing application were used for the comparison. The first code was the standard fault tolerant version of the image
processing application with the MPI-FS library linked into it and the fault percentage set to 0. The second was the image processing application with checkpoints but without the MPI-FS library linked in. Each application was run to a laplacian of 0.5 on an image sized 192x360; this gives 1300 iterations to completion. The codes were run on 16 processors of Ness. The version linked with MPI-FS took on average 10.00425s to complete, compared with 2.976514s for the version without MPI-FS linked in. Clearly the version not using the MPI-FS library is much quicker.

The reason that the version linked with MPI-FS is slower is due to the bottleneck created where every process has to check the global error flag before completing any communication. Ness is a shared memory system that caters well for what could be considered a “shared variable”. If the performance on this system is poor the expected performance on distributed memory systems would be even worse. This shows that the bottleneck is significant and that the MPI-FS library is not optimal for performance. Performance however was not a design criterion for the library but it would be something to consider as a piece of further work on the library.

Finally the accuracy of the output produced by the fault tolerant application after a run where multiple faults have been recovered from was compared with the output produced by a run of the original serial code with the same input parameters. Below the output produced from both codes is displayed. A diff on the two output files shows them to be identical.

![Serial](image1.png) ![Fault Recovery](image2.png)

Figure 38: Image Processing Output Comparison
6.5 Conclusion

Despite having been able to generate these results the image processing code does still appear to exhibit occasional problems. Due to a lack of time all of these bugs have not been removed and the code does experience problems on a few runs. These seem to appear randomly as running the code several times with the same settings will generate runs which complete along with runs which stall at some point. These bugs may be due to some subtle timing errors within the image processing application and are most likely associated with the collective communications. No bugs of a similar nature have been seen when running with the task farm code and the only routines used by the image processing code and not the task farm code are collective operations.

This problem was first noticed when code was added to ensure the application was as efficient as possible with memory. This meant ensuring all memory was freed each time a fault occurred. It was thought that there must be some subtlety involved with memory management in this application which causes the bug. However, rolling back to previous versions without the memory freeing routines also began to display the bug. Unfortunately there was no time available to complete the debugging process and further work would be to make the image processing code more reliable and stable.

Subtle timing errors are not normally an issue with MPI applications. This is because synchronisation is not explicitly needed as it is inherent in the communications calls. However, as this project uses MPI-2 single-sided communications synchronisation becomes an issue. The library has to be able to deal with situations similar to “race conditions” in OpenMP. MPI is not really designed for this and hence the scope for bugs to appear in the code is quite large. The library was designed to be as robust as possible and through the test cases an iterative process of removing bugs was gone through. It appears though that some bugs still remain and there has been insufficient time to find and correct them.

In reality most applications would not use a disk checkpointing method for fault tolerance. Writing to disk is simply too slow to be used and not have a serious impact on the applications performance. Many applications however do already write results regularly and this is much like a checkpoint. With such applications were a fault discovered then the user would most likely restart the application by submitting the job again with the input data as the last created set of results. The implementation here has some advantages over this method. The recovery from a fault is autonomous, requiring no user intervention. This means that the recovery process will happen as soon as the fault occurs rather than at some point when the user realises there has been an error. The method implemented as part of the project will also ensure that the user’s application does not lose its slot on the machine being run on. All the fault recovery is contained within the program itself allowing it to recover from the fault and continue computation within the same processing slot.

At the project outset it was aimed that another method, data redundancy, for making a fault tolerant domain decomposition application would be looked at. With the large number of issues encountered when developing the disk checkpointing image processing code it took longer than expected and it was decided to concentrate on the
production of some results and discussion around them rather than to attempt data
redundancy.

Data redundancy may well have been more efficient than disk checkpointing due to the
fact that data is kept locally on the processes rather than being written to disk. However, using data redundancy would have given the same structure of capturing the
error, reconstructing the data somehow and then restarting with the recovered data as
disk checkpointing. Only the method of reconstructing the data before a restart would
have needed changing. This means that although it would have been interesting to look
at data redundancy and compare it with disk checkpointing, nothing new would have
been learnt about the interface needed between a fault tolerant library and applications
wanting to use it.
Chapter 7

Conclusions and Further Work

Overall the project was very successful. Firstly an interface between a fault tolerant MPI and the users application was decided on. This involved the returning of an error code to surviving processes when a fault occurred as well as providing a valid communicator of the surviving processes for future use. A fault simulation library which provides this interface was then developed. This library generates random hardware faults, deals with the resolution of outstanding messages, informs the user application of the fault and provides a communicator containing the surviving processes so as to allow for a fault recovery method to be instigated. This library provides the use of synchronous point-to-point communications and some basic collective operations such as a broadcast and summation reduction.

By using this fault simulation library two fault tolerant applications have been developed, a taskfarm and an image processing application. Disk checkpointing is used in both these applications so as to make them fault tolerant. During the computation checkpoints are regularly created. Once the application detects a fault through monitoring communication call return values, it then responds by loading the last checkpoint, switching to the communicator of valid processes provided by the library and then restarting the actual computation element of the application. It may also be interesting to implement a version of the fault simulation library which uses the time-out implementation described. This would allow performance comparisons between the two implementations to be looked at.

Disk checkpointing is not the perfect solution for fault tolerance but it serves as a good example of the libraries functionality. Having created these fault tolerant applications it shows that the interface provided by the fault simulation library is effective and useful. Any fault tolerant implementation of MPI needs to provide some sort of interface to user applications and from this project I would suggest that this interface must include the following features:

(i) A mechanism for informing all involved processes when a fault occurs

(ii) Provision of a valid communicator of surviving processes once a fault has occurred

73
There are several possible areas for further work that could be carried out to extend this project. Further work could involve extending the library functionality. This could include adding asynchronous point-to-point communications, including a full complement of collective communications and/or allowing users to use more than one communicator at a time in their application. By doing this it would increase the range of applications which could use the library and hence allow the investigation of fault tolerance for these applications.

A second area of further work would be to look at alternative fault tolerance methods for the two applications used in this project, a taskfarm and image processing code. Details of these schemes are provided as part of the background section in chapter 2. Initially these methods were to be included as part of this project but the library complexity led to a changing of the timescale available for developing fault tolerant applications and hence only disk checkpointing was looked at. Despite this fact disk checkpointing does illustrate the techniques required for making an application fault tolerant. The application must detect the fault, respond in some way to recover any data needed and then restart on the set of active processes left. This general method would need to be used for any form of fault tolerance.

Finally, if the functionality of the fault simulation library were extended it could allow for the fault tolerance of other applications to be looked at. Making a fault tolerant molecular dynamics, computational fluid dynamics or quantum chromodynamics application would be a significant challenge and very interesting to look at. I believe that the interface provided within the library would allow fault tolerance for these more complex applications to be tackled.

The project overall was a success with a interface for fault tolerance implementations decided on, a fault simulation library having been developed and fault tolerance for all applications specified at the project outset looked at.
References


