The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student) ________________________________

Fault tolerance in parallel distributed systems
Marc Johnson
Computer Science
2003/2004
Summary

This project involved an investigation into fault tolerance in distributed and parallel systems and implementation of fault tolerance in a number of example programs, exhibiting common characteristics of parallel programs, using the MPI standard.
Acknowledgements

This project is dedicated to the memory of my mother.

I would like to thank Pete Jimack for his support and advice as my supervisor even when he was very busy and Sarah Fores for listening to all the silly questions I had about the project and giving good advice.

I would like to thank my father for his support and Dorolyn Parker for proof reading the final draft of this report.

I would also like to thank all Mark Conny for recommending to me the valgrind memory analysing software even though it was probably too late into the project for it to be very useful. Mark should also take some credit for inspiring my interest in software engineering in the first year.
# Table of contents

Summary .................................................................................................................................................. I

Acknowledgements ................................................................................................................................... II

Table of contents ................................................................................................................................. III

1 Background.......................................................................................................................................... 1

Introduction ............................................................................................................................................. 1

1.1 Parallel computing ......................................................................................................................... 1

1.1.1 What is Parallel Computing? ................................................................................................. 1

1.1.2 Shared Memory Computers ................................................................................................. 1

1.1.3 Distributed Memory Systems ............................................................................................ 3

1.1.4 Programming Paradigms .................................................................................................... 4

1.1.5 Selected Environment ....................................................................................................... 8

1.2 Distributed Systems ..................................................................................................................... 8

1.2.1 What is a distributed system? ............................................................................................ 8

1.2.2 Transparency ..................................................................................................................... 9

1.2.3 Openness ............................................................................................................................ 10

1.2.4 Scalability .......................................................................................................................... 10

1.3 Grid computing ............................................................................................................................ 11

1.3.1 What is grid computing .................................................................................................... 11

1.4 Fault Tolerance ............................................................................................................................. 12

1.4.1 What is a fault? .................................................................................................................. 12

1.4.2 Failure Models .................................................................................................................. 12

1.4.3 Fault tolerance ................................................................................................................... 12

1.5 Previous work in this area .......................................................................................................... 14

Summary .............................................................................................................................................. 15

2 Objectives and Planning .................................................................................................................... 16

Introduction ........................................................................................................................................... 16

2.1 Aim ................................................................................................................................................ 16

2.2 Objectives .................................................................................................................................... 16

2.2.1 Objectives .......................................................................................................................... 16

2.2.2 Minimum requirements ..................................................................................................... 16

2.2.3 Possible extensions ............................................................................................................ 17

2.3 Methodology ................................................................................................................................ 17

2.3.1 Development ..................................................................................................................... 17

2.3.2 Testing Strategy .................................................................................................................. 17
2.3.3 Evaluation ................................................................................................................. 18

2.4 Planning ......................................................................................................................... 19
  2.4.1 Initial Plan ............................................................................................................... 19
  2.4.2 Characteristics analysis ......................................................................................... 19
  2.4.3 Example programs to develop .............................................................................. 20
  2.4.4 Extended plan ........................................................................................................ 20
  2.4.5 Problems with the extended plan ........................................................................... 21
  2.4.6 Final progress ....................................................................................................... 21
  2.4.7 Justification of alterations ..................................................................................... 21

Summary ............................................................................................................................ 22

3 Matrix multiplication example programs ................................................................................... 23

Introduction ........................................................................................................................... 23

3.1 Matrix multiplication algorithm ...................................................................................... 23

3.2 Master slave centralised work farm model ....................................................................... 23
  3.2.1 Other applications ................................................................................................. 23
  3.2.2 Master slave model .............................................................................................. 23
  3.2.3 Work farm distribution model ............................................................................. 24
  3.2.4 Key features of the original parallel program ........................................................ 24
  3.2.5 Implementation of the original program ............................................................... 25
  3.2.6 Fault tolerance design and implementation ......................................................... 26
  3.2.7 Evaluation ........................................................................................................... 29
  3.2.8 Summary .............................................................................................................. 31

3.3 Semi-distributed block decomposition model ...................................................................... 32
  3.3.1 Other applications ................................................................................................. 32
  3.3.2 Semi distributed model ........................................................................................ 32
  3.3.3 Block decomposition model ................................................................................ 32
  3.3.4 Key features of the original parallel program ....................................................... 33
  3.3.5 Implementation of the original program ............................................................... 33
  3.3.6 Code evolution ...................................................................................................... 34
  3.3.7 Fault tolerance design .......................................................................................... 35
  3.3.8 Implementation discussion .................................................................................... 36
  3.3.9 Evaluation ........................................................................................................... 37
  3.3.10 Summary ............................................................................................................. 40

3.4 Distributed block decomposition model ............................................................................. 40
  3.4.1 Distributed model ................................................................................................ 40
  3.4.2 Block decomposition model ................................................................................ 41
  3.4.3 Key features of the basic parallel program .......................................................... 41
1 Background

Introduction

The project undertaken concerns the area of fault tolerance in parallel computing, this leads me to introduce the subject in general. This chapter covers the hardware architectures and software paradigms of parallel computing and an overview of distributed systems and fault tolerance. Followed by a brief look at grid computing and previous work relating to fault tolerance in parallel computing.

1.1 Parallel computing

1.1.1 What is Parallel Computing

Parallel computing according to Allen and Wilkinson (1999), is a general term for utilising multiple processors in order to increase computational speed. Primarily this involves a method of programming involving splitting the problem into separate parts.

This is of particular interest as there are many problems that cannot be computed in reasonable time with the current speed of processors and for many of these it is unlikely that computers will reach speeds necessary to solve them in reasonable time in the near future. Examples of this sort of problem involve DNA mapping, weather prediction and aircraft simulation.

Currently there are 2 main hardware architectures, multiprocessor computers and distributed memory, along with 3 software paradigms in parallel computing. First covering the hardware architectures moving onto the software paradigms later in this section.

1.1.2 Shared Memory Computers

Both Wilkinson and Allen (1999) and Tanenbaum and van Steen (2003) agree that the main premise behind this architecture is that of multiple processors that can all access the memory (otherwise known as a single address space) of the computer. This is a natural extension of the single processor architecture found in standard personal computers (Wilkinson and Allen (1999)).

This same architecture is often referred to by several similar names such as the Shared Memory Multiprocessor (often referred to as SMP) architecture (Wilkinson and Allen (1999)), mainly due to the fact that this approach involves a tightly coupled group of processors in a system with access to shared memory.

Below the advantages and disadvantages (based upon Wilkinson and Allen (1999) and Tanenbaum and van Steen (2003)) of this architecture are outlined, although not all of these apply to all implementations of the architecture, as noted.
Advantages of Shared Memory Architecture

- **Reliability:** The system is a single computer, which means all hardware is tightly coupled and usually provided by a single vendor, so each processor and hence the entire system is very reliable.

- **Programming difficulty:** The programming paradigms (see below) for this approach are very similar to single processor development with a small number of additional directives.

- **Availability of data:** As all processors can access the shared memory, all data is available to all processors, and as such the overhead of synchronisation is reduced.

- **Low latency:** As stated above, all communication can be done via shared variables, thus is of low latency.

Disadvantages of Shared Memory Architecture

- **Technological Advancement:** Due to the tightly coupled nature of the computer, the different components cannot be easily upgraded to utilise new technology and so systems like this become out of date very quickly.

- **Scalability:** The problems outlined below (bus interconnection and cache synchronisation) mean that the scalability of the system is limited. Possible attempts at reducing this have included the provision of Non Uniform Memory Access (NUMA). This is achieved by utilising a hierarchical organisation of memory, allowing processors quick access to their own local memory and access to other processors memory at reduced speed. Though this introduces complexity in the design of programs, as they must be positioned correctly in memory in order to achieve high performance.

- **Cost:** The specially designed hardware and increased complexity of the Operating System means these systems are usually very expensive.

- **Bus interconnection:** If all of the processors are connected to the shared memory by a single bus, this quickly becomes a bottleneck in the system. Tanenbaum and van Steen (2003) suggest this can happen in systems with as few as 4 or 5 processors. Adding a high-speed cache to each processor reduces its need to access main memory but, as shown below, this introduces its own problems. Another possible solution to this problem has been to introduce switches by which many processors can access the shared memory simultaneously, although this adds considerable extra complexity to the hardware.

- **Cache Synchronisation:** Introducing local caches in order to reduce the need to fetch data from main memory introduces unique challenges for the operating system; the most important of these is the synchronisation of data in the many processors caches with that of the shared memory.
1.1.3 Distributed Memory Systems

Historically these systems have been single machines e.g. the Cray T3E, though the networked computers approach (see below) is becoming more popular.

1.1.3.1 Multi-computer Systems

Both Allen and Wilkinson (1999) and Tanenbaum and van Steen (2003) agree that the main concept behind this architecture is the utilisation of individual processors each with their directly connected memory and a means of communication between them.

Tanenbaum and van Steen (2003) make a distinction between homogenous and heterogeneous multi-computer systems, although this is limited to the nature of the individual computers that are connected together. In the homogenous case the computers involved are all identical, whereas in the heterogeneous the specification of individual computers can vary widely.

One important distinction made by Tanenbaum and van Steen (2003), relates to the interconnection network used in these systems, which may vary in speed from commonly available 100 Mbps Ethernet connections to proprietary connections e.g. Massively Parallel Processors (MPP). This is obviously reflected in the overall performance and cost of a system. In the case of the Ethernet this probably limits the number of computers that can effectively be used in the system.

Outlined below are the advantages and disadvantages (based upon Wilkinson and Allen (1999) and Tanenbaum and van Steen (2003)) of this architecture, although again not all of these apply to all implementations of the architecture, as noted.

Advantages of Multi-computer Systems

- **Accessibility**: If networks of standard workstations or PC’s are being used then much of the hardware is already available.
- **Scalability**: New computers (either PC’s or special units in homogeneous systems) can be easily added to the system to provide extra performance, as long as the interconnection method utilised can accommodate the extra communication required
- **Cost**: In the case of networked PC’s or workstations, these may already be available in an organisation and so incur little or no cost. In either homogenous or heterogeneous (networked PC’s) the cost of such systems is considerably lower than specially designed tightly coupled Shared Memory systems.
- **Future**: With the idea that considerable computing power will soon be ubiquitous and globally linked (see section 1.3) this type of architecture will be readily available to use.
Disadvantages of Multi-computer Systems

- **Programming Difficulty**: The paradigms associated with distributed memory are considerably different to sequential programming techniques and are argued by (Wilkinson and Allen (1999)) to be considerably more difficult and error prone in comparison.

- **Different Hardware\Software**: The different computers in the system may have different hardware and software increasing the programming complexity in areas such as load balancing, although this only applies to heterogeneous systems.

- **Communication and synchronisation**: The original data and results of any processing may only be available to a single processor and as such there is a high overhead in communicating this to one (or more) of the other processors involved, as and when they need it. This adds a certain overhead, which will reduce the overall performance of the system and increase the complexity of any software developed for this type of system.

- **Reliability**: Individual computers and the network connecting them, in the system will not generally be as reliable as the specialised hardware in a more tightly coupled system.

In relation to last point, the Mean Time Between Failure (MTBF) of modern processors and memory is lower than the computational time required for many modern problems and therefore a means of coping with failures without losing computation is required, which has lead to the subject of Fault Tolerance (see section 1.4).

1.1.3.2 Distributed Memory Architecture

Both of the previous architectures have different advantages and disadvantages, the Distributed Memory architecture, according to Tanenbaum and van Steen (2003) is seen to be a compromise between the Shared Memory Architecture (see section 1.1.2) and the Message Passing Architecture (1.1.3).

Wilkinson and Allen (1999) argue that this approach is just an alternative paradigm to the Message Passing approach when used in a multi-computer environment.

Unfortunately Tanenbaum and van Steen (2003) suggest that although large amounts of research have been conducted in this area, this approach cannot fulfil initial expectations of this to be as easily programmable as the Shared Memory architecture but as flexible as the Distributed Memory approach and so cannot be considered as a suitable alternative for this project.

1.1.4 Programming Paradigms

Following coverage of the various architectures available for parallel computing, it is now relevant to explain the different programming approaches available. It is important to note at this point that some of the programming paradigms can operate on a number of different architectures.
1.1.4.1 Parallel Directives

This approach as described by Wilkinson and Allen (1999) is a new parallel programming language with “special parallel programming constructs and statements that allow shared variables and parallel code sections to be defined” The advantages and disadvantages of this approach are summarised below (based upon Wilkinson and Allen (1999)).

Advantages of parallel directives

- **Programming Difficulty**: This approach is very similar to sequential programming and as such is probably the easiest programming paradigm to utilise.

Disadvantages of parallel directives

- **Shared Memory specific**: Threads can only be used in shared memory parallel computing architectures.
- **Compiler dependant**: Parallel directives have to be supported in the language and compiler.
- **Platform dependant**: Parallel directives rely on support in the underlying operating system; they are only available for shared memory systems.

1.1.4.2 Threads

Threads are defined in Silberschatz et al (2000) as the “basic unit of CPU utilization”. They go on to expand on this definition stating that all threads belonging to the same process share all of their resources. This idea has direct parity with the hardware architecture for shared memory as this is a relatively simple method of defining “tasks” (i.e. threads in this case) that can be executed simultaneously on individual processors but sharing all of their data. This makes them an ideal paradigm for utilisation on this type of parallel computer. This is confirmed by Silberschatz et al (2000) when they state “Utilization of multiprocessor architectures” as one of the four key benefits of using threads.

Threads are probably the most common programming paradigm for parallel programming and are used extensively in non parallel architectures (i.e. modern desktop PC’s) for a wide range of tasks, e.g. updating one thread updates a Graphical User Interface (GUI) while another does some processing in the background.

Threads have a number of advantages and disadvantages associated with them (Based upon Wilkinson and Allen (1999), Tanenbaum and van Steen (2003) and Silberschatz et al (2000)).

Advantages of threads

- **Easy to program**: Threads can be developed using existing programming languages and tools and as such are very easy for programmers to develop programs without much extra knowledge beyond traditional sequential programming.
- **Low latency**: Threads do not encounter the same delays required by message passing as all communication is done via shared memory, though message passing can be used on Shared Memory architectures with similar results (see section 1.1.4.3).

- **Efficient use of resources**: Threads from the same process share all of their resources and as such will use less memory and require fewer resources to create, according to Wilkinson and Allen (1999).

- **Established**: Threads are already used extensively in modern Operating Systems (OS) such as SUN Solaris and Windows NT.

- **Operating System Independence**: According to Wilkinson and Allen (1999) there is a standard for thread creation that is OS independent. This standard is called Pthreads, and although some languages, e.g. Java, do not use Pthreads, they do provide thread capability. This ensures that the program will be compatible with any system that has a Java implementation.

### Disadvantages of threads

- **Shared Memory specific**: Threads can only be used in Shared Memory parallel computing architectures.

- **Deadlock**: Both Wilkinson and Allen (1999) and Silberschatz et al (2000) provide an explanation of deadlock, which is a major problem with thread based execution using shared resources.


#### 1.1.4.3 Message Passing

Traditionally according to Wilkinson and Allen (1999) and Tanenbaum and van Steen (2003) this type of approach has been used on homogenous collections of individual processor and memory interconnected together, this is different to that of the Shared Memory architecture as each processor can only access its own local memory.

Both Tanenbaum and van Steen (2003) and Allen and Wilkinson (1999) later go on to describe the modern implementation of this slightly differently. Allen and Wilkinson (1999) go on to describe networked computers as a platform for message passing, suggesting that it is now widely recognised that the clusters or networks of workstations are an ideal alternative to tightly coupled homogenous multi-computers and then studying specific examples of the library approach to implementing this (e.g. PVM and MPI). Whereas Tanenbaum and van Steen (2003) describe this approach from the perspective of distributed systems, suggesting that message passing (or, as they refer to it, “Message Orientated Communication”) is a form of middleware (see section 1.2) utilised for high performance parallel computing in distributed systems.
Although both of these vary in their viewpoints and hence their bias to their particular areas, they both convey the same central idea; that message passing is no longer largely implemented on tightly coupled homogenous multi-computers, it is now used on groups of loosely coupled heterogenous computers. This definition of the message passing architecture is the one used for the duration of this project. The advantages and disadvantages of this approach are outlined below (based upon Wilkinson and Allen (1999) and Tanenbaum and van Steen (2003)).

**Advantages of Message Passing Architecture**

**Accessibility:** Message Passing can be implemented on many standard workstations that are networked in some manner.

**Hardware architecture independent:** Message passing can be used on shared memory and distributed memory systems.

**Portability:** Message Passing libraries have been implemented on many different OS and distributed system middleware, for example MPI can be used on Linux, Windows and grid computing architectures.

**Disadvantages of Message Passing Architecture**

**Programming Difficulty:** The paradigms associated with message passing are considerably different to sequential programming techniques and are argued by Wilkinson and Allen (1999) to be considerably more difficult and error prone in comparison.

**Different Hardware Software:** The different computers in the system may have different hardware and software increasing the programming complexity (e.g. load balancing).

**Availability of Data:** The original data and results may only be available to a single processor and as such there is a high overhead in communicating this to one (or more) of the other processors involved as and when they need it.

**Latency:** There is a considerable latency due to the delay in messages being passed (although this will be reduced to a minimum if message passing is used on a shared memory system).

**Message Passing Interface (MPI)**

MPI is the current standard for Message Passing. It is portable and has been implemented on many different combinations of hardware, operating system and network architecture, making it very popular, especially in the academic community.

It is a library of functions designed for current programming languages such as C/C++ and Fortran. Tanenbaum and van Steen (2003) suggest this was the result of a need for a hardware independent communication library and, critically to this project, the decision was taken to assume that any serious failures in the system are fatal and as such no method of automatic recovery (fault
tolerance) is defined, although as can be seen in section 1.6, there have been attempts made to implement this.

A good introduction to the basic functionality of the MPI library can be found in Jimack and Touheed (1999) and along with many parallel-programming ideas in Allen and Wilkinson (1999). Snir et al (1996) provides a good reference to the standard.

1.1.5 Selected Environment

Message passing is the ideal choice of environment as it can be used on the widest range of platforms, specifically distributed memory systems, which should make the project relevant to the largest audience. Message passing is also ideally placed to take advantage of the availability of large numbers of heterogeneous computers and with new developments such as Grid computing, to undertake very large parallel problems. The scale and distributed nature of these systems increase the likelihood of faults and so increase the need for fault tolerance. MPI is the most appropriate choice for message passing library, as it is portable and widely implemented, including being available on the School of Computing’s computing equipment.

1.2 Distributed Systems

1.2.1 What is a distributed system?

Tanenbaum and van Steen (2003) define a distributed system as “a collection of independent computers that appear to its users as a single coherent system”. This is an example of the distributed memory architecture for parallel computing and as such, it is important to understand this topic in more detail. They also suggest that in order for a distributed system to support heterogeneous computers and networks, while maintaining this single system view to the user, they are implemented by a software layer between the operating system of each computer and the user. This layer is known as middleware. This initial definition of a distributed system may seem relatively sensible and easy to implement but as discussed later there are a number of problems, which prevent us from providing this ideal.

Tanenbaum and van Steen (2003) suggest that there are a number of objectives that a distributed system should aim to conform to:

- Easy user access to resources
- Transparency
- Openness
- Scalability

Though Efford (2003) which uses Tanenbaum and van Steen (2003) extensively, does not stress the first of the objectives listed above, and adds security to this list, which Tanenbaum and van
Steen (2003) must also believe is important as they devote an entire chapter to it, but do not include it in the initial definition.

From the viewpoint of parallel computing using message passing, security is handled by the underlying operating system(s) of the individual computers, so is not a factor directly related to this project therefore more attention is paid to the other objectives, relating each one to the specific type of distributed system used.

1.2.2 Transparency

A transparent distributed system is defined by Tanenbaum and van Steen (2003) as being able to “present itself to users and applications as if it were only a single computer system”. As such, transparency is relevant at two levels in a distributed system; it concerns transparency to the end user, e.g. migration transparency, by hiding the location of a server in a chat system (Efford, 2003) or to the programmer (or application) e.g. by hiding the complexity of communication, such as sockets, with a higher level library.

Tanenbaum and van Steen (2003) define 8 concepts of transparency in a distributed system. These are repeated in Table 1.1 below.

<table>
<thead>
<tr>
<th>Transparency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>Hide differences in data representation and how a resource is used</td>
</tr>
<tr>
<td>Location</td>
<td>Hide where a resource is located</td>
</tr>
<tr>
<td>Migration</td>
<td>Hide that a resource may be moved to another location</td>
</tr>
<tr>
<td>Relocation</td>
<td>Hide that a resource may be moved to another location while in use</td>
</tr>
<tr>
<td>Replication</td>
<td>Hide that a resource is replicated</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Hide that a resource may be shared by several competitive users</td>
</tr>
<tr>
<td>Failure</td>
<td>Hide the failure and recovery of a resource</td>
</tr>
<tr>
<td>Persistence</td>
<td>Hide whether a (software) resource is in memory or on disk</td>
</tr>
</tbody>
</table>

Table 1.1 – types of transparency in a distributed system.

Access transparency is handled by message passing and specifically in MPI by defining platform-independent data types that are used to communicate between individual computers.

As the MPI standard does not define any method of allocating resources Allen and Wilkinson (1999) the concepts of location, migration and relocation are not directly relevant to this system. Also concurrency, replication and persistence are outside the remit of MPI, and as such depend on the underlying system.

Although many implementations of MPI identify machines that may be used by the use of a machine file in which a URL, e.g. cslin111.comp.leeds.ac.uk, may identify machines, Tanenbaum and
van Steen (2003) suggest that the use of URL’s for locating resources provides location and migration transparency.

The MPI standard itself does not specify any sort of fault identification or tolerance, which is major justification for this project. The system cannot achieve any attempt at fault transparency as any failure is fatal to the system. Although there have been many attempts to achieve this in specific implementations and other methods (see section 1.6), this is still regarded by Tanenbaum and van Steen (2003) as “one of the hardest issues in distributed systems and is even impossible when certain apparently realistic assumptions are made” and as such is an intriguing and active area of research. As this is the main emphasis of this project, this topic is covered in more detail in section 1.3.

In conclusion the transparency of MPI is dependant upon the underlying operating system (or distributed system middleware such as the grid architecture (see section 1.4)) and the implementation of the standard, as the standard only defines communication, which Tanenbaum and van Steen (2003) suggest is only one part of a distributed system.

1.2.3 Openness

Tanenbaum and van Steen (2003) define openness in this context as “a system that offers services according to standard rules”. They go on to state that in order to achieve this many systems use interfaces that describe the syntax of the services available. This is similar to the description of the commands in the MPI standard.

This provides the ideas of interoperability and portability, which suggest that different implementations of the same system can co-exist and work together (although this isn’t always the case) and that any program written for the interface/standard can be used on any of the implementations.

MPI is a particularly “open” distributed system due to it’s definition as a standard, which will hopefully allow me to create example programs that will work on any implementation of the standard, hopefully including implementations such as MPICH-G (see section 1.3) in order for them to also work with minor modifications in a grid environment.

1.2.4 Scalability

There appear to be many different definitions of scalability and how it can be measured. Tanenbaum and van Steen (2003) suggest that scalability of a distributed system can be measured along at least three dimensions, which they credit to Neuman (1994).

These are:

- **Size scalability**: The ease at which it is possible to add more users and resources to the system.
- **Geographical scalability**: The distance at which users and resources can be apart.
- **Administrative scalability**: The ease at which it can span many independent administrative organisations.
Within this definition the scalability of MPI is defined by the nature of the underlying workstations, network and software (i.e. the operating system of each workstation and any other middleware that may be being used) that the implementation is using.

Whereas Ferreira et al (2002) suggest that scalability in terms of “Parallel CPU capacity”, is the performance increase (i.e. how many times faster the application is) relative to the increase in number of processors, in which “a perfectly scalable application will, for example, finish 10 times faster if it uses 10 times the number of processors.”

Both of these are valid definitions of scalability in the correct context, though Tanenbaum and van Steen (2003) are more concerned with the scalability of the underlying distributed system whereas Ferreira et al (2002) are concerned about the performance scalability of an individual application. The type of scalability of particular importance to parallel computing is this definition linked to “Parallel CPU capacity” as it is largely concerned with optimal performance in solving very large and complex problems.

Following directly from this, the next section covers the latest development in distributed systems, grid computing, which has the potential to provide massively distributed systems that can hopefully be used for parallel computation, although this scale does raise its own issues.

1.3 Grid computing

1.3.1 What is grid computing

Grid computing is described by Ferreira et al (2002) as “distributed computing taken to the next evolutionary level”. They describe the key aim of grid computing as to create the illusion of a “powerful self managing virtual computer from a large collection of connected heterogeneous systems”. This definition is similar to the basic idea of distributed systems and as such certain types of distributed memory architectures for parallel computing. This would suggest there is an ideal opportunity for utilising such a system for large parallel problems.

The computing community has not neglected this idea and there are currently grid-enabled versions of MPI in development, one of the most prominent being MPICH-G (see section 1.3.2). Although Ferreira et al (2002) do warn that grid computing cannot be made to solve all problems much quicker, or any quicker at all.

Although this is not the only application that grid computing can be used for, the system is designed to provide access to many different resources. These applications can be as relatively simple as distributed storage, which is also called a “data grid” (Ferreira et al, 2002), to the reserving of an electron microscope connected to the grid. In this case reserving refers to the principle of arranging an agreed time period in which the resource can be used exclusively Ferreira et al (2002). The Globus alliance (www.globus.org) are involved in designing and implementing technologies for grid computing. They provide a toolkit for setting up a Grid and an implementation of the MPI 1.1
standard called MPICH-G2 that is Grid compatible, so the programs developed in this project could theoretically be used within a grid environment with only minor alterations.

1.4 Fault Tolerance

Tanenbaum and van Steen (2003) define fault tolerance as the idea of maintaining provision of its defined services in the presence of faults. In the case of parallel computation a service may be performing some mathematical operations on some data in order to provide results.

1.4.1 What is a fault?

Tanenbaum and van Steen (2003) define a fault to be the cause of an error which results in a system failing. In the context of parallel computing, system failure is the either no results or incorrect results. Faults can be classified as one the following:

- **Transient**: A fault that occurs once and then disappears
- **Intermittent**: A fault that can occur and vanish at random intervals
- **Permanent**: A fault that continues to exist until is repaired

1.4.2 Failure Models

Failures can be classified into different types, which Tanenbaum and van Steen (2003) define in table 1.2 (based upon Christian (1991) and Hadzilacos and Toueg (1993)).

<table>
<thead>
<tr>
<th>Type of failure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash failure</td>
<td>A process halts, but is working correctly up until this point</td>
</tr>
<tr>
<td>Omission failure</td>
<td>A process fails to respond to incoming requests</td>
</tr>
<tr>
<td>Receive omission</td>
<td>A process fails to receive incoming messages</td>
</tr>
<tr>
<td>Send omission</td>
<td>A process fails to send messages</td>
</tr>
<tr>
<td>Timing failure</td>
<td>A processes response lies outside a specified interval</td>
</tr>
<tr>
<td>Response failure</td>
<td>A processes response is incorrect</td>
</tr>
<tr>
<td>Value failure</td>
<td>The value of the response is incorrect</td>
</tr>
<tr>
<td>State transition failure</td>
<td>The process deviates from the correct flow of control</td>
</tr>
<tr>
<td>Arbitrary failure</td>
<td>A process may produce arbitrary response at arbitrary intervals</td>
</tr>
</tbody>
</table>

Table 1.2 – Types of failure in distributed systems.

1.4.3 Fault tolerance

Tanenbaum and van Steen (2003) suggest that the best approach a system can take to fault tolerance is to “try to hide the occurrence of failures from other processes“. In this case process may refer to other systems utilising or building upon the system or possibly end users.
Tanenbaum and van Steen (2003) outline 3 different kinds of redundancy, which they credit to Johnson (1995) and are described below (From Tanenbaum and van Steen, 2003).

Information redundancy involves providing extra bits to allow for garbled bits, an example of this would be a Hamming code to “recover from noise on the transmission line” (Tanenbaum and van Steen, 2003). This approach appears to be too low level for use in this particular project, although it is likely that the platform will utilise Hamming codes at a much lower level as this is used in many physical protocols.

Time redundancy is the idea that an action can be performed and then if required can be performed again. An example of this would be the use of Transactions (Chapter 5, Tanenbaum and van Steen (2003)) as if a transaction is aborted then it can be performed again without any problems. This idea is very useful for transient or intermittent failures, but may be outside the scope of this project, as failures may be considered as final.

Physical redundancy is the idea of providing extra equipment or parts of the system to allow for the failure of malfunction of some components. An example of this would be Triple Modular Redundancy (Tanenbaum and van Steen (2003)), which they suggest as being a very general technique that gives a clear idea of what a fault tolerant system is. This type of redundancy may be appropriate in Distributed Systems as availability is probably more important than performance but for parallel applications, performance is often the most important factor and as such this redundancy could be seen as wasted performance.

Tanenbaum and van Steen (2003) go on to discuss the specific subject of process resilience which, although provides a means of protection against process failure, may be deemed as too much of a compromise, as it involves replication of identical processes in groups to provide reliability and so performance is affected. However some of the design issues for these groups appear to be very similar to problems which parallel programs may encounter so they are presented below.

According to Tanenbaum and van Steen (2003) process groups may be organised in either a flat group where all processes are equal and no process has control, or a hierarchical structure where one process is a coordinator of the group. The flat group has the advantage that the group has no single point of failure; if one process fails the group simply becomes smaller but may continue to operate. However this makes coordination and decision making difficult. The hierarchical group is in many ways the opposite; it has a central coordinator which means decisions can be made easily. However this coordinator is a single point of failure where if it fails the entire group fails.

Tanenbaum and van Steen (2003) also present many other distributed system specific issues including reliable client-server communication, the need for reliable point-to-point communication and the need for reliable group communication, via the idea of “reliable multicasting”, which in particular they decide as being “surprisingly tricky”. Although Tanenbaum and van Steen (2003) present these ideas from a distributed systems perspective, it should be clear that many of these ideas...
are also applicable to parallel computing, especially as many systems are a combination of them, though the subject of distributed commit is not directly relevant to this project.

Finally Tanenbaum and van Steen (2003) discuss the idea of recovery from failure, which is an important part of this project. They outline two main ways of recovering from failures which are summarised below.

- **Backward recovery** involves the principle of bringing the “system from its erroneous state following failure back into a previously correct state”, this is achieved by means of storing the state of the system at regular intervals and restoring this state in the event of problems. This process is known as checkpointing and as seen in section 1.5 is a popular method for recovery.

- **Forward recovery** on the other hand, is the principle of attempting to move the system into a new correct state when it enters an erroneous state, however, according to Tanenbaum and van Steen requires prior knowledge of what errors may occur and how to handle them.

### 1.5 Previous work in this area

This area is a current and active area of research and as such large amounts of work have been undertaken. However, as seen below, there are many different approaches to fault tolerance in a distributed parallel system and which do not directly interfere with this project. Below is a small summary and analysis of some of the research in this area.

Bronevetsky and Marques (2003) outline a solution that involves implementing an underlying layer which intercepts all MPI calls and re-routes then through its own middleware. This unfortunately makes the approach platform specific and very inflexible.

Recently Dongarra and Fagg at the University of Tennessee have made a proposal for a new implementation for MPI, known as FT-MPI. The paper (Dongarra and Fagg, 2003) published concerning this proposal describes the semantics and implementation details for this new implementation along with an analysis of currently available check pointing MPI libraries, most of which are based upon other distributed systems such as Condor. The new implementation relies on an underlying distributed system known as HARNESS g_hcore along with numerous optimisations of the underlying networking libraries. The intriguing features of this implementation are the support for dynamic communication worlds and provision for different recovery methods by the programmer.

There appears to be a common theme of developing adaptations or completely new implementations of MPI in order to provide fault tolerance and, although this is designed to hide the complexity from the programmer, it means that most of the systems are inflexible and usually are platform specific or require extra software to be installed. Whereas the approach undertaken in this project is within the program itself, which unfortunately increases the complexity, but will hopefully increase the programmer’s ability and be platform independent (as much as MPI).
Summary

This chapter introduced the concepts of parallel computing and distributed systems, along with their importance in the future of computing and current developments in this area, such as Grid computing. This has raised the interesting problem with the opportunity to utilise large numbers of computers to perform complex calculations, it is necessary to stop the failure of a single computer wasting all of the processing that has been done. This is the concept behind fault tolerance and is the topic of this project.

The next chapter discusses the objectives and minimum requirements of this project along with a plan (and final progress) in order to achieve them. This will lead to an investigation into the characteristics of parallel programs that make them susceptible to faults.
2 Objectives and Planning

Introduction
Following the previous chapter covering an introduction to the parallel computing, distributed systems and fault tolerance, this chapter defines the aim and objectives of this project more clearly, along with the minimum requirements (and possible extensions) needed in order to achieve this. The chapter finishes with the initial plan for the project, a look at the characteristics of distributed parallel computing that affect how fault tolerance can be achieved and the final progress of the project.

2.1 Aim
The aim of this project was to develop a set of example programs, which illustrate the problem of faults occurring in distributed parallel systems and possible fault tolerant versions of these programs.

2.2 Objectives
The objectives required for the project to achieve its aim and possible enhancements that could extend the project, along with a list of minimum requirements linked to these objectives (objectives shown in brackets next to minimum requirement) are outlined below. See section 2.4.2 for justification of the choice of programs developed.

2.2.1 Objectives
1. Familiarity with parallel and distributed systems.
2. Familiarity with the concept of fault tolerance.
3. Determine the characteristics of parallel programs that relate to fault tolerance.
4. Familiarity with MPI.
5. Production of example programs highlighting the characteristics related to fault tolerance and fault tolerant version of these programs.
6. Evaluate the success of this approach

2.2.2 Minimum requirements
- Background reading chapter (1, 2)
- Discussion of characteristics (3)
- Matrix multiplication example programs exhibiting characteristics relating to fault tolerance and fault tolerant versions of these programs (4, 5)
- Evaluation of the example programs along with a conclusion as to whether this is an appropriate way of providing fault tolerance (6)
2.2.3 Possible extensions

- Other example programs showing other characteristics relating to fault tolerance and fault tolerant versions of the programs (4, 5)
- Research alternative methods for fault tolerance in MPI parallel programs (6).

2.3 Methodology

Due to the largely new nature of the subject matter, there was a large emphasis upon background reading and developing understanding of this. The development of the example programs required an iterative approach due to this lack of knowledge and experience in parallel programming and debugging.

2.3.1 Development

This project will involve an iterative development methodology outlined below.

- Define possible solution (and accordingly design for program)
- Implement example program
- Evaluate the effectiveness of the program with respect to fault tolerance (see section 2.3.2)
- Repeat this until satisfied with effectiveness of the solution

In most cases the first iteration of this process will involve a simple program that does not have any fault tolerant features, and by the end of the process the program will be tolerant to certain (possibly all) faults. Once this has been achieved it will be possible to progress onto the next set of programs illustrating different characteristics.

2.3.2 Testing Strategy

Unfortunately, due to the nature of faults, they are not reliably predictable, so in order to test the fault tolerance of my example programs, a means of emulating the occurrence of faults is needed. This can be achieved by one of two methods.

- Artificaly induced: Where a fault is created, e.g. by killing a running process or unplugging a network cable or by inducing failure of processes within the program (possibly based upon probabilities).
- Simulated: Where the program is executed in a simulated environment.

The chosen approach for testing the final programs was a means of artificially inducing faults during the execution of the programs (see section 3.1 for more details).
2.3.3 Evaluation

The criteria used to evaluate the individual programs are shown below.

- **Fault tolerance**: An analysis of the design and implementation of fault tolerance for the example program, highlighting particular limitations and problems.

- **Fault tolerance tests**: Execution of the fault tolerance versions of the example programs with faults occurring during the execution.

- **Development time, difficulty and code size**: A qualitative discussion regarding whether the development time and difficulty of implementing the example programs that are fault tolerant is worthwhile considering the effectiveness of the programs.

- **Theoretical performance**: The theoretical performance of the fault tolerant and original versions of the programs. This is based upon the examples in Wilkinson and Allen (1999) and Foster (1995) but with certain simplifications made and only modelling of some parts of the programs, as the intention is to compare the fault tolerant versions to the original program rather than comparisons to sequential programs or developing an entirely accurate model. The fault tolerance overhead is deemed to be acceptable if the impact is less than the overhead of re-executing the original program with one less process.

- **Performance tests**: The relative performance of the original and fault tolerant versions the program using specific numbers of processes and problem size.

As there is no directly similar research, the only option for comparison is to the research conducted into general lower level solutions to this problem, which is outside of the scope of this project.

Traditionally according to Foster (1995) performance evaluation involves 3 main metrics. The first is the efficiency and speed-up of the parallel code. However this is not relevant to this project as parallel programs are being compared to each other, rather than to a sequential version of a program. The other 2 metrics are according to Foster (1995) quantitative metrics of scalability with a fixed problem size and scalability with a scaled problem size. Though as this project involves the impact of adding fault tolerant features to a program and as Jain (1991) states the need to set goals for performance evaluation is critical, the goal of this evaluation to determine if the cost of adding these features is worthwhile. With this in mind, scalability with a fixed problem size is an appropriate metric for determining the impact of adding the fault tolerant features and whether they can scale with the number of processes executing the program.

Also, the results gained from the performance tests may be affected by external factors such as workload (as highlighted by Feitelson (2002)), due to testing in a public environment, so any conclusions should be regarded with caution. Executing each program 10 times and averaging the results should help to reduce the effects of this, along with attempting to ensure the tests were performed at quiet times with no other users utilising the computers involved.
2.4 Planning

2.4.1 Initial Plan

The initial plan for the projects was approximately the following.

1st Semester
- Background reading into parallel and distributed systems
- Research into MPI and implementation of simple parallel programs
- Determine the characteristics of parallel programs that relate to faults occurring
- Begin development of example programs

2nd Semester
- Further research into fault tolerant methods
- Research into appropriate metrics for performance and scalability evaluation
- Development of example programs
- Further research into Grid computing
- Development of Grid compatible versions of the example programs

As can be seen clearly in the above plan, it lacked enough detail for effective milestones to be defined. In order to do this a greater understanding of the characteristics of parallel programs which affect fault tolerance, was required to choose a suitable set of example programs to develop. The characteristics decided upon are outlined below.

2.4.2 Characteristics analysis

Following on from the research in chapter one, it is possible to determine three main characteristics concerning parallel programs which affect the technique required in order to develop fault tolerance.

2.4.2.1 Centralised or distributed control

The ideas presented below are heavily based upon the idea process groups outlined in section 1.4.3. A parallel program can have a single “master” process, which handles the distribution of data to “slave” processes, which undertake the required calculations and relay the results back to the master program. This design of program allows for the master to determine any faults in a “slave” process and act accordingly. However it exhibits an undesirable feature in distributed system, as the “master” process is a single point of failure. Unfortunately this means that a fault relating to the “master” process is fatal to the computation and must be guarded against, though this is a considerably harder
problem as a “slave” process must have the logic to determine this fault has occurred and react accordingly.

This leads onto the idea of distributed control, where (in the simplest instance) all processes are aware of what data they need and the calculations required and so do not require any controlling “master” process. This removes the single point of failure problem, but introduces extra complexity in detecting and dealing with faults.

2.4.2.2 Separation of data

The degree to which data may be separated defines the amount of inter-process communication required in a parallel program and as such the reliance each process has on each other. This makes handling faults in the system considerably harder as it is necessary to resend\redirect certain communication to the correct process. It may also be necessary to reallocate data to different processes, which is a considerable overhead.

2.4.2.3 Iterative nature of the program

Many parallel programs require repeated computation of data. This may require that all processes synchronise some or all of their data/results with other processes and repeat the required computation. This increases the amount of inter-process computation and introduces the problem of faults occurring during synchronisation, which makes fault tolerance considerably harder. However there are established techniques for doing this, known as check pointing (discussed in section 1.4.3).

2.4.3 Example programs to develop

In order to illustrate the characteristics outlined above the following example programs were originally going to be developed.

- Matrix multiplication using centralised control and work pooling.
- Matrix multiplication with semi distributed and distributed control.
- Area under a curve – both static and dynamic.
- Iterative finite difference for simulating heat distribution.

2.4.4 Extended plan

Following on from this it was possible to develop a more detailed plan for the project, which is shown in table 2.1.

<table>
<thead>
<tr>
<th>Milestone</th>
<th>Expected start (week beginning)</th>
<th>Expected end (week beginning)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background reading</td>
<td>06/10/2003</td>
<td>08/12/2003</td>
</tr>
<tr>
<td>Learn MPI and implement simple programs</td>
<td>06/10/2003</td>
<td>08/12/2003</td>
</tr>
</tbody>
</table>
2.4.5 Problems with the extended plan

The extended plan shown in table 2.1 did not allow time for the progress meeting, writing of the draft chapter, writing of the final project report and evaluating and testing the example programs. Table 2.2 shows the final plan for the project with added milestones and actual progress.

2.4.6 Final progress

<table>
<thead>
<tr>
<th>Milestone</th>
<th>Start (week beginning)</th>
<th>End (week beginning)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background reading</td>
<td>06/10/2003</td>
<td>08/12/2003</td>
</tr>
<tr>
<td>Learn MPI and implement simple programs</td>
<td>06/10/2003</td>
<td>08/12/2003</td>
</tr>
<tr>
<td>Determine the characteristics related to fault tolerance</td>
<td>06/10/2003</td>
<td>08/12/2003</td>
</tr>
<tr>
<td>Matrix multiplication example programs</td>
<td>03/11/2003</td>
<td>26/02/2004</td>
</tr>
<tr>
<td>Iterative example programs</td>
<td>02/02/2004</td>
<td>16/02/2004</td>
</tr>
<tr>
<td>Grid computing research</td>
<td>23/02/2004</td>
<td>15/03/2004</td>
</tr>
<tr>
<td>Grid computing example programs</td>
<td>23/02/2004</td>
<td>15/03/2004</td>
</tr>
</tbody>
</table>

Table 2.2 – final progress of project.

2.4.7 Justification of alterations

Inexperience of the subject matter and the type of project led to initial planning mistakes and slower progress than originally expected.

The difficulty of parallel programming and debugging, along with more complexity than initially envisaged led to the development of the example programs taking much longer than originally
expected. The lack of a debugger and until very late in the project the use of a memory analyser, along with problems with Microsoft Visual C++ (which doesn’t report segmentation faults in all circumstances) used for part of the development also added to this time. However, as more of the issues and characteristics related to parallel programming were covered in the matrix multiplication examples than originally envisaged meant that the area under a curve examples did not need to be developed, as they would add less to the overall scope of the project and the iterative example showed more interesting characterises, so was the best use of the remaining time.

It should be noted that the start and end weeks given for the example programs show the time when the majority of the design, development and testing of these occurred; however, due to the iterative nature of the project and an inexperience in parallel programming, debugging and testing, this process continued until very late into the project.

Summary

This chapter outlines the aim, objectives and minimum requirements of the project, followed by an explanation of the evaluation strategy, initial plan, characteristics related to fault tolerance (in order to complete the original plan) and an extended original plan, finishing with an outline and discussion of the final progress of the project.

The next chapter discusses the design, implementation and evaluation of the matrix multiplication example programs.
3 Matrix multiplication example programs

Introduction

Initially this chapter describes the matrix multiplication algorithm used throughout these examples. The rest of the chapter describes, for each of the example models, the initial design and implementation, followed by the design and implementation of fault tolerance, concluding with an analysis and evaluation of the performance and effectiveness of the fault tolerant features for each. Also it should be noted that many of the terms and MPI functions used from this point on are described in appendix B.

3.1 Matrix multiplication algorithm

The matrix multiplication algorithm used is simply the standard method for multiplying two matrices together (e.g. Lipschutz and Lipson (1997)) or any other good discrete maths book). The algorithm presented here for matrix multiplication is not intended to be complicated or an example of the most efficient algorithm; it is intended as a basis of the demonstration of three specific models used in parallel programming and how fault tolerance may be added in general to programs using these paradigms.

In all of the example programs, the matrices are stored as 1-dimensional arrays, and the ordering is represented by appending row or column onto the naming convention e.g. matrixARow. The first matrix is referred to as A, the second as B and the matrix computed by multiplying the two together C.

3.2 Master slave centralised work farm model

This section provides a simple example of the master slave model using a work farm distribution model.

3.2.1 Other applications

The master slave model for parallel computation is probably the most obvious and simple model and has been used extensively in many areas such as using a genetic algorithm for ground water remediation design (see Babbar and Minsker (2002)).

3.2.2 Master slave model

In a master slave parallel program, a single process is referred to as the master (also known as root or controller) process which controls the distribution and collection of work (usually referred to as jobs) to all of the other processes. These other processes are referred to as slaves (or workers) and all perform the same calculations on different data. This data can be determined in many different ways.
e.g. block decomposition (see sections 3.4 and 3.5) or, as in this example using a distribution model. In some cases, but not here, the master may also act as a slave, performing the same calculations on part of the data, though this has little effect on the fault tolerance of the solution, other than the possibility of it tolerating an extra failure. This model is similar to the hierarchical process group outlined in section 1.4.3.

### 3.2.3 Work farm distribution model

In this case, each job involves distributing a request for a column of matrix C to slave processes for computation. Initially all slave processes are allocated a column of the matrix, starting at 0. This request is simply an integer value, as all slave processes already have access to the entire contents of both matrices (In this case the first matrix is generated on each process and the second is generated on the master and broadcast to every slave process, though this could easily be adapted to reading either matrix from a file or similar. Once a slave completes this work and the result is sent back to the master, the master then allocates the next column in the matrix, on a first come first serve basis. This also allows for simple load balancing as the slave processes which perform the calculations the fastest will be allocated another column (if possible) in preference to other slower slave processes.

The exact details of this algorithm are less important than developing a program that exhibits typical communication patterns for this model. A diagram demonstrating the program is shown in figure 3.1.

![Figure 3.1 – Master slave program using a work farm distribution model](image)

### 3.2.4 Key features of the original parallel program

The program approximately follows the structure below

- Master process broadcasts the matrix size to all other processes.
- All processes generate matrix A (Could be adapted to read from a file accessible by every process.).
- Master process generates matrix B (Could be adapted similarly to matrix A).
- Master process broadcasts matrix B to every slave process.
- Master process initially allocates columns of matrix C to be computed, until there are no more slave processes that haven’t been allocated a column or all columns of matrix C have been allocated.
- Slaves compute allocated column of matrix C and send results back to master.
- Master waits to receive columns of matrix C. Upon receipt of a column, it sends a request for the next column in C to be computed to the slave it received the last column from.
- This is repeated until all columns of matrix C have been calculated upon which the master sends a “completed” message to every slave (allowing them to exit), outputs matrix C and exits.

This program can be seen in its entirety in the accompanying file MasterSlave.cc.

3.2.5 Implementation of the original program

This section discusses implementing certain parts of this program in more detail, with reference to possible features relevant to the design and implementation of fault tolerance in the program.

3.2.5.1 Implementation of work farm distribution

This is implemented with a counter for the current column to be distributed to the slave processes. Once this counter exceeds the size of the matrix, the master can stop distributing column requests. Along with a boolean value for each column (stored in an array) which records whether the master has received a calculated column, once all of these are true the master can send the “completed” message to each slave process. When the master sends a request to a slave process it stores which column has been requested against that process in a 1-dimensional array, so that when the result is received from the slave, the master knows the correct position for the column in matrix C.

3.2.5.2 Blocking communication

All communication in the program, both collective and point-to-point communication is blocking i.e. all functions only return when the operation has completed, so failure of any process involved will result in all processes halting indefinitely (apart from point-to-point sending which may not, depending on the implementation, see section 3.2 6.1).

There are numerous instances of blocking point-to-point communication that must be adapted to be sure of fault tolerance.

There are two instances of collective communication in this program; both are simple broadcast operations involving all processes, one for the size of the matrix and one for matrixB. Using
MPI’s built in functions for collective communication is important as it provides the developer with a
consistent, concise and probably optimised method of performing this collective communications e.g.
section 8.7 in Foster, (1995) shows the significant performance gains that can be made over using
individual point-to-point communication under certain circumstances. Though these functions require
all processes involved to be working and as such are intolerant to fault tolerance.

3.2.5.3 Code evolution

Initially the original program was implemented in a single large function, using decision
statements for dividing logic between master and slave logic. It soon became apparent that this method
would not provide a program that could be easily understood or modified to add fault tolerance. Later
versions of the program utilise separate functions for the master and slave logic. This has the distinct
advantages that the division of logic was always clear and could easily be independently changed,
though it meant care should be taken with collective communications, e.g. MPI_Bcast function calls
which must be nearly identical on each process in order to operate as expected and also made noticing
discrepancies between any of the communication difficult.

3.2.6 Fault tolerance design and implementation

The design and implementation sections for this example have been combined due to the
iterative nature of development largely due to unfamiliarity with the MPI standard, parallel
programming, debugging and fault tolerance.

There are two core features that a fault tolerant program must exhibit, all communication
between processes must be non-blocking or only involve working processes and individual processes
must be able to determine and recover from faults occurring in other processes.

3.2.6.1 Non-blocking point-to-point communication

All (or most depending upon the level of fault tolerance) point-to-point communication in the
original program must be replaced with the non-blocking equivalent; this is usually done with a pair of
MPI function calls.

For receiving messages this is necessary as, if the sending process fails before it can send the
data, then the receiving process will wait indefinitely for the data and effectively stop responding. This
can be alleviated by replacing the blocking MPI_Recv by an MPI_Irecv which posts the receive but
does not block until it completes. An accompanying MPI_Test function call tests for the completion of
a posted receive (using a request object) and sets a flag to true if the receive has completed. This can
be used within a while loop to continually check for messages until a particular condition occurs e.g. a
‘complete’ message has been received, a threshold has been reached or a timeout has occurred. When
a message is received the timer is reset.
For sending messages, the semantics of completion may differ between implementations; either the function call may complete when the sending data has been copied to a buffer if buffering is used; otherwise it completes when a corresponding receive function call is made on the destination process. In the second case, similarly to receiving data, if the receiving process fails before it can receive the data then the sending process will wait indefinitely for the data to be received at the destination and effectively stop responding. In this example the completion of the send operation is unimportant (as it is assumed that if the destination process is still alive then it will receive the data) and so rather than use an MPI_Test or MPI_Wait, the request can be freed without checking. This provides for greater flexibility and performance of performing multiple sends, however if the implementation uses buffering then there is a small possibility that the buffer may be overwritten before completion, this is alleviated by having a buffer for every process (implemented using a simple array) as only a single message is sent from the master to any particular slave at any one time.

3.2.6.2 Collective communication

As the only collective communication occurs near the beginning of the program, it is assumed that no process will fail until after these have completed, so no provision has been made to make these parts of the program fault tolerant (however this is attempted in later examples, see section 3.3).

3.2.6.3 Determining and recovering from worker failure

The first key feature of being able to handle worker fault tolerance is for all the master processes send and receives must be converted to non-blocking.

Originally it was intended to monitor each slave process explicitly, using non-blocking receives and a timeout on the master process to determine that a particular process has failed before a particular requested column has not been calculated and sent back to the master. This would indicate that the process has either failed or is performing too slowly (both may be classed as faults see section 1.4) and resend the column request to another slave process.

This requires a certain amount of computational and time overhead. A simpler and more elegant solution, which the final program uses, is to keep requesting columns to each worker process until all of the columns have been requested once and then start re-requesting any columns that haven’t been sent back to the master from a slave process until all columns have been received. This does mean that some columns near the end of the matrix may be requested more than once without it being necessary, but as only processes that have completed their column request are used this does not adversely affect performance.
3.2.6.4 Determining and recovering from master failure

Originally, a slave process determined the possible failure of the master process by waiting for a timeout to occur whilst waiting for a response (either a column request or a 'complete' message) after sending a calculated column back to the master. If no message arrived before the timeout, the slave explicitly queried the master by means of a query and response pair of messages. If this is still unsuccessful then the slave concluded that the master has failed and begins the recovery process. In later versions, a slave process does not explicitly query the master affect a timeout has occurred, it begins the recovery process immediately as there is an unnecessary overhead and delay associated in explicitly contacting the master process.

This process is only possible as the slave process uses non-blocking communication (see section 3.2.6.1 for a detailed discussion of how this is implemented).

The single most important part about recovering from failure of a master process is the election of a replacement process from the slave processes still alive. Tanenbaum and Van Steen (2003) outline two widely used algorithms that are summarised and evaluate below.

In the bully algorithm, the process (or processes) that determined the master has failed, sends a message to all processes with rank below it (or any other agreed numbering) to query if they are still active. If no one responds then it wins the election and takes over as master. If any process responds then the original process takes no further part in the election and the responding process (or processes) initiate their own election until only one process remains, which takes over as master.

Whereas in the ring algorithm, the process (or processes) that determined the master has failed sends a message to all processes with rank below it (or any other agreed numbering) to query if they are still active. If no one responds then it wins the election and takes over as master. If any process responds then the original process takes no further part in the election and the responding process (or processes) initiate their own election until only one process remains, which takes over as master.

For this example the simpler bully algorithm is more appropriate as the extra information that the ring algorithm determines regarding the state of all processes is not required and as such this extra overhead is not necessary. Once a new master process has been elected the algorithm has to be restarted and all columns have to be re-requested from the remaining worker processes.

Unfortunately it was not possible to integrate the bully election algorithm into the example programs, it works independently to the example program (and is provided in the code) but causes an MPI error when utilised from within the program. To accommodate this a specific slave process takes over as the master process and the others to continue as normal, though this limits the tolerance to just a single master failure.
3.2.7 Evaluation

3.2.7.1 Fault tolerance

The final version of this program can tolerate up to N-2 slave failures (where N is the total number of processes) as there must always be a master and at least one worker process, or 1 master failure and up to N-3 slave failures (with the specified process that takes over from the master not failing due to problems encountered implementing the election algorithm). This is in itself quite limited but could be quite easily expanded to cope with up to N-2 slave or master failures if the election algorithm could be successfully integrated or using a new communicator every time the master fails (see sections 3.3.5 and 3.3.7 for more details about communicators). Although if the master process is also a slave then it would be possible to increase the fault tolerance by 1.

The example programs provide good fault tolerance transparency, as the user is unaware that any problems have occurred, apart from some performance degradation that anything has failed. Unfortunately this only holds for the limits outlined above, if these limits are exceeded then the program will either run continuously or fail completely.

A particularly important issue in developing and testing the fault tolerance for this program is the importance of deciding an appropriate value for the time out of messages sent from the master to the slave for determining master failure (this is discussed in more detail in section 3.3). It should also be noted however, that in order for the example programs (including some of the later examples) to handle failure of the machine initiating the execution of the mpi-run command, all output should probably be written to files rather than using standard output as this is forwarded to this machine.

Both fault tolerant versions of the program accompany this project, MasterSlaveSlave.cc provides only slave fault tolerance and MasterSlaveMaster.cc provides both master and slave failure.

3.2.7.2 Fault testing

All of the details of MPI functions and implementations mentioned below are based upon content in Snir et al (1996). During the development process, faults were induced by prematurely finishing a specified process at particular stages in the program. This was done with a call to the MPI_Finalize function (usually used to end the use of MPI in a program) and then exiting the program. However this cleaned up the internal state of MPI and so did not accurately model real faults occurring. When a process was exited via code without using the MPI_Finalize function, which more accurately modelled faults occurring, an MPI error occurred and all processes running the program were terminated. This was due to the default behaviour of MPI to treat errors as fatal. It was possible to rectify this in the final version of the programs (including later examples) by a call to the MPI_Errhandler_set function using the parameter MPI_ERRORS_RETURN. This informs the MPI implementation to attempt to deal with any errors and report them to the running program. However
the errors that can occur are implementation specific and the MPI standard does not define whether an implementation should be able to continue performing after an error. This limits the effectiveness of the example programs to the ability of the underlying MPI implementation to handle lower level problems and continue. Also if a function has failed with performing an MPI_Finalize, the other processes involved will stop responding when they reach the MPI_Finalize function call used to exit normally as it attempts to cleanly finish the MPI of all the processes involved, replacing this with the MPI_Abort function which only attempts to make a “best attempt to abort all tasks in the group” (Snir et al, 1996). However this does affect the transparency of the fault tolerance, in this and all subsequent examples, as an error message is reported at the end of the program. It was not possible to confirm whether these altered example programs can handle ‘real’ faults due to the problems outlined below.

The other means of testing the software involved terminating an individual process from the command line, using the kill command. However the MPI daemon running on the Linux machines in the School of Computing appeared to forward this command to all processes involved in that execution of the program, so this method did not allow for testing. It was not possible to test the programs by restarting machines during execution, so it was not possible to conclusively determine whether the example programs (including later examples) could handle ‘real’ faults.

3.2.7.3 Development time, difficulty and code size

The code for the final version of this program has expanded by approximately 100 lines, which is a significant proportion of the original program, which is approximately 300 lines and has been through many iterations during the development process. The development time increased substantially due to problems with C\C++ dynamic memory allocation, MPI and the considerable difficulty of debugging problems in parallel programs.

3.2.7.4 Theoretical performance

Due to the implicit nature of slave failure tolerance in this system, the model for slave failure is relatively simple. Assuming only a single slave process fails, then only a single row of matrix C needs to be re-requested. This is clearly less than re-executing the original program, with N-1 (where N is the number of processes originally used) processes.

In the case of master failure, the entire matrix multiplication must be restarted with N-1 processes, so this is on a par with manually re-executing the original program with N-1 process; the only advantage is that this process is initiated automatically.

3.2.7.5 Performance Tests

These, and all subsequent tests, tests were performed on 2, 4, 8 and 16 processes running on separate machines in the level 10 lab at the University of Leeds. The code was compiled with the mpiCC command which uses g++ version 3.2.2.20030222 and executed using MPICH version 1.2.5
release 2003/01/13. A matrix of 512 x 512 values was chosen as this appeared reasonably large to accommodate the effects of scale and a timeout value (for the master version) of 0.25 seconds, though this shouldn’t affect the program when no failure occurs. The results are shown in figure 3.2

![Figure 3.2 - Performance test results for master slave example programs.](image)

The results clearly show that the impact of the fault tolerance (with no failures) is very small, as there is no explicit communication and very little extra processing, and may in fact have a positive effect on performance due to the use of non-blocking communication. However the upward curve of the graph between 8 and 16 processes is surprising.

### 3.2.8 Summary

These example programs demonstrate that it is possible to implement fault tolerance for a simple master slave program at the application level using MPI with very little performance impact if no or only slave failures occur, although it does have some considerable limitations, such as the number of tolerable failures and the position in the program execution at which these failures occur. A key issue in this program is that collective communications are not fault tolerant. However as they only occur twice near the beginning of the program, the likelihood of failures is small. This may be much more important in other programs, as can be seen in section 3.3. The cost of recovery from master failure means it may not be worth implementing on larger programs, alternatively just detect and report the failure and exit as cleanly as possible on all still functioning processes.
3.3 Semi-distributed block decomposition model

This section provides an example of the semi-distributed block decomposition model.

3.3.1 Other applications

This and the following example (section 3.4) are both considered part of a larger class of models referred to as partitioning and divide and conquer models (Wilkinson and Allen (1999)). There are numerous examples of these models being used for scientific computation including numerical integration (deDoncker et al (2000)) and physical simulations such as gravitational n-body problems (see Nils Dorband et al (2003)).

3.3.2 Semi distributed model

In a semi-distributed parallel program, all processes perform an approximately equal (or proportional, if difference performance of machines running the processes is known) amount of the work involved. The only difference between the processes is that one process (known as the root, all others are referred to as children) is responsible for distributing the initial data and collecting the final results for outputting; other than that no process has control over another and in many cases little communication occurs between processes. This model and the distributed model discussed in section 3.4 are both very similar to the idea of flat process groups discussed in section 1.4.3.

3.3.3 Block decomposition model

This is a method of decomposing the work involved in a particular algorithm between processes in approximately equal continuous blocks of initial data for processing. In the example program described here, the root process distributes an equal number of columns from matrix B (in this program the blocks must all be of the same size) to itself and all the other processes for multiplication with matrix A (which every process generates) for the corresponding columns in matrix C. This program is shown in figure 3.3.
3.3.4 **Key features of the original parallel program**

The program approximately follows the structure below.

- Root process broadcasts the matrix size to all processes.
- All processes generate matrix A (could be replaced with being read from a file or input).
- Root process generates matrix B (could be altered similarly to above).
- Root process equally distributes columns of matrix B to itself and all other processes.
- All processes calculate their part of matrix C.
- Root process collects the results from itself and all other process.
- Root process outputs the final results.

3.3.5 **Implementation of the original program**

This section discusses implementing certain parts of the original program in more detail, including reference to any features possibly relevant to the design and implementation of fault tolerance in the program.
3.3.5.1 Distribution of matrix B

This is achieved via a call to the MPI function MPI_Scatter, which distributes contiguous data such as the contents of a 1-dimensional array (or defined MPI type for accommodating structures and non-contiguous data) on a particular process in equal parts to all processes in an MPI communicator. The exact function call is shown below.

\[
\text{MPI\_Scatter(matrixBCol, (matrixSize * colsPerProcess), MPI\_INT, tempCols,}
\text{(matrixSize * colsPerProcess), MPI\_INT, 0, MPI\_COMM\_WORLD);}\
\]

This scatters the contents of matrixBCol in blocks of size (matrixSize * colsPerProcess) from process 0 to all processes in the communicator MPI_COMM_WORLD (including 0).

All communication in MPI is conducted within (or between, but this is not covered here) communicators. There is a single defined communicator (excluding communicators for internal MPI use or associated libraries etc) when an MPI program is executed; it contains all processes involved in this execution of the program with ranks starting from 0 to the number of processes - 1. The ranks given to processes are relative to the particular communicator that is being used at the time.

As mentioned in section 3.2 collective communication is intolerant to failure of any processes involved and as such is particularly relevant to the design and implementation of fault tolerance for this example.

3.3.5.2 Collection of parts of matrix C

This is very similar to the distribution of matrix B, but in reverse. This is achieved with a call to the MPI function MPI_Gather, which gathers contiguous data of equal size e.g. a 1-dimensional array on every process into a buffer on a specified process. The exact function call is shown below.

\[
\text{MPI\_Gather(tempResults, (matrixSize * colsPerProcess), MPI\_INT, result,}
\text{(matrixSize * colsPerProcess), MPI\_INT, 0, MPI\_COMM\_WORLD);}\
\]

This gathers the contents of the 1-dimensional array tempResults of size matrixSize * colsPerProcess on every process in the communicator MPI_COMM_WORLD into the 1-dimensional array result on process 0.

As with the scatter operation (section 3.3.5.1) this is particularly important regarding fault tolerance in this example.

3.3.6 Code evolution

The code for this example is quite short at approximately 200 lines and development was quite quick once the MPI functions used above for collective communication were understood.
3.3.7 Fault tolerance design

The core feature of this program that needs to accommodate faults occurring is the 3 occurrences of collective communication. In this case the broadcast of the matrix size and the scattering of initial data occur very close to the beginning of the program and as such it is assumed that any failures occur after these operations and so no attempt has been made to make them fault tolerant. This is because the likelihood of processes failing this close to the beginning of the program is very small (as they must have been working at the start of execution of the program) and the little benefit over restarting the program.

In order for the final collective communication to complete successfully, all processes involved in the operation must be working, so the key element of fault tolerance for this example is determining if any processes have failed and excluding them from the collective communication. This will result in some columns (blocks) of matrix C (stored in the result array in the original program) to be missing upon completion of the collective communication, so some means of recalculating these columns is required.

3.3.7.1 Determining process failure

In order to determine if any processes have failed involves synchronisation between all of the processes; no collective MPI functions e.g. MPI_Barrier can be used to do this due to their intolerance to failure. The original design for this involved a particular process, in this case the root, sending a query message to every process except itself, then using MPI_Test within a while loop (similar to master slave example program, section 3.2) to receive a response message from the other processes, storing that the process is working when a response is received, until a timeout occurs. At this point the root process checks to see if any process failed to send a response message, then sends a confirmation message to every process indicating if any processes have failed.

Any process, except the root, determines that the root has failed by using a similar time out method to the root; but for messages to be received from the root, if either of the two messages sent from the root fails to arrive before the timeout occurs then the root is deemed to have failed.

3.3.7.2 Excluding processes from the collective communication

As MPI collective communication functions involve all processes in a communicator, it is necessary to be able to create a communicator with all processes except those that failed. Unfortunately the creation of communicators involves explicit communication with all processes involved in the original communicator and as such will not complete if any processes have failed before or during the creation of a communicator, so any communicator to be used for collective communication must be generated when all processes are working which, in this example, is most likely at the beginning of the program.
Each generated communicator is referred to by a unique value (an integer in MPICH implementation) on each process, which must be stored on each process to be used. If a communicator is generated for every possible combination of processes this would require large amounts of storage and does not scale particularly well with the number of processes running a program; so instead the number of failures is restricted to 1 and so only require the generation of one communicator per process running the program, in order to demonstrate the principle.

Communicator generation involves creating a group (using MPI_Comm_group function) based upon an existing communicator, in this case MPI_COMM_WORLD, then excluding a set of processes from the group (in this case just a single process) and creating a new communicator based upon this group (using MPI_Comm_create function).

Also as the collective communication will involve 1 less process if a failure occurs, the data gathered by the root process needs to be correctly positioned in the result array (matrix C) to accommodate the failure.

3.3.7.3 Calculating missing columns of matrix C

As the design of the program can only handle one failure, one possible method of calculating the missing columns is to repeat the algorithm of the original program dividing the missing block equally between the remaining processes. As the missing block may not divide equally between the remaining processes, this is done by increasing the portion of the matrix C to be calculated until this can be divided equally between the remaining processes, meaning that some columns may be recalculated, this then needs to be gathered by the root and correctly positioned in matrix C.

Due to the increased complexity of the program, the calculation of missing columns is not designed to be tolerant to any faults and as such the program is limited to a single failure in total (though possible means of increasing this are discussed in section 3.3.9.1).

3.3.8 Implementation discussion

Implementation of the fault tolerance in this example partly overlapped the design and undertook an iterative approach, as many of the issues mentioned in the design were only discovered by development of smaller programs testing certain features such as communicator generation, integrating this into the main program and fixing any bugs and problems.

Also the synchronisation method is simplified in the later versions of the fault tolerant programs, removing the initial query message from the root to all processes, as this is an unnecessary message; but the time out value has to be altered accordingly to be able to handle any discrepancies between the performance of individual processes, which would not be as important in the original design as the root sending a message to initiate the synchronisation reduces any possible performance differences between child process. During later development and testing of the distributed example the
usage of the timeout value is altered slightly within the synchronisation of processes (see section 3.4). This change has also been applied here.

Both fault tolerant versions of the this program accompany this report in the files SemiDistChild.cc handling a single failure of all processes except the root and SemiDistRoot.cc handling a single failure of any process, though the differences in these programs are minor due to the recovery process being well suited to any process failure.

3.3.9 Evaluation

3.3.9.1 Fault tolerance

As mentioned in section 3.3.7.2, the number of failures has been deliberately limited to 1 process, due to the problems associated with the creation of communicators to be used in collective communication and in order to demonstrate that fault tolerance is possible in this type of application.

There are numerous places in the program where faults cannot be handled. If a fault occurs before or during either the initial communicator generation, broadcast or scatter then all processes will block indefinitely during the corresponding operation. Also there is a small chance that a process may fail after the synchronisation has occurred but before the completion of the gather operation, resulting in all processes blocking indefinitely. These gaps could be reduced by synchronising more often, although this would add considerable (and possibly unwarranted) overhead.

However the program does provide good transparency and robustness within these constraints as the user is unaware if a single failure occurs, except some performance degradation, and fails robustly if more than one detected failure occur.

The fault tolerance could be improved to handle more than 1 failure per iteration of the program, as currently there are only two possible iterations, the initial iteration and the iteration to recalculate missing columns. If more iterations were allowed, until all only one working process remained, the main alteration would be to generate a new set of communicators based upon the one used in the previous iteration, as all processes in that communicator were working. Although a failure could potentially occur in similar circumstances in this later iteration to the first iteration, resulting in all the processes blocking indefinitely.

The program could also be adapted to handle a certain number of failures per iteration, by generating more communicators to accommodate this failure, however this would also require an alteration of the synchronisation process to return a list (possibly implemented as a dynamic array output parameter) of processes that have failed. It would still be difficult to handle a failure of the root process and a number of other processes. This may be covered by performing extra synchronisations, which may not be possible as it may be necessary to generate communicators to do this. Alternatively an election algorithm such as the ring algorithm (see section 3.2) would report all of the current working processes and if these processes have a communicator already generated for them (depending
on how many combinations are covered by the generation of communicators) then it would be possible to proceed if not fail gracefully. However it should be noted that a large number of failures in a single iteration of the algorithm should be unlikely, though this is dependant upon the reliability of the underlying architecture and the scale of the problem being solved. If a large number of failures occur then it may be preferable for the program to fail gracefully and report these failures than attempt to continue. However, any process taking over as the root must have access to the entire contents of matrix B.

If either or both of these alternatives are implemented, then the method for calculating the missing columns may have to be adapted. If more than one failure per iteration is tolerated then these may occur in non-continuous sequences, so the current method will not operate correctly, unless all columns in between are also recalcuated; but this will have a substantial performance cost.

A better approach may be to divide the missing block(s) into unequal sizes between processes; this would be possible by using the information about failed processes and removes the need for padding the recalculation with columns that are not required. This is made easier by the provision of the MPI functions, MPI_Scatterv and MPI_Gatherv, functions that provide support for different sizes scattered/gathered blocks (see chapter 4 and appendix B for more details).

It is also important to mention that with synchronisation algorithms that rely on a timeout value, deciding upon an appropriate value is both essential to avoid ‘false failures’ (where a process is deemed to have failed, where in fact it is just slower than other processes) occurring if too small and wasted time if too large, but this is not a trivial task and is dependant upon many factors of the particular program and of the underlying architecture.

### 3.3.9.2 Development time, difficulty and code size

This example probably took the longest development time due to the unfamiliarity of the program model and the increased difficulty (and corresponding amounts of research) of the fault tolerance. These factors also make the code probably the least well organised and could be streamlined considerably (although the operation and communications would remain the same).

### 3.3.9.3 Theoretical performance

This example provides a much more complicated and explicit example of fault tolerance, thus the analysis of the impact is more difficult and so will require a model of the appropriate parts of the program. In the model presented below, N is the number of processes the program was originally executed with and M is the size of the matrix (M rows by M columns). The model ignores any time needed for other computation e.g. correctly positioning the gathered data.

The original program requires broadcast and scatter operations, followed by the computation of their parts of the matrix by each process and a gather to collate the results. It is assumed that all collective operations require N messages and the computation of M/N rows on each process takes the
same amount of time on every process. So the execution of the original program requires the following amount of time.

\[ T_{\text{msg}} = \text{Time for sending a single message (e.g. 1 integer).} \]
\[ T_{\text{bcast}} = \text{Time for performing the broadcast operation (N } T_{\text{msg}} \text{ at most).} \]
\[ T_{\text{scatter}} = \text{Time for performing the scatter operation (N } (M/N) \text{ } T_{\text{msg}} \text{ at most).} \]
\[ T_{\text{gather}} = \text{Time for performing a gather operation (N } (M/N) \text{ } T_{\text{msg}} \text{ at most).} \]
\[ T_{\text{comp}} = \text{Time for performing the computation of } M/N \text{ rows of matrix C.} \]

\[ T_{\text{orig}} = T_{\text{bcast}} + T_{\text{scatter}} + T_{\text{comp}} + T_{\text{gather}} \]

In all cases of execution for the final fault tolerant program, there is always the extra overhead of synchronising the processes. This requires 2(N-1) messages if no failures occur, (N-2) messages within the timeout time on the root process, and N-2 messages confirming the failure to all functioning processes, if a single non-root process fails, and (N-1) messages and the occurrence of a timeout on each non-root process, if the root process fails. In the case of a timeout occurring it is assumed that this time encompasses the time taken for sending the other messages.

The recovery process for calculating the missing rows in the case of a failure requires at most 2(M/N)-1 rows due to padding, which require calculating and gathering to the root of the functioning processes. The time for performing the second scatter, computation and gather are related to the times for these operations in the original program but with a much smaller problem size. So for the fault tolerant version of the program to be successful, the following inequality must hold. Use of the variables from the original program should be adapted for N = N-1.

\[ T_{\text{timeout}} = \text{Time out value used for synchronisation.} \]
\[ T_{\text{scatter2}} = \text{Time for performing the second scatter operation (2(M/N)-1 } T_{\text{msg}} \text{ at most).} \]
\[ T_{\text{gather2}} = \text{Time for performing a second gather operation (2(M/N)-1 } T_{\text{msg}} \text{ at most).} \]
\[ T_{\text{comp2}} = \text{Time for performing the computation of } (2(M/N)-1)/N-1 \text{ rows of matrix C.} \]

\[ (N \ T_{\text{timeout}}) + T_{\text{scatter2}} + T_{\text{comp2}} + T_{\text{gather2}} < T_{\text{orig}}(N-1) \text{ for non-root failure} \]
\[ (2N \ T_{\text{timeout}}) + T_{\text{scatter2}} + T_{\text{comp2}} + T_{\text{gather2}} < T_{\text{orig}}(N-1) \text{ for root failure} \]

As the second collective operations and computation are considerably smaller than those needed for re-execution of the original program, the dominant value in this inequality is the timeout value chosen, which must be a compromise between detecting the faults and using more time than re-execution would require.
3.3.9.4 Performance testing

The tests were performed in a similar manner to the previous example (see section 3.2). Figure 3.4 shows the results of the tests.

![Performance test results for the semi-distributed example programs.](image)

These results show that the impact of synchronisation on the overall performance is small; however there are signs that this impact grows with the number of processes so may not be scalable to large number of processes.

3.3.10 Summary

This section has shown that it is possible to provide tolerance for one process failure for this type of problem relatively easily once the issues and development environment are understood sufficiently, but it is a much harder problem to handle more than 1 failure adequately. Also some key issues are raised concerning the synchronisation of processes such as timeouts and problems in detecting if the first few processes fail.

3.4 Distributed block decomposition model

This section provides an example of a distributed block decomposition program, although it is very similar to the previous in many ways, there are a few features that alter the design and implementation of fault tolerance.

3.4.1 Distributed model

This is a natural progression from centralised control (master slave, section 3.2), through semi-distributed (section 3.3) to this model which involves all processes performing the same
operations on different data, known traditionally as Single Program Multiple Data (SPMD) according to Wilkinson and Allen (1999), with no process having any control over any other. In this example there is also no communication between processes involved, but this may not always be the case.

### 3.4.2 Block decomposition model

Although the decomposition model is the same as in the previous example (see section 3.3.3), there is no communication between any of the processes and as such all processes need access to the two matrices independently. In this example they are generated in their entirety by each process, but this could be adapted to read all or just the required parts from a file (or similar). It should also be noted that, similarly to the previous example, the matrix must be equally divisible by the number of processes and matrix C is divided by row instead of by column; but this has only a small effect on the program and no effect on the fault tolerance design or implementation. Figure 3.5 demonstrates this program.

![Figure 3.5 – Distributed block decomposition program.](image)

### 3.4.3 Key features of the basic parallel program

The program approximately follows the structure below.

- All processes generate the entirety of both matrices (though either or both could be read from a file).
- All processes calculate their part of matrix C.
- All processes write their part of matrix C to a file named “ResultRow[processrank].txt”

### 3.4.4 Implementing the original program

Implementation of the original program was relatively simple compared to the two previous examples, though this is probably due to the similarity with the previous example and lack of communication involved.
3.4.5 Fault tolerance approach

As there is no communication involved in the program, there appears to be no need to generate communicators and synchronise between processes to allow for collective communication or any need for any adaptation of blocking to non-blocking communication. However, with no communication there is currently no way of determining if any process has failed, and as such the program will just fail to calculate and output the part of matrix C where a process fails. For fault tolerance to be implemented for this program there is a need to determine which processes have failed and recover from these failures if possible. This leads to utilising many of the fault tolerant features of the previous example (section 3.3). There is a need to determine which processes have failed and to calculate the missing rows of matrix C and output them.

3.4.5.1 Determining process failure

This can be achieved in a very similar manner to the previous example, but should occur after outputting the results to a file, so that the program can determine which process or processes have failed at the end of the algorithm and attempt to calculate the missing rows.

3.4.5.2 Recovering from process failure

Calculation of the missing rows can be done in a very similar manner to the previous example but, as all processes have access to the entirety of both matrices, no scattering of data is required and the missing rows and amount of padding needed can be determined dependant upon the process that has failed (from the result of synchronisation).

In order to maintain fault transparency by outputting the correct file there is a need to gather the recalculated data to a single process in order to write to a file. This unfortunately imposes the same limits as in the previous example as communicators need to be generated for collective communication and so the program is only intended to tolerate a single process failure.

3.4.6 Implementation discussion

The most important alteration to the code that occurred during development and testing is the alteration of the use of the timeout value for synchronisation. The timeout value has been decreased considerably but is now multiplied by the number of processes running in the current execution of the program, meaning it is more scalable to increases in processes. Also for child processes timing out testing for a response from the root process is double that of the timeout for the root timeout, as when they were equal it was possible that some child processes would falsely determine that the root process in synchronisation had failed which either caused erroneous results or the program halting indefinitely.

Also the root process’ checking procedure has been refined slightly by defining an explicit receive for each process’ synchronisation message rather than a more general receive that could receive any of the processes responses, but had to reset after ever message. This was accompanied
with an MPI_Testany to replace the MPI_Test, which allows for testing all the receives set up at the beginning of a synchronisation in one function call, allowing for internal optimisation by MPI and a slight improvement in readability.

Both fault tolerant versions of this program accompany this report in the files DistributedChild.cc handling a single failure of all processes except the root and DistributedRoot.cc handling a single failure of any process, though the differences in these programs are minor due to the recovery process being well suited to any process failure.

3.4.7 Evaluation

3.4.7.1 Fault tolerance

Many of the limitations and problems that affect the semi-distributed example programs also affect this example and as such the program is only tolerant to one process failure, but may be expanded and adapted in the similar ways to those described for that example, with the only difference being the output of files rather than the collection of the entirety of matrix C in a single array to be outputted.

Although these example programs demonstrate that fault tolerance can be implemented in a distributed model, it does involve making compromises regarding the models design, in that a particular process must take some form of control if a failure occurs and that some communication is involved in detection and recovering from failures.

An appropriate value for the timeout value used within these programs is possibly more important than that of other programs, as part of the motivation for a distributed model may be the ability to handle high communication cost or latency (or variation of these).

If fault transparency is not as important then it would be possible to avoid the need to gather the data and just allow the processes to output to secondary files, which would reduce the fault tolerance limitations. However, it would not remove all of the problems as, if more than one failure occurs in a single iteration, then the blocks may not be contiguous which may make writing to files, in a sensible manner, difficult. If this approach were adopted then there would always be problems and limitations to writing to a file as if multiple iterations of the algorithm are allowed naming files would become difficult. Whichever alternative approach is considered, the results would need considerable post-processing which may be greater than the transparent version. Although if the motivation for using the distributed model is that communication costs are very high (files may be written to a local store and collated at a later time), then this may be preferable to the transparent example.

3.4.7.2 Code size, development time and difficulty

The development of the programs for this example seemed less problematic and considerably shorter and easier than the previous example, but this may be due to the similarity with the previous
example and a better understanding of MPI programming and debugging, though it is still not simple
to develop these solutions and with much more complex programs the development time could still be
substantial. The code has still grown considerably more complicated and larger. The final fault tolerant
version is approximately 400 lines compared with approximately 150 in the original program, which
highlights the need for writing sections of code which can integrated into other programs relatively
easily (e.g. the synchronisation function).

3.4.7.3 Theoretical performance

Due to the similar nature of this program to the semi-distributed example (section 3.3)
especially considering that the fault detection and recovery methods are very similar and require a
similar overhead it is reasonable to use a very similar inequality for determining the successfulness of
the fault tolerant program.

\[ T_{\text{timeout}} = \text{Time out value used for synchronisation.} \]
\[ T_{\text{comp}} = \text{Time for performing the computation of M/N columns of matrix C.} \]
\[ T_{\text{comp2}} = \text{Time for performing the computation of } (2(M/N)-1)/(N-1) \text{ columns of matrix C.} \]
\[ T_{\text{gather}} = \text{Time for performing the gather operation } (2(M/N)-1 \text{ T}_{\text{msg}} \text{ at most}). \]

\[(N T_{\text{timeout}}) + T_{\text{comp2}} + T_{\text{gather}} < T_{\text{comp}} (N-1) \text{ for non-root failure}\]

\[(2N T_{\text{timeout}}) + T_{\text{comp2}} + T_{\text{gather}} < T_{\text{comp}} (N-1) \text{ for root failure}\]

The time for computing the missing part (and padding) of matrix C requires much less an
overhead than re-executing the original, the success of the final fault tolerant program is dependant
upon the timeout value chosen and the overhead of the gather operation, which may be crucial if the
distributed model is used, due to high communication costs.

3.4.7.4 Performance tests

These tests were performed in a similar manner to those in sections 3.2 and 3.3, though it is
important to mention that, as the times are taken from root process only, the results for the original
program may be slightly inaccurate, though from execution comparing all the times the results are
very similar for each process.
These results show that, similarly to the semi-distributed example, there is a performance cost to synchronisation, which increases with the number of processes, however it is more extreme here due to the better performance of the original program. The difference between the non-root and root fault tolerant versions of the program running with 8 processes may just be an anomaly of the testing process as there should be little difference.

3.4.8 Summary

This section demonstrates that similar fault tolerant methods can be applied to a fully distributed program than can be applied to a semi-distributed program, with some compromises as the distributed nature of the program, by adopting some communication and a root process controlling certain aspects of the fault detection and recovery.

Summary

In this chapter, all three of the matrix multiplication examples have been covered, including the initial application, designing, implementing and evaluating the fault tolerant features. These examples have raised some key issues involving fault tolerance, including the need for non-blocking point-to-point communication, synchronisation and communicator generation for handling collective communication and more subtle issues involving the position of failures in the programs and the choice of timeout value for messages. The following chapter considers a considerably different paradigm for parallel programming, known collectively as iterative methods.
4 Iterative example program

Introduction

Initially this chapter begins with an explanation of iterative programs and a description of the original program used. The rest of the chapter describes the features and design of fault tolerance for the program, followed by the implementation of fault tolerance for the program, concluding with an evaluation of the performance and effectiveness of the fault tolerant features. The original program discussed in the chapter was developed by P. Jimack and N. Touheed and is discussed in detail in Jimack and Touheed (1999). The version accompanying this project has only had minor changes made.

4.1 Iterative programs

Iterative programs involve the repetition of a particular algorithm until a particular criterion is satisfied e.g. a value or error has fallen within a tolerable range.

4.2 Finite difference

This is a particular example of an iterative program for calculating the heat dissipation across a surface where heat is introduced at the edges of the surface. This is achieved by dividing the surface into a number of discrete points and averaging the surrounding points to calculate the new value for a point every iteration until a certain minimum error occurs. For more details of the algorithm and the original program described below see Jimack and Touheed, (1999);

4.3 Key features of the original parallel program

4.4 Fault tolerance design

There are three key features that need to be implemented in order for the program to be fault tolerant. Two of these have been seen in the previous examples and so are only discussed briefly in this section.

4.4.1 Non-blocking communication

As seen in the previous example programs, all point-to-point communication must be non-blocking in order to be fault tolerant. Although the original program uses non-blocking MPI_Irecv functions to send data between neighbouring processes, these are completed using an MPI_Wait function which blocks until the operation is completed. If the sending process has failed, this operation will never complete and halt the process, so it must be replaced with an MPI_Test function as seen in the previous examples.
4.4.2 Synchronisation

In the original program synchronisation occurs when the maximum error is gathered to all processes. As seen in the previous example programs, collective communication must only involve working processes in order to complete; therefore prior to performing the gather it must be possible to determine if any processes have failed. This can be achieved by synchronisation between all of the processes, which is handled in an almost identical manner to the previous examples (see sections 3.3 and 3.4).

4.4.3 Check pointing

As every iteration of the algorithm is dependant upon the previous one, a failure in any iteration will prevent the next one being calculated correctly. One method of avoiding this would be to restart the algorithm when a failure occurs, but the cost of this would be substantial if, for example, the problem requires a large number of iteration and failure occurs relatively near to completion. A more scalable approach would be to copy the result of a particular iteration, so that the algorithm may restart from this iteration if a failure occurred, this process is known as check pointing. This will have a substantial communication and processing overhead as most of the data stored on every process must be collected and stored on a particular process (or processes). Thus the interval at which this occurs must be carefully chosen so that the overhead of this and the restoration of the data do not exceed the overhead of performing the calculations and communication required for the previous iterations up until the checkpoint.

4.5 Implementation discussion

The first part of this involved altering the original program so that the processing for an iteration was separated from the main function in the code so that it was easier to add the extra control necessary to implement fault tolerant features.

4.5.1 Check pointing

The original program uses a dynamically allocated 2 dimensional array for storing that processes part of the surface (+ extra ‘ghost’ rows for calculation purposes). However it is not possible to use a dynamically generated 2 dimensional array as the data for an MPI_Gather (or other MPI function), neither is it possible to define an MPI type to handle this structure. This meant that the required data (not all of the data on all processes, as 2 copies of boundary rows between processes are not required) needs to be copied into a temporary 1 dimensional array so that it could be used in an MPI_Gather function call.

Due to the uneven number of rows stored on each process (even though program is limited to equal divided between processes, some have ghost rows that are not required), the use of an MPI_Gatherv function is required to simplify the collection process. This function is similar to a
normal MPI_Gather, with the extra ability to take an array of receive counts and displacements to allow for greater flexibility. The receive counts and displacements are calculated using a similar method to that used by each process to determine the amount of the problem it is responsible for.

4.5.2 Restoration of checkpoint

Restoration of the checkpoint data involves a similar process to the collection of data, except in reverse. Similarly to this, an MPI_Scatterv function call is used to distribute the data to the remaining functioning processes. This is similar to the MPI_Scatter function, however it also allows for send counts and displacements to be specified in a similar manner to the MPI_Gatherv function. This simplifies the restoration process by allowing different amounts of data to be easily distributed to the remaining processes and the displacements provide a means of overlapping this sending data so that the boundary rows between processes can be distributed to both processes.

The only extra things needed for restoration are the copying of this data from a temporary 1-dimensional buffer to the 2-dimensional array used for calculations and the specifying of the initial conditions in the other 2-dimensional array.

However, due to problems with reallocating the 2-dimensional arrays used originally, new 2-dimensional arrays were required, limiting the program to a single restoration of checkpoint data. Any failures detected by synchronising the processes after this restoration cause the program to terminate cleanly prematurely.

4.6 Evaluation

4.6.1 Fault tolerance

Due to the problems with memory allocation specified in section 4.5 and the restrictions imposed by communicators used in collective communication (see section 3.2) the program is limited to a single failure overall. However if the memory allocation problems could be rectified then the program could be adapted to handle multiple restorations of data with a single process failing in a particular iteration. Similarly to previous discussions it would also be possible to remove the restriction imposed by the limited generation of communicators, but this leads to other issues (see section 3.2 for further discussion of this).

As with the previous examples there are a number of places where failures may occur which are not handled, for example between the synchronisation of processes and the MPI_Allreduce collective communication (see section 3.2 for further examples of these problems and Appendix B for details of the MPI_Allreduce function).
4.6.2 Fault tolerance tests

Due to the problems highlighted in section 3.2, it was not possible for conclusive testing of the fault tolerant capabilities to be undertaken. However with similar modifications as mentioned in the same section, the program is tolerant to induced failures within the limitations mentioned (see section 4.6.1).

4.6.3 Code size, development time and difficulty

Due to increased confidence and experience in developing parallel applications by this stage of the project, development of these example programs was less difficult than the previous examples once extensive experimentation and investigation had cleared up the problems outlined in section 4.5.

4.6.4 Theoretical performance

To simplify the analysis of this example, it is assumed that the time taken for a particular iteration is fixed for a particular number of processes (N) and problem size (M). The model also assumes that the time required for a successful synchronisation is small and so is not a significant factor in the time required for an iteration. This analysis is considerably different to the previous examples, as the program is iterative and so repeats many operations a multiple number of times dependant upon the problem size. Due to the iterative nature of the program, the cost of recovering and continuing from the failure is dependant upon which iteration the failure occurs. The worst case of this involves failure a single iteration before a checkpoint occurs (defined by the checkpoint interval). The initial model for the time required for the fault tolerant model is summarised below.

\[(\text{number of checkpoints before failure} \times \text{time needed for a checkpoint}) + \text{time for unsuccessful synchronisation} + \text{time for restoration of checkpoint} + \text{Time for remaining iterations}\]

So the inequality used for successfulness of this example is shown below.

\[I_M = \text{The number of iterations required for a particular problem size.}\]
\[I_{\text{interval}} = \text{The checkpoint interval used for checkpointing.}\]
\[I_{\text{failure}} = \text{The iteration at which the failure occurred}\]
\[I_{\text{lastcpt}} = \text{The iteration at which the last checkpoint occurred (calculated using the checkpoint interval and iteration of failure).}\]
\[T_{\text{sync}} = \text{Time required for an unsuccessful synchronisation (dependant upon failure, see section 3.3, but small in comparison to other costs).}\]
\[T_{\text{iteration}} = \text{Time for performing a single iteration of the problem for a specific N and M (including successful synchronisation).}\]
\[T_{\text{checkpoint}} = \text{Time for performing a checkpoint.}\]
$T_{\text{restore}} = \text{Time for restoring from a checkpoint.}$

$\left( \left( \frac{I_{\text{failure}}}{I_{\text{interval}}} * T_{\text{checkpoint}} \right) + T_{\text{sync}} + T_{\text{restore}} + \left( (I_{M} - I_{\text{lastcpt}}) * T_{\text{iteration}} \right) \right) < (I_{M} * T_{\text{iteration}})$

This can be simplified to the following inequality.

$\left( \left( \frac{I_{\text{failure}}}{I_{\text{interval}}} * T_{\text{checkpoint}} \right) + T_{\text{sync}} + T_{\text{restore}} \right) < (I_{\text{lastcpt}} * T_{\text{iteration}})$

It is clear from the above inequality that the most significant variables for the success of the final program are the interval at which checkpoints occur and the cost of checkpointing the data (as the cost of restoring the data is probably similar). If this exceeds the amount of time required to re-execute all the iterations up until the last checkpoint then the fault tolerant version is unsuccessful. This makes the choice of checkpoint interval crucial and very difficult.

### 4.6.5 Performance tests

These tests were performed in a similar manner to that of the previous examples, with one major difference, as these are C programs, they were compiled with the mpicc command, which uses gcc version 3.2.2.20030222. The checkpoint value (65) was chosen so that only a single checkpoint was performed and the timeout value was 0.25, however this should not affect these results. Figure 4.1 shows the results of the tests.

![Figure 4.1 – Performance test results for iterative example programs.](image)

These results show that, even with a single checkpoint, there is a significant performance cost in checkpointing the data, so it may only be appropriate where there is a large amount of computation...
involved per iteration, whereas here it dominates this cost. It does appear to scale quite well to an increase in the number of processes, as the gap between the original and fault tolerant versions of the program is approximately 3 seconds for 4 and 8 processes and 3.5 seconds for 16 processes. However the choice of problem size may have been too small, as all the programs perform worse with a larger number of processes, though this is not directly relevant to this project.

Summary

This chapter discusses the design and development of fault tolerant versions of an existing program, highlighting certain characteristics and solutions unique to an iterative program. The chapter finishes with an evaluation of the programs; highlighting the significant performance impact that check pointing has on programs of this nature. The next chapter concludes the entire project and suggests possible extensions and new directions for future projects.
5 Conclusion and possible extensions

5.1 Overall conclusion

Testing of the example programs produced for this project led to the belief (although not entirely confirmed) that the effectiveness of the fault tolerance in these examples is dependant upon the ability of the underlying MPI implementation to handle errors e.g. premature TCP/IP socket connection terminations. This would require a large amount of work to correctly determine, so is outside the scope of the project. This discovery combined with the substantially increased development time for even simple example programs means it is more appropriate for large scale or commercial systems to investigate alternative provision of fault tolerance, e.g. fault tolerant implementations of MPI or middleware libraries.

5.2 Possible extensions

Assuming that the underlying MPI implementation is deemed to be able to handle errors, there is significant scope for developing the example programs to handle multiple failures per execution of the program or iteration of the algorithm within the program.

There is also scope for more extensive testing and profiling of the existing example programs to determine to exactly what degree the programs can handle faults. This could also include an investigation into the stability of the underlying MPI implementation.

The programs in this project have a high dependence on the time out value and checkpoint interval; there is also a need for research into determining appropriate timeout or checkpoint interval values under different circumstances e.g. a heterogeneous mix of computers or different network capabilities. There is scope for analysing the potential for automatically determining timeout values dependant upon the underlying architecture and possibly developing tools to do this. Alternatively research into determining an appropriate checkpoint interval dependant upon analysis of the algorithm a program uses.

There is an opportunity to investigate the effect of Grid computing on fault tolerance and possibly to adapt the current example programs (or entirely new programs) for use upon this architecture.

It would also be useful to investigate whether the new features in MPI2, such as the ability to dynamically spawn processes, would have a positive effect on the development of fault tolerant MPI applications and whether these features create any new problems.

Also considering the uncertainty of the effectiveness and appropriateness of the current example programs, it could be useful to investigate the alternatives provided by projects such as FT_MPI.
Bibliography


Dongarra, Jack & Fagg, Graham, (2003), *FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world*, Department of Computer Science, University of Tennessee.


Appendix A – Reflection

As with most final year projects, time management played an important role in the success of the project, all students are aware of the problems yet it appears to affect all projects. There were specific moments in this project where better time management would have helped a great deal, a lack of large amounts of work in the Easter holiday, meant a very busy period after the holiday.

More specifically to this project, any student considering taking on a project in a subject they are unfamiliar, should be aware of the constant pressure of not understanding a great deal and the likelihood that planning and progress will not be as easy as in another project with which the student is more familiar.

Also I found that many of the problems encountered in this project were due to missing simple solutions to problems, I believe this was largely down to the lack of experience in parallel programming and a feeling of being out of my depth at times. On a very specific note, any student using Microsoft Visual C++ as a development environment for all or part of their project should be aware of the apparent disregard of memory allocation problems within debug executables it creates. Any student developing software using C\C++, especially parallel software that uses dynamic memory allocation should investigate memory analysis software, such as valgrind, recommended by Mark Conmy.

However, a project in a completely new subject area can be very rewarding and often very interesting, even if it does not go as well as planned. I feel much more confident in my ability to develop parallel programs and have developed a good understanding of the MPI standard.

I would recommend a project in this subject area to a dedicated student as it provides an opportunity to get involved in cutting edge development and experience the very large amounts of research within this field.
## Appendix B – Glossary

**Terms and MPI Functions**

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point-to-Point communication</td>
<td>Communication between 2 individual processes in a system.</td>
</tr>
<tr>
<td>Collective communication</td>
<td>Communication between all (or subset of) processes in a system e.g. scatter, gather and broadcast operations.</td>
</tr>
<tr>
<td>Blocking communication</td>
<td>Communication that waits until completed.</td>
</tr>
<tr>
<td>Non-blocking communication</td>
<td>Communication that is started but does not wait until completed.</td>
</tr>
<tr>
<td>Communicators</td>
<td>The MPI standards means of grouping processes together, all process ranks are only relevant to a specific communicator.</td>
</tr>
<tr>
<td>MPI_COMM_WORLD</td>
<td>The communicator containing all processes in the execution of a program.</td>
</tr>
<tr>
<td>Status objects</td>
<td>Provide information regarding the status of a particular operation, e.g. source, tag and error.</td>
</tr>
<tr>
<td>Request objects</td>
<td>Used to refer to a particular non-blocking send or receive within the MPI_Wait or MPI_Test functions.</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>Sends some particular data to a particular process. Is an example of blocking communication and so requires a corresponding receive function call (MPI_Recv, MPI_Wait or MPI_Test) on the destination process.</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>Receives some particular data from a particular process.</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>The non-blocking equivalent of MPI_Send, starts the sending process but does not wait for completion. When completion occurs is dependant upon the MPI implementation.</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>The non-blocking equivalent of MPI_Recv, posts a receive to the MPI daemon, but does not wait until the data is received and placed in the receive buffer.</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>Waits until the completion of a particular non-blocking send or receive.</td>
</tr>
<tr>
<td>MPI_Test</td>
<td>Tests to see if a particular non-blocking operation has completed, sets a flag (passed as a parameter) to true if is complete.</td>
</tr>
<tr>
<td>MPI_Broadcast</td>
<td>Broadcasts some particular data from a specified process to all processes in a communicator.</td>
</tr>
<tr>
<td>MPI_Scatter</td>
<td>Sends equal divisions of some data from a specified process to all processes in a communicator.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>MPI_Gather</td>
<td>Receives equal amounts data from all processes in a communicator.</td>
</tr>
<tr>
<td>MPI_Reduceall</td>
<td>Receives some particular data from all processes in a communicator to a specified root process, performing some mathematical operation e.g. average, minimum on the received data.</td>
</tr>
<tr>
<td>MPI_Scatterv</td>
<td>Vector variant of MPI_Scatter, provides support for variable amounts of data sent to each process in a communicator.</td>
</tr>
<tr>
<td>MPI_Gatherv</td>
<td>Vector variant of MPI_Gather, provides support for receiving variable amounts of data from each process.</td>
</tr>
</tbody>
</table>