Executive Summary

With petaflop/s supercomputers already in place (top 7 supercomputers as per the list released by TOP500 in November 2010 [1]) and the trend slowly moving towards exa-scale computation, faults and failures become common place. In such scenarios, reliability and resilience of the system become major concerns. Methods like Checkpoint and restart, message logging and MPI semantic modification, which are being adopted now in the industry, can yield only few degrees of fault tolerance. As the nodes involved in a computation grow beyond 100,000, such methods may actually stop helping or worse, create problems.

Making the programs fault tolerant by using inherently fault tolerant algorithm seems to be an attractive option to address the issue. Advantages include accuracy of the result despite faults and failures, no effort needed in recovery, no extra overhead of checkpointing. This project is one such attempt using one of the seven dwarfs, The Monte Carlo algorithm.

In this project, an attempt has been made to build a fault tolerant prototype application which runs on the commodity cluster and implements Monte Carlo algorithm using MPI for message passing between the nodes. Programming models and design methodologies to make a typical MPI program immune to errors at the application levels are investigated. The impact of the errors and process crashes on the system is measured. Monte Carlo Numerical integration is chosen to be the application for simplicity. This can be extended to much complicated applications later.

Main contributions and achievements of the project are –

- A novel idea of using the randomness property of Monte Carlo to design a fault oblivious and scalable algorithm for parallel computers (section 3.3)
- A server-worker programming model for the algorithm paired with the implicit acknowledgement design methodology for achieving fault tolerance at application level (section 4.4)
- An investigation into the extent of fault tolerance achieved by the project in terms of accuracy and performance (section 5.2)

Acknowledgement

I would like to thank my supervisor Mr Simon McIntosh Smith for his continuous support and guidance throughout the project. His expertise in high performance computing has proved invaluable over the project tenure.

I humbly acknowledge the assistance of Mr Callum Wright during the project with the installation of various libraries on the BlueCystal.

I cannot forget to express appreciation to the administrators of BlueCrystal, the High Performance Computing cluster at the University of Bristol, for being there whenever I had issues with BlueCrystal.

I would also like to thank the Library staff who helped me in gathering a lot of information for this project.

I am grateful to all the teaching and non-teaching staff of the University of Bristol, who assisted me for the successful completion of the project.

Lastly, I would like to recognise the gratitude for my family and friends for the constant support and encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

Today's high performance computing involves hundreds of nodes working in parallel to solve complex computations. Its areas of applications include, but not limited, to Earth system modelling, climate modelling, molecular dynamics, fluid dynamics, large scale data analysis, particle physics, nuclear physics, biomedical applications, etc. MPI is the *de-facto* library widely accepted and used for the message passing between the nodes. This specification assumes the integrity of the messages sent/received in the network and takes no measures for fault tolerance.

As the number of processors working on a single program rises, the chances of faults and failures also increase. Even if the failure rate is as low as 0.01%, with the number of nodes in a clusters rising to as many as 10^{10}, the chances of a node failure would be significant and cannot be ignored. The very big concern with the HPC is that when an error is encountered the whole program aborts and the data computed till then is lost. The whole computation has to be again taken up from the scratch. When thousands of nodes are working in parallel to compute a complex mathematical problem, a failure at one node which halts the whole system is unacceptable.

At present, checkpointing and restart is the most popular way of dealing with the issue. Status of all the nodes is saved periodically and when the program aborts due to an error, it is restarted from the last check pointed value. Many MPI implementations today extend checkpointing facility. However this method is tedious and consumes a lot of processor time which could otherwise be used for computation. Besides, as the number of nodes in a program increases, the MTBF (Mean Time Between the Failures) also increases which could be a bottleneck for this method [**2**].

With High Performance Computing gaining popularity in every field of computation, the need for a robust fault tolerant parallel system has gained importance like never before. Lot of research is been going on in this field. Papers are published on diskless checkpointing [**3**] [**4**], attempts have been made to alter the MPI semantics [**5**], to bring in MPI fault tolerance through network fault tolerance [**6**] and even to use fault tolerant algorithms [**7**]. Though each of them has succeeded in achieving some fault tolerance, none of them have been able to provide complete solution to the issue.

This project attempts to explore the idea of making an application fault tolerant by using Monte Carlo algorithm in a prototype program for numerical integration. Monte Carlo is an easily parallelisable algorithm widely used to solve complex problems in HPC. It is based on randomness and probability, and is very simple to understand. Its most important area of application includes computing the multidimensional numerical integrations and electronic structure problems. It has a natural property for fault tolerance due to the randomness involved, which is exploited in this project to design a fault tolerant MPI application. Since MPI doesn't provide any straight forward measures for handling faults, methods to get the best results by what MPI can offer are investigated.

In Chapter 2, the aims and the objectives of the project are listed. Chapter 3 details the basic background needed for understanding the project. Few related work and present day HPC systems are surveyed and the degree of fault tolerance achieved by each of them is discussed. This chapter also introduces Monte Carlo algorithm, Numerical Integration and Error analysis in Monte Carlo numerical integration. A very brief note on BlueCrystal supercomputer is also presented. Chapter 4 presents the work carried out for the project, including the issues encountered, the decisions made and the rationale behind each of them. The final prototype program is also explained in detail. Compilation of the result is done in Chapter 5. Observable patterns and behaviour of the system with the algorithm are tabulated and plotted. The results are logically reasoned and conclusions are drawn. Chapter 6 critically examines decisions taken during the project to check if they have worked out for the best or worst of the project. It summarises what's working in the project and what would have made it better. Chapter 7 presents possible improvements to the project and the future expansion ideas. The section Appendices presents some important excerpts from the code of the prototype program. Finally, Bibliography section lists all the references and citations made in the project.

Chapter 2. Aims and Objectives

The aim of the project is to develop a fault tolerant algorithm for massively parallel computers, write a prototype program for the same and to demonstrate the fault tolerance achieved through result and analysis.

Stated below are the objectives set for the successful completion of the project-

Step 1: Set the stage

a) Choose the right MPI implementation to work with
b) Configure the BlueCrystal accordingly
c) Write a prototype parallel program using C and MPI for Monte Carlo Numerical Integration and get it working on BlueCrystal

Step 2: Achieve Program Fault Tolerance

a) Identify the weak points of an MPI program
b) Explore the ways to avoid them in the program
c) Make the program immune to errors and process failures

Step 3: Program for Algorithmic Fault Tolerance

a) Integrate the prototype program with the methods found to make the program error immune
b) Check for the proper working of algorithm and the functionality

Step 4: Result and Analysis

a) Induce errors in the arguments to MPI functions and measure the impact of the failure on the stabilised program
b) Check for the application response to processor crash and fail-stop process failure
c) Compile the results and analyse them to draw a logical conclusion

Chapter 3. Background and Context

This chapter explains all the background needed to understand the work carried out and rationale behind few decisions made for the project. The first section explains what we exactly mean by fault tolerance in MPI, second section details all the work done thus far in the industry for fault tolerance and the third sections lays foundation for the concept which led to the developed of this project.

3.1 MPI and fault tolerance

MPI is the *de facto* message passing model for the cluster computing. It is an API specification or a library of routines and pre-processor macros, used to communicate between the nodes/processes in a cluster. OpenMPI, LAM-MPI and MPICH, are some of the popular implementations of MPI available in C, C++ and FORTRAN.

The MPI specification assumes the ideal case of no process failure. It assumes the integrity of the messages being sent or received. Though this is true for most of the transactions, it does not hold every time. The situation with MPI implementations today is that, none of these (except for FT-MPI to an extent [**5**]), supports process failures. This means that when there is a process failure, the whole program aborts. The control is not returned to the application and hence there is nothing much one can do in application level to make it fault tolerant. This limits the use in a lot of ways. If a short application is running on less number of core, the chances of process failure over the run of the application might be very less but as the number of processes working for an application increase or if the application takes a lot of processing time, the probability of the processor failure over the execution time increases. This makes the application very fragile.

To tackle this problem, each implementation has come up with few fault tolerant features. Most of them are falling back on checkpoint and restart or checkpoint and rollback. In this method, the status and the results from each processor is logged at a regular intervals. If there is a failure, the program restarts or rolls back to the latest check-pointed value and starts all over again. Few of the implementations have tried implementing other kinds of fault tolerant features. In this chapter, each of those is discussed in detail.

3.2 Attempts for MPI Fault tolerance so far

As mentioned, there have been a lot of attempts to address the issue of fault tolerance in the field. Classification of those attempts can be made based on various factors. Each author or university has their own take on the classifications. Some of the most popular classification criterions are stated below-

- They can be either automatic, where the faults are detected and handled during the runtime or Non-automatic, where the execution has to be halted and restarted.

- Based on the levels of the software stacks the fault tolerance is implemented, they can be classified as a framework, an API or a Communication Library.
- Based on the approach, they can be broadly classified into Checkpoint based i.e., checkpointing/restart or checkpointing/rollback, Log based i.e., message logging, and Algorithm based. [**3**]

Other than message logging and checkpointing, there have been attempts to make the MPI programs fault tolerant [**3**]. LA MPI, which concentrates on making the network fault tolerant, so that the message integrity is maintained and CIFTS, which aims at system level fault tolerance though coordination among all the components of the system are two examples which are discussed in the following sub sections.

3.2.1 Checkpoint and Rollback

Checkpointing is very popular because of two reasons. Firstly, it is simple to understand and easy to implement. Secondly, it does not require changing the algorithm of the program as it can be implemented as a separate module by itself. There are three ways of checkpointing, namely, coordinated checkpointing, uncoordinated checkpointing and Communication Induced checkpointing.

1. In coordinated checkpointing, the network is flushed before it is check pointed, so that all the processes are rolled back to the same status snapshot and they are synchronous when they restart. As only one proper checkpoint is needed at any given time, the memory utilisation is low in this method. However, the problem with this method is that the time taken to checkpoint is comparatively high, and hence the performance goes down.
2. In uncoordinated checkpointing, each process has its own checkpoint and every process restarts independently of the other. This has many disadvantages, major one being the Domino effect where system will roll-back to the original state causing massive computation losses. It also needs a lot of memory as each process will have to maintain multiple checkpoints over the entire execution.
3. Communication Induced Checkpointing method combines both the methods stated earlier. There are two kinds of checkpoints- Local and Forced. Local checkpoints are taken by the processors independently and the forced are the ones taken by all the processes to maintain a system level consistency.

But the drawback of these, or for that matter any check point and rollback approach, is that some of the precious processor time which could be used for further computation would be spent on checkpointing. As the number of failures increase over the execution time, the cost of checkpointing goes higher and higher.

Since checkpointing and rollback is out of the project's scope, it is not discussed in detail in the thesis. Importance is given to those features of the implementation which do not checkpoint, but achieve fault tolerance by other means.

3.2.2 LAM based MPI-FT

LAM stands for Local Area Multicomputer and a fault tolerant version of the LAM/MPI is MPI-FT. It has an interesting way of achieving fault tolerance. An observer – peer architecture as shown in the Fig 3-1 is adopted from [**4**]

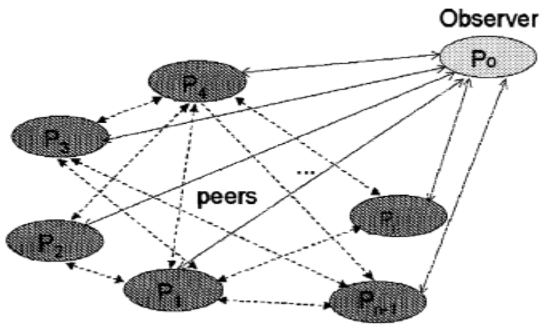


Fig 3-1: Peer Observer Setup

1. 'Observer' is the master process (rank = 0), which informs all other processes of any failure which occurs in the network and takes up the action for recovery. Message storing can be done either by the observer or by each of the peer processes.
2. At the start when the MPI is initialised in MPI_FT_Init() routine, a non-blocking receive is initiated in all the peer processes which wait for the failure message from the observer. A Don't Send table is maintained in each process which consists of all the messages it receives from other peers.
3. A UNIX script checks if all the processes are alive at regular intervals of time and reports to the observer using a FIFOs
4. When there is a failure, the process is re-spawned and a message is sent indicating the failure to all the living peers along with the rank of the dead process.
5. Peers use MPI_Test command to know if any failure message is received. Upon receiving, the recovery process is triggered. They consult the routing table and resend the messages which were supposed to be sent to the dead peer.
6. All the peers send the information from their respective Don't Send table to the replacement process about the messages they had earlier received from the dead node. This helps the replacement process to create a table of its own

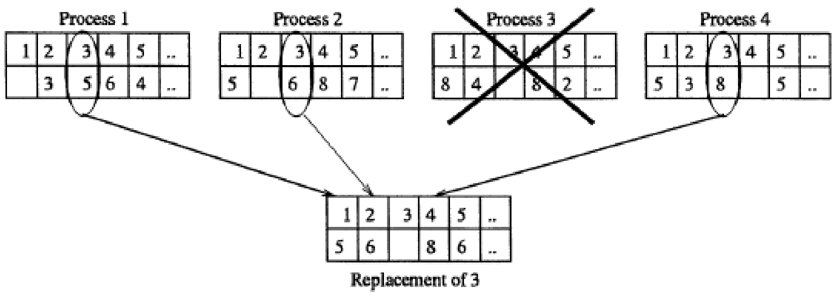


Fig 3-2: When a process has to be replaced

7. A communicator matrix is created at the start to enable the re-spawning of the failed process. If there are p peers, p communicator worlds are created, with one node kept out of it every time. The Observer world is used for normal operations. Other worlds are used only in case of a failure. When the ith peer fails, the communicator world which had excluded the ith peer is used for the re-spawning.

World

		0	1	2	3	4
R	0	-	0	0	0	0
a	1	0	-	1	1	1
n	2	1	1	-	2	2
k	3	2	2	2	-	3
	4	3	3	3	3	-

The disadvantages include intensive utilisation of resources upon failure recovery and requirement of huge storage space. Another major negative point is that MPI-FT has fault tolerance addressed only to the peer processes. It assumes that the master node or observer never fails. Though this approach was a very good effort in its own right, it wastes a lot of computation space and time for the fault tolerance. The overhead is just too much.

3.2.3 LA-MPI

Developed by Los Alamos National Laboratory, LA-MPI is an open source [**5**]implementation of MPI which focuses on the network fault tolerance unlike other implementations which target the process fault tolerance. It uses a checksum/retransmission protocol instead of the standard TCP/IP, as it is claimed to be 'highly efficient and more reliable compared to TCP/IP'. [**5**] The architecture of LA MPI can be divided into user level (ULM) and kernel level Messaging (KLM) as shown in the Fig 3-3. The MPI implementations are supported at the user level messaging, which has two other sub layers

- MML (Memory and Message Layer) – manages the activities related to messages like routing, message tag matching, etc. an abstraction network data path object called 'path' is used to send/receive messages. The path scheduler takes care of binding the messages to a path between a source and destination. Path manages fragmenting/assembling the message during the transmission. A path specific checksum is used to ensure the faultless transmission. In case of corruption, the fragment retransmission occurs.
- SRL (Send and Receive Layer) takes care of sending and receiving the messages.

One major drawback that can be seen in this model is that the process failure is not handled, though other link level failures and network level failures are efficiently addressed.

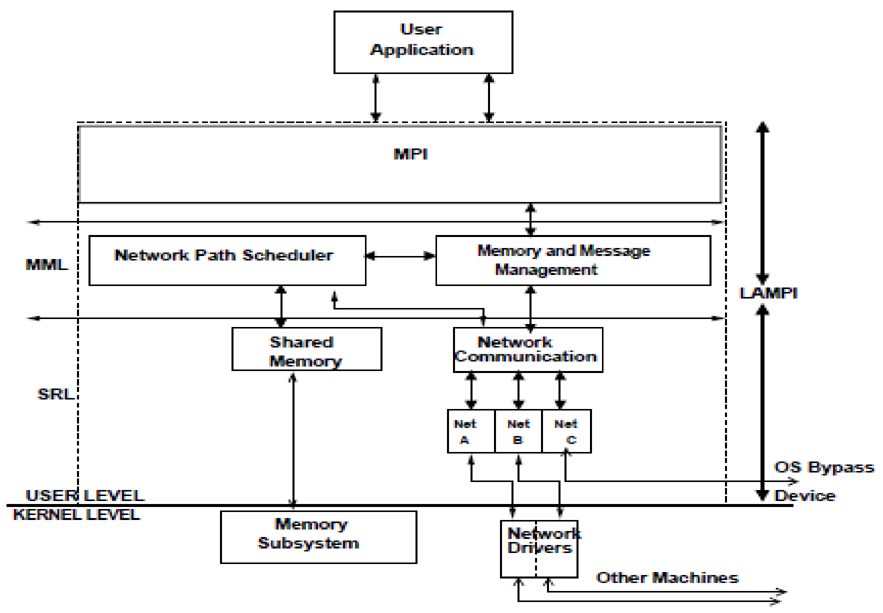


Fig 3-3: LA MPI Structure

3.2.4 CIFTS

CIFTS stands for Coordinated Infrastructure for Fault Tolerant Systems.

Considering the fact that fault can be triggered at any level, including hardware, software, middleware, network, etc., there is a need for coordination among all the features of the system components to achieve system wide fault tolerance. CIFTS proposes to increase the reliability of the system by actively sharing the error or warning information with the system as soon as it occurs, so that a proactive action can be taken up. An asynchronous messaging backplane called FTB - Fault Tolerant Backplane is used to communicate between the system software programs which can include variety of softwares, starting from the OS to any high level user applications. FTB is a layered architecture and its software stack consists of network layer, manager layer and client layers.

When they are FTB enabled, the system softwares are called 'FTB clients'. These FTB clients are connected to the demons called 'FTB agents' which are arranged in the self-healing tree topology in the FTB framework. These agents are responsible for the book keeping and decision making in the framework. The clients can use FTB Client APIs to initialise themselves, publish events or subscribe/unsubscribe to events. Events can be received either by polling or call-back methods. Though there is no restriction on the information the client publishes, it is asked to however specify the namespace where it is publishing it. The clients who wish to receive such information should also be registered to the same namespace. Fig 3-4 shows the CIFTS framework as shown in [**6**].

If the MPI implementations like MPICH and Open-MPI, fault aware libraries and middleware are FTB enabled, they can be plugged in to the FTB infrastructure, and the CIFTS system can try to improve the fault tolerance of a large system by coordination.

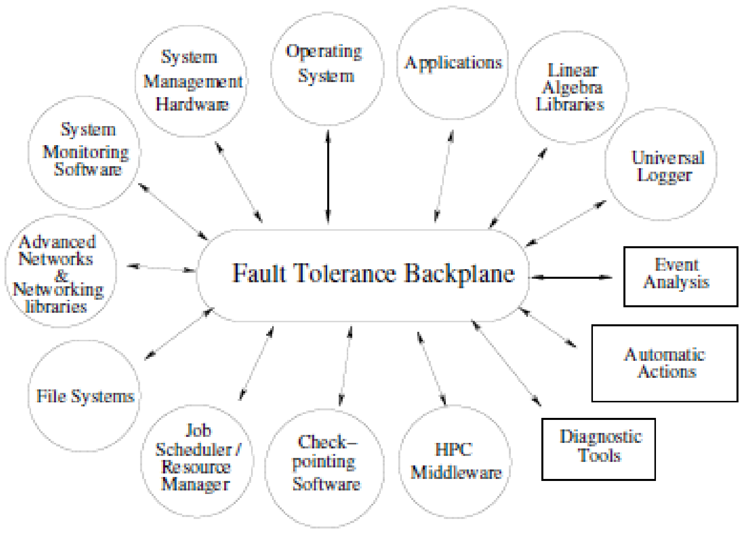


Fig 3-4: CIFTS Framework

Organisations like Argonne National Laboratory, Indiana University, Lawrence Berkeley National Laboratory, Oak Ridge National Laboratory, Ohio State University, University of Tennessee, Knoxville are some of the organisations leading the CIFTS effort. Recent release of CIFTS include FTB enabled OpenMPI 1.5.2, FTB enabled MPICH 1.3.1, and MVAPICH2 1.6 which supports both checkpoint/restart and process migration methods of fault tolerance [**7**].

3.2.5 FT-MPI

FT-MPI is a partial MPI2 implementation developed by the University of Tennessee, Knoxville, in which attempts are made to change the MPI semantics to make it fault tolerant [**8**].

Faults of two types are identified – communication error and process exit. When there is a failure detected in the communicator, all the processes are informed about it. Communicators are simply updated if it was a communication error. If it was a process exit, the communicator having the process is modified according to the failure mode chosen by the developer. Four failure modes are defined –

- Abort – default mode as in the MPI_ERRORS_ARE_FATAL
- Shrink – reduces the Communicator and rearranges the process ranks to be contiguous
- Blank – leaves the gap
- Rebuild – creates a new process to fill the gap

Some of the major semantic changes are

- The communicator states which were just VALID and INVALID in the MPI specifications, are extended to include 3 states - {OK, PROBLEM, FAILED}
- Processes which had OK FAILED are extended to have 4 states – {OK UNAVAILABLE, JOINING, FAILED}

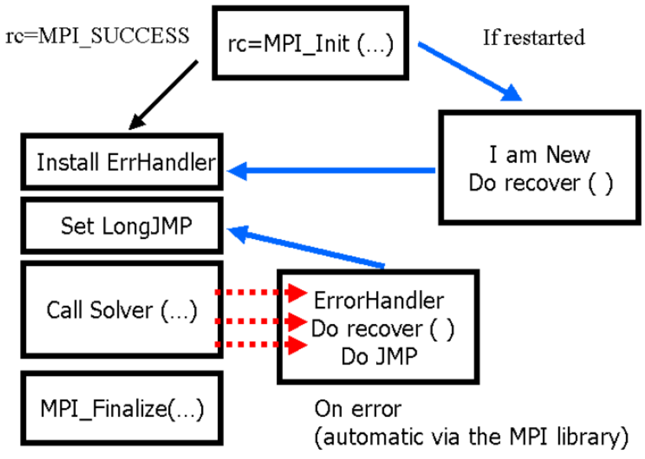


Fig 3-5. FT-MPI program flow

The program flow is as below

1. MPI_Init informs if the process is a newly started one or a restarted node in the program
2. If it is a new process, normal execution of the application is followed, i.e.,
   a. Installation of Error Handler
   b. Setting a LongJMP to return to the top levels of the function to handle the program flow during recovery
   c. Calling the parallel solver to check in case there is an error
3. If there is an error detected, Do_recover() routine is triggered where the error is handled according to the error mode selected by the developer
4. On completion, MPI_Finalize is executed and the application is terminated.

FT-MPI has made an effort to modify the MPI semantics by adding all the extra features stated. There can be blanks in the communicator, or even the ranks can be redefined on the run. The dead process can be re-spawned and assigned the same job. This is a powerful feature of the implementation; none of the other standard implementations do anything necessary for the fault recovery as does FT-MPI. Hence thus far, FT-MPI has done more than any other mentioned attempts in making MPI programs fault tolerant.

3.3 Algorithmic approach – Monte Carlo

This chapter introduces the algorithmic approach to fault tolerance in HPC. Algorithm based approach is one in which fault tolerance is aimed to achieve by the property of the program used in the computation rather than the implementation or framework. This can be categorised under the application solutions, as the fault tolerance is not in the API or the system software, but in the application which runs on it.

As the field of high performance computing grows bigger and wider, the number of processes involved in computing a scientific application increases in equal rates, which can only mean increase in the probability of one process failing. Another factor of importance in such scenarios is MTBF or Mean Time Between Failures. It is the average time elapsed between two failures during the system runtime. As the probability of failure increases, the MTBF also decreases. A system with a million processes can see a failure once in every few minutes [**9**] If checkpointing/restart is the method adopted for the fault tolerance, all the processes has to be stopped and reloaded from the previous check pointed data. Halting a million processes and reinitiating the computation every time there is a failure in the system not only wastes a lot of computation time but also makes it nearly impossible to complete the computation. Hence one can clearly see that there is a strong need for an algorithm which is inherently fault tolerant to sustain such MTBF and still deliver the result with little or no variation in the computational accuracy.

Being inherently fault tolerant means that the algorithm should be fault tolerant through its mathematical properties, that the lost processes need not be allocated to other nodes for calculation, as the algorithm compensates for the lost processes through its natural fault tolerance.

Scale invariance is another property of such algorithm. It means that each process in a huge computation has a definite higher limit to the number of other processes it can communicate with, throughout the execution time. This helps because when a node fails, it makes sure that the other nodes which do not communicate with the dead node are not affected. This also has a drawback. If a new node or a process is added to the system, it may go un-noticed by the other nodes with which it does not communicate. However, the major advantage of a scale invariant algorithm is that it localises the fault.

Monte Carlo algorithm is one such algorithm. This chapter provides an introduction to the algorithm and why it is inherently fault tolerant, considering a simple example of numerical integration using Monte Carlo algorithm.

3.3.1 Introduction to Monte Carlo

Monte Carlo algorithm is a method of arriving at an approximate solution to a complicated problem using randomness. In few computations, it is highly impossible to simulate every possible phenomenon accurately. In such cases, one can compute single realisation of the problem for a set of several random values in the probability distribution and arrive at an approximate answer. There are a lot of different definitions for Monte Carlo algorithm available. Ripley defines the Monte Carlo method as a "stochastic simulation", while Halton (1970) defines it as, "representing the solution of a problem as a parameter of a hypothetical population, and using a random sequence of numbers to construct a sample of the population, from which statistical estimates of the parameter can be obtained." [**10**]

Monte Carlo method is used widely not only in scientific computations but also in variety of other problems which involves statistical analysis. Weather prediction, environmental research, financial analysis, semiconductor device research, computer graphics are few examples of the fields in which it is extensively applied. Though the algorithm seems to be used in solving only random processes, it can and has also been applied to solve few of the non-random processes like the evaluation of complex integers, solving differential equations, solving inverse and non-inverse simultaneous equations etc. It has been said that 'it is the most effective way to solve large dimensional integrals or sums as the error is statistical i.e., it converges at a rate of $1/N^{\frac{1}{2}}$ for N configurations as opposed to $1/N^{\frac{1}{d}}$ in standard numerical integration techniques for d dimensions.' [**11**]

3.3.2 Numerical Integration

In this project, the Monte Carlo method for a numerical integration is considered. The numerical integration and Monte Carlo methods for numerical integration are discussed in this section with the help of an example. Simply put, Numerical Integration is an approximation of a definite integral. For instance, numerically calculating the approximate area under the curve f(x), closed at the interval [a,b] defined by the equation as below can be regarded as a simple one dimensional numerical integration.

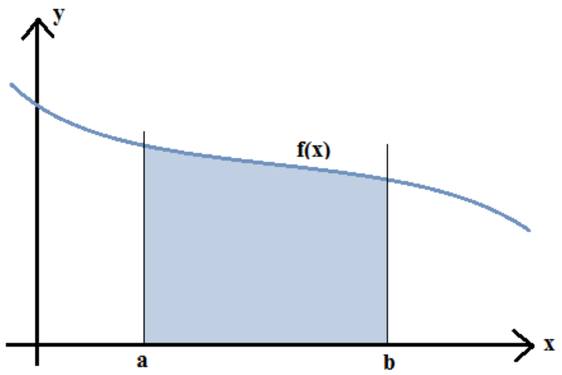


Fig 3-6: Numerical integration

$$I = \int_a^b f(x)dx$$

--- eq. 3.1

However, the complexity of the equation increases as the dimensions in the problem increases and the cost of computation grows exponentially with it. [**12**]. There are a lot of methods using which one can solve the numerical integration. Most popular ones are the Newton-Cotes formulas or quadrature formulas, Mid-Point rule, Trapezoidal rule, the Simpson rule, and Monte Carlo method.

3.3.3 Monte Carlo Integration

Monte Carlo method is used in the numerical integration mostly for multidimensional integrals. It involves the generation of a large number of random points within the area of interest to use for calculating the equation, to check if it 'hits' or 'misses' i.e., if it lies under or above the curve f(x) and the integration is evaluated. Upon repeating it thousands of times, one can approximate the area under the curve.

Consider a multidimensional volume V with N random points, x_1, x_2, x_3,… x_n Evaluate the function f(x) at each value of x and average it over N. i.e.,

$$\int_a^b f(x)dx = \frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

--- eq. 3.2

Considering a rectangle of height h and width (b-a) around the area of interest, as shown in the Fig 3-7, and generate n random points such that $p_i = (x_i, y_i)$, where x_i ranges between a and b, and y_i ranges between 0 and h.

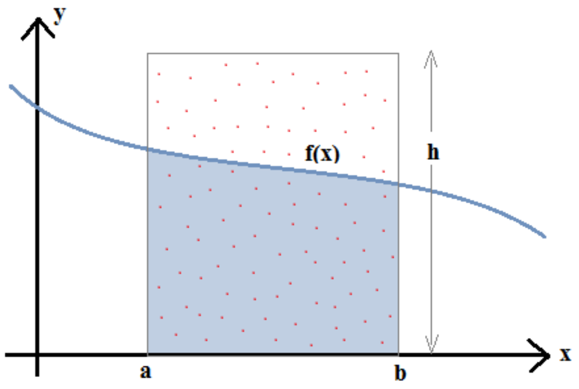


Fig 3-7: Monte Carlo numerical integration

Area under the curve is given by the fraction of the points which satisfy the condition $y_i \leq f(x_i)$ times the area of the rectangle. Out of the n generated random points, if n_x of them come under the curve, then the estimate of the area is given by,

$$\mathrm{F} = \mathrm{h}\,(\mathrm{b}-\mathrm{a}) * \frac{n_x}{n}$$

--- eq. 3.3

A multidimensional region of integration R, a Monte Carlo equation would look like

$$I = \int_R f(x,y,z)\, dx\, dy\, dz$$

--- eq. 3.4

3.3.4 Fault Oblivious Monte Carlo

As we have seen in the previous example, the integral can be easily parallelised since each computation is highly independent of the other and only at the end it is being added. If there are n samples for which the integral has to be calculated, and p processors, then the calculation can be made n/p times faster by sharing the burden equally among the p processes. Only the master process has to run a little more in the beginning and in the end to complete the computation. Due to this property, Monte Carlo applications are categorised as embarrassingly parallel applications. The parallelism can be so easily exploited that, the amount of speed up achieved in computation by increasing the number of processes is almost directly proportional.

When computing a problem with a huge number of samples (n), involving a huge number of processes, each node or the process would be calculating a very tiny part of the final answer. If a failure event occurs in any one of the nodes, we still can end up with an answer which would vary from the actual answer by an insignificant level. Talking in terms of the previous example, if suppose z is the number of the computations which fail and hence does not return a value for the final summation, it would only mean that the Monte Carlo Simulation was carried out for (n-z) samples. Hence, the algorithm is inherently fault tolerant or ‘fault oblivious’ in nature. In case there are a large number of processor failures, it might affect the final computation in two ways-

1. The accuracy of the final result may change if the failure is evenly distributed across the rectangle in which the samples are taken. This is not of much harm as the overall result still remains and the shape of the curve f(x) is not harmed. Hence such kind of failures can be, to an extent, ignored.

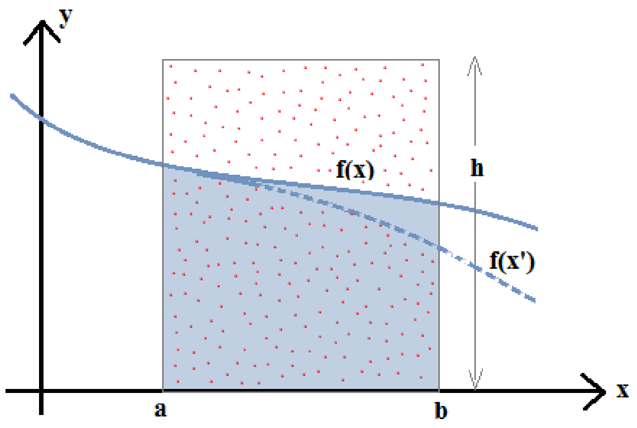


Fig 3-8: Failure which is dominant in one part of the area of interest

2. If the failure occurs dominantly in one part of the rectangle, then the samples for which the result is calculated will not be uniformly distributed across the rectangle. This might lead to the wrong calculation of the final integration. The final result calculated will be for f(x’) and not f(x) as shown in the Fig 3-8. In such situations, the failure events are not independent of the resulting sample values,

and hence have to be taken care of. The failure events must be recalculated for the samples if such failures probabilities are not insignificant to be ignored. [**13**]

The second scenario occurs only when the given area is divided into sub-regions and each region is calculated by one of the processes in the program. Recalculating as mentioned is not easy since the algorithm is very random. However, the random number generators are in reality pseudo random and with the same seed input, they can generate same sequence of numbers. Hence it is of great importance that every worker calculates the answer taking into consideration the entire area of interest and not just one part of the area. If recalculation is necessary to maintain the same accuracy, master node/process should be aware of how many random numbers are needed, how many are lost and how many have to be redone, so that if a failed computations be easily recalculated on new nodes with the same seed. This might however increase the computation time a little. [**13**]

By now one can notice that the algorithm's success greatly depends on the quality of the random number generator which will be used. The numbers so generated should be equally spaced and should not cluster at or around any given point. Other desirable properties include the ability to be reproduced, property of being unpredictable, portable, and requiring less memory for storage.

3.3.5 Error Analysis

Error due to Monte Carlo Numerical Integration cannot be calculated accurately in case of an integrand whose actual value is not known. In this project, the value of π is being calculated, whose value is very well known up to many decimal points and hence it is just an absolute difference of the obtained value and the real value. But if the integration is three dimensional and the area of interest is not easily defined, say for instance, a 3D shape as shown in the Fig 3-9, then the absolute error calculation is not possible. Determining the number of random numbers needed to get the obtained value to a required accuracy level cannot be known definitively.

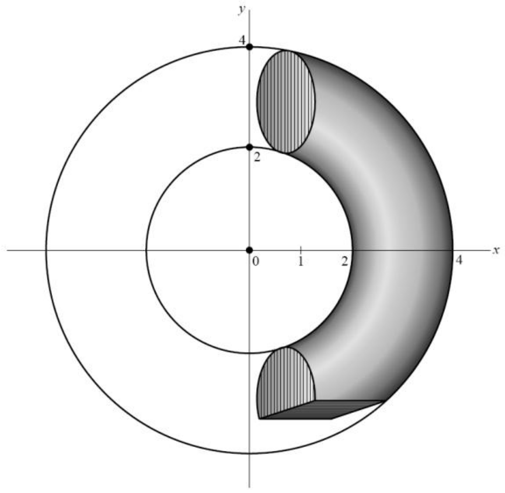


Fig 3-9: A Three Dimensional Numerical Integration

The best that can be done in such cases is to make sure that the actual value V is within a certain range of the obtained value V_n If the integrand were to be a constant, then the obtained value Vn is always equal to V and there is not error. This is the limiting behaviour of the numerical integration. Hence, for an N trial numerical integration, an approximate measure of the error is the variance which is defined by-

$$\boldsymbol{\sigma^2 = \langle f^2 \rangle - \langle f \rangle^2}$$

--- eq. 3.5

Where,

$$\langle f \rangle = \frac{1}{N} \sum_{i=1}^{N} f(x_i)$$

--- eq. 3.6

And

$$\langle f^2 \rangle = \frac{1}{N} \sum_{i=1}^{N} f(x_i)^2$$

--- eq. 3.7

If this translated to the Monte Carlo, considering there were n random numbers generated and M trials taken to estimate the value of a function whose actual value is I. Let I_m be the value the trials resulted in in each trial, as each trial is taken with a different stream of random numbers. According to the limiting behaviour stated earlier, all these values of I_m obtained should be distributed around the actual value I. Taking the differences between these values would give a fair approximation of the mean value I. Hence, taking the standard deviation of the means σ_m^2 which is defined by –

$$\boldsymbol{\sigma_m{}^2 = \langle I^2 \rangle - \langle I \rangle^2}$$

--- eq. 3.8

Where,

$$\langle I \rangle = \frac{1}{M} \sum_{n=1}^{M} I_n$$

--- eq. 3.9

And

$$\langle I^2 \rangle = \frac{1}{M} \sum_{n=1}^{M} I_n{}^2$$

--- eq. 3.10

Though σ_m^2 gives an estimate of the error, it cannot be obtained by making some more measurements, as it is impractical. It can be shown [**14**] that

$$\sigma_m = \frac{\sigma}{\sqrt{n-1}}$$

--- eq. 3.11

And for the high values of n, it can be approximated to

$$\sigma_m \approx \frac{\sigma}{\sqrt{n}}$$

--- eq. 3.12

3.3.6 Monte Carlo pseudo code

Given below are the pseudo codes for a serial and parallel Monte Carlo algorithm, adopted from [**15**]

```
//serial program for Monte Carlo algorithm
monte_carlo_serial()
{
      -> read inputs
      -> Decide on the number of realizations
      -> generate random numbers as per the number required
      -> calculate the properties for the realization
      -> repeat the same to all the realization
      -> calculate the average
      -> exit
}

//parallel MPI program for Monte Carlo algorithm
monte_carlo_parallel()
{
      -> MPI initialize
      -> If my_rank is 0, then
            -> read the inputs
            -> Broadcast the input parameters to all procs
      -> Decide on the number of realizations
      -> generate random numbers as per the number required with a seed
      -> calculate the properties for the realization
      -> calculate the local running average
      -> If my_rank is 0, then
            -> collect properties from all the nodes
         else,
            -> send properties to the rank-0 process
      -> calculate the global average from all the local averages
      -> output the overall average
      -> MPI - finalize and exit
}
```

3.3.7 Monte Carlo Application

As mentioned, Monte Carlo is being used in varied fields of scientific computation. It suits very well for the modern day high performance computation for two obvious reasons, first is the easily parallelisable nature of the algorithm which can be made best use of in grid/cluster computing and the second is that it is fault oblivious.

Monte Carlo is very versatile and popularly adopted method for solving multidimensional problems. While numerical integration is one the primary usage of the algorithm, other notable applications include, Ray tracers in the field of General purpose computing, Expectation Maximization in Machine Learning, MapReduce in Data Base, Option pricing, 'Monte Carlo methods for matrix computation on the grid' [**16**], 'Using Parallel Monte Carlo Methods in Large-Scale Air Pollution Modelling' [**17**], 'Monte Carlo Ising model' [**11**], 'Parallel Hybrid Monte Carlo Algorithms for Matrix Computations' [**18**], are few examples worth stating.

Google has recently developed a framework called MapReduce which works on the principle similar to the Mont Carlo. It is also used in the fields of nuclear medicine, biophysics, computational chemistry and physics in applications like studying the phase and velocity of electrons and other subatomic particles. In Finance, it is used to approximate the value of a stock in near future considering its initial price, the present price, and the path it took to get there. Other application fields include porous media, material sciences, transport theory, nuclear weapon design and computer graphics [**19**].

3.4 Introduction to BlueCrystal Phase 1

BlueCrystal Phase 1 is the university supercomputer with 96 nodes, each with two dual core Opteron processors and 4 thick nodes each with four dual core Opteron processors. Silverstorm Infiniband connectivity is used for the message passing between the nodes. The RAM available on each node is 8GB, divided between 4 cores. The think nodes have 32GB RAM divided between 8 cores. BlueCrystal Phase1 user guide is available online [**20**] for more information.

Chapter 4. Work Carried Out

This chapter outlines all the work carried out to achieve the objectives set out for the project. It explains, in the same order, the choice of MPI implementation, the weak points of MPI programming and ways to avoid them, the software module best suited and finally, the prototype program that was designed for the project.

4.1 Choosing MPI Implementation

The first step of the project was to decide which implementation of MPI to work with. As stated earlier, there are many extensively tested and adopted implementations available today. Few of them are free and are available to download under public licence. OpenMPI, MPICH and its flavours like, MVAPICH, MPICH-V etc., LAM MPI, WinMPI, FT-MPI etc. Few of them are MPI 1.2 implementations while others have implemented the version 2 of MPI specification as well.

4.1.1 Why not FT MPI

As we have seen in the section 3.2.5, FT-MPI offers very good immunity to processor crash. In an n processor program, it claims to withstand up to n-1 processor crashes [**5**]. Since the control was given to the application upon the processor crash, other surviving processors could carry on without being aborted. This was the most important feature required for the project and hence it was the preferred choice in the beginning. Other reasons which made it an attractive option were, there were options to re-spawn the crashed process, and there were error modes like -BLANK, SHRINK, REBUILD, CONTINUE- which comparatively gave a greater control on what we would like to do with the communicator in case of a process crash. It was under public licence and hence available for free download. All these reasons made it the first choice even though it was only an implementation of MPI version 1.2.

The implementation was downloaded on the university supercomputer (BlueCrystal Phase1) and was attempted to install. The installation procedure wasn't smooth and there were a lot of problems. It could only be run interactively and couldn't be submitted as a job in the queue. The documentation was not very well maintained and there was absolutely no help for the problems in their website or in the internet. Even their website was found to be last updated only in November 2003. Hence due to the lack of help and bad documentation, working on FT-MPI had to be given up.

4.1.2 MVAPICH v/s OpenMPI

Amongst the other implementations, the preference was for the one which could perform the best on BlueCrystal. It was also critical that it had comprehensive documentation to go back to and online help in case there was an issue which was missing in FT-MPI. OpenMPI and MPICH were the most popular implantations which fit all the conditions. Out of the many flavours of MPICH, MVAPICH was chosen as it was tailor made for Infiniband (see section 3.4). Comparison was carried on between

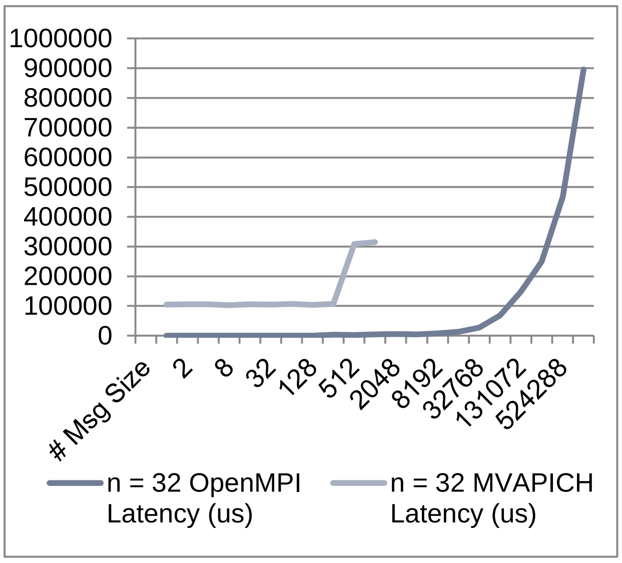
OpenMPI and MVAPICH using the benchmark program OSU Micro-Benchmarks Version 3.3 [**21**]. Table 4-1 shows the list of all the tests conducted for the comparison.

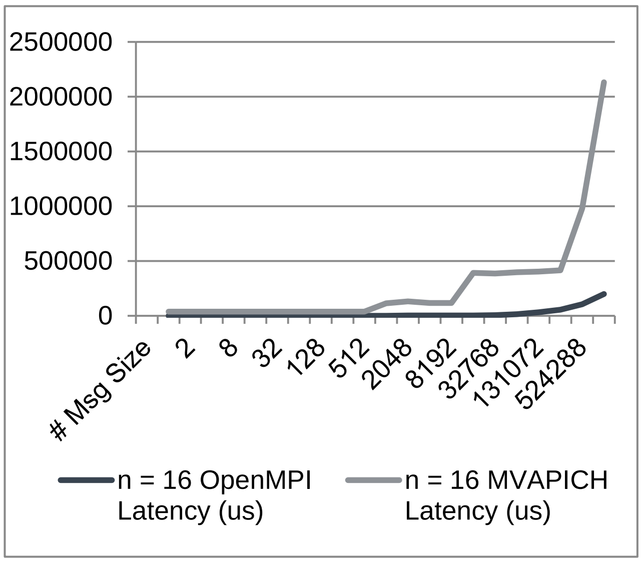
Table 4-1. Tests conducted

MPI Version tested	No. of nodes	Program	Description
MPI 1	any	osu_alltoall.c	MPI All-to-All Personalized Exchange Latency Test
MPI 1	min 2	osu_bcast.c	Broadcast Latency Test
MPI 1	2	osu_bibw.c	MPI Bi-Directional Bandwidth Test
MPI 1	2	osu_bw.c	MPI Bandwidth Test
MPI 1	2	osu_latency.c	MPI Latency Test
MPI 1	min 2	osu_mbw_mr.c	MPI Multiple Bandwidth / Message Rate Test
MPI 1	any	osu_multi_lat.c	MPI Multi Latency Test
MPI 2	2	osu_acc_latency.c	MPI One Sided MPI_Accumulate Latency Test
MPI 2	2	osu_get_bw.c	MPI One Sided MPI_Get Bandwidth Test
MPI 2	2	osu_get_latency.c	One Sided MPI_Get latency Test
MPI 2	2	osu_latency_mt.c	MPI Multi-threaded Latency Test
MPI 2	2	osu_put_bibw.c	One Sided MPI_Put Bi-directional Bandwidth Test
MPI 2	2	osu_put_bw.c	One Sided MPI_Put Bandwidth Test
MPI 2	2	osu_put_latency.c	One Sided MPI_Put latency Test

The tests were conducted for the bandwidth and latency. Latency was tested using blocking send and recv functions for different data/message sizes. The Bandwidth was tested using non-blocking send/recv functions to check the maximum sustained data rate which can be attained.

OSU MPI All-to-All Personalized Exchange Latency Test – test the average time

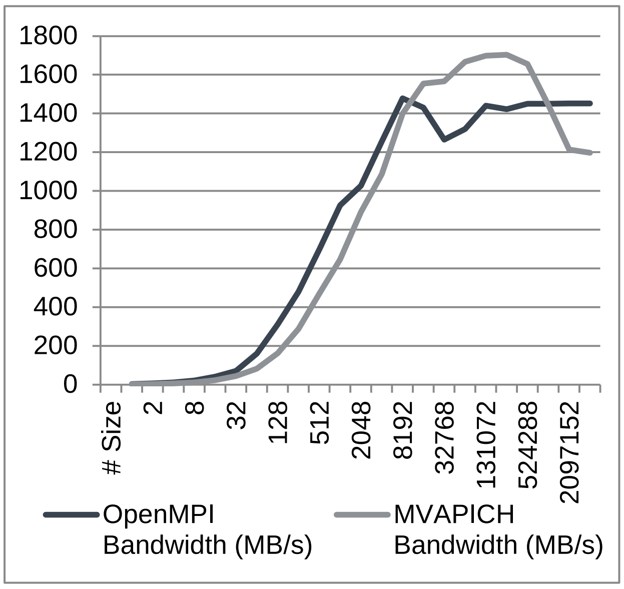


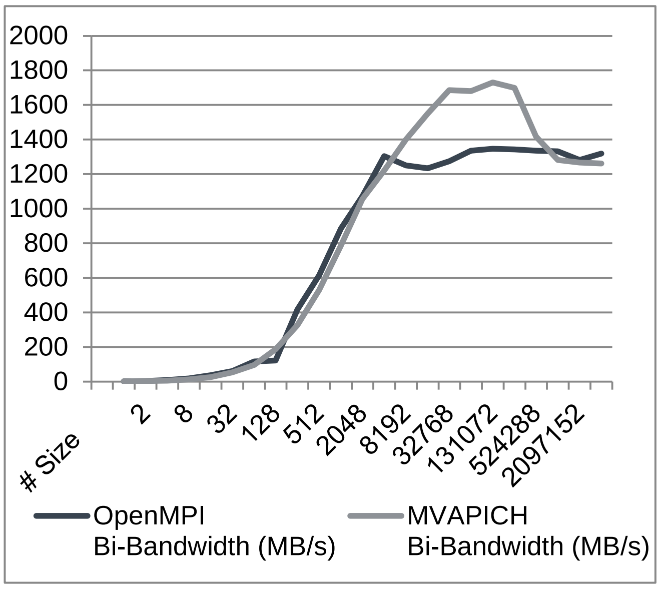


taken for API Alltoall where all the processes call it. Test was carried out for message size 1 to 1,048,576; also varying number of total processes involved 8, 16 and 32.

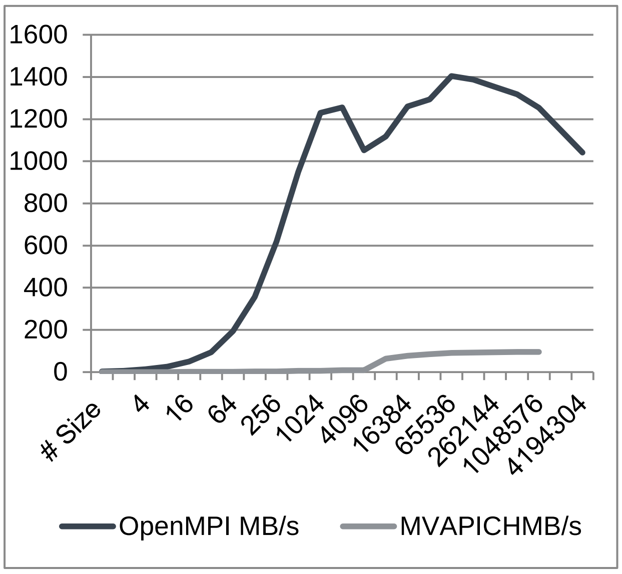
Latency of MVAPICH for 32processes was very bad and after a while it stopped responding. Hence there are no values to plot when the message size crosses 512

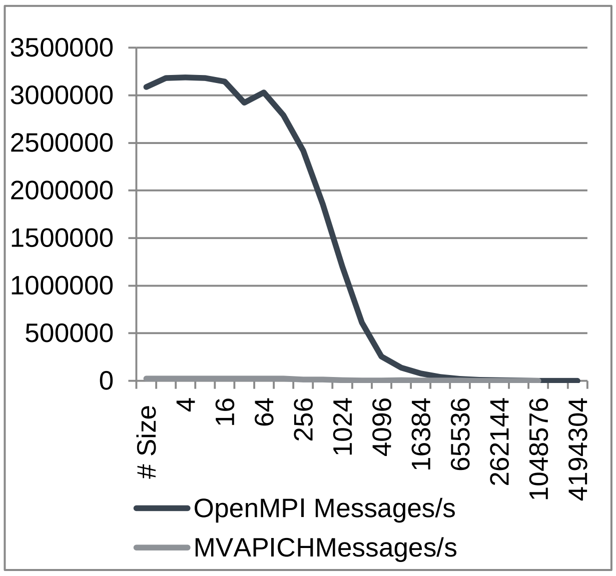
OSU MPI Bandwidth Test – using a non-blocking send and receive the bit rate measurement was carried out in which the sender sends back to back messages and waits for the acknowledgement from the receiver. This was carried out for several iterations and results were plotted for the maximum sustained data rate. The same was repeated, but this time with both the nodes transmitting and waiting for the acknowledgement – bi directional bandwidth test.



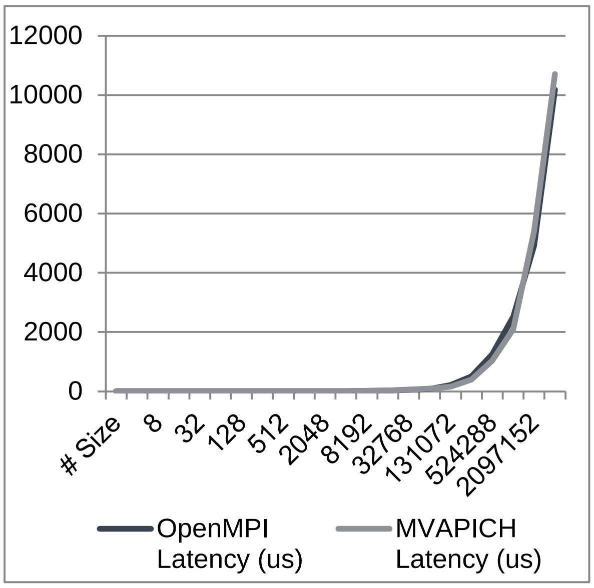


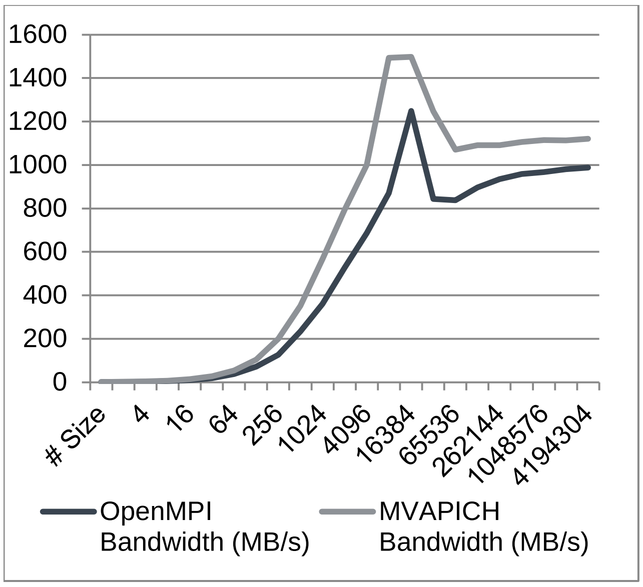
Multi pair bandwidth test was carried out for different number of node pairs – 1, 4, 8 and 16. Each node is paired with another to which it sends back to back messages before waiting for the acknowledgement. This is repeated many times to measure the aggregate unidirectional bandwidth between the pair. Achievable message rate between the nodes is also evaluated. Results again indicated that OpenMPI message rates were much better than that of MVAPICH





One sided accumulate latency was tested for the API MPI_Accumulate by using the combining operation MPI_Sum. One sided bandwidth test was also carried out on similar lines where a fixed number of back to back gets are called and waited until synchronisation occurred.





As it is evident in the plots, the OpenMPI outperformed MVAPICH in almost all the tests conducted. Hence it was decided to go ahead with the OpenMPI implementation. Other reasons for choosing OpenMPI [**21**] were

- Confirmation to MPI-2 standards
- Network is heterogeneity supported
- Thread safety and concurrency

4.2 Identifying weak points of a typical MPI program

First step in writing a fault tolerant program is to identify the weak points which make the program fragile. Two major weak points of an MPI program were identified during the course of the project. This section describes those weak points and then proposes ways to avoid them. A description about how these were used in the project to make it fault tolerant is explained later in the chapter (in the section 4.4.5).

4.2.1 MPI_COMM_WORLD problem

Communicator is a virtual network formed by a set of processes grouped together. Simply put, it is a group of all the processes which can 'talk' to each other. The communication between the processes which belong to the same communicator is called intra-communication. It includes both, point to point and collective message passing between the processes. Each communicator has its own error handler and each process in the communicator has a unique rank. The default communicator is MPI_COMM_WORLD, which includes all the processes in the program. Fig 4-1 shows a conceptual representation of the MPI_COMM_WORLD communicator in a program which has 9 processes working.

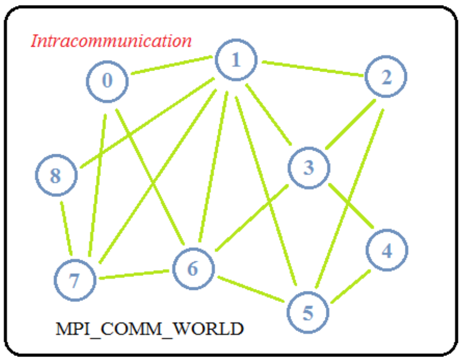


Fig 4-1: MPI_COMM_WORLD, the default communicator

Constructing a program using only the default communicator is a bad practice in MPI programming. It makes the program very fragile. When the processes are grouped together in a single communicator, failure of one process affects all other process due to the collective communication, even when there is no direct connection. Fig 4-2 is a conceptual representation of how the fault spreads across the communicator when there is one fault in message passing. As the state of a process after a fault cannot be determined, all the processes which receive message from the faulty process are also considered to be in a state of fault, which eventually makes all the processes faulty. Thus the program has to be aborted to start the computation fresh again.

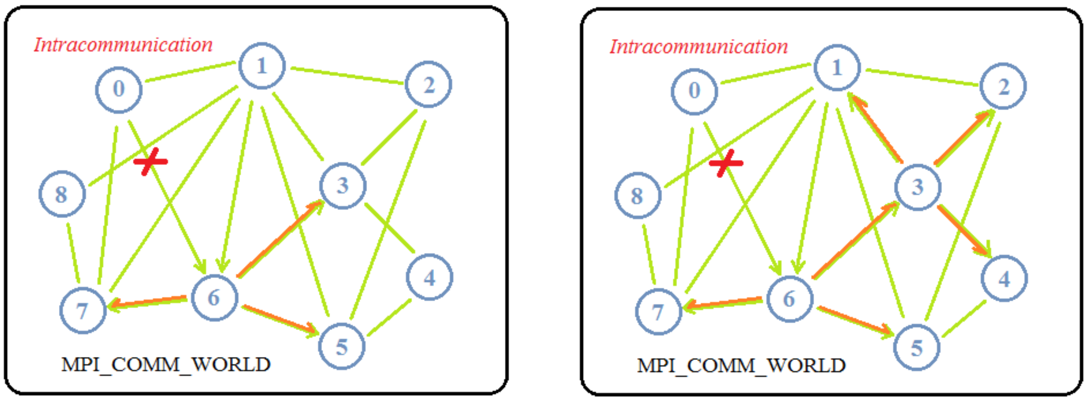


Fig 4-2: How the default communicator makes a program fragile

Using Intercommunicators is the best way to avoid this problem. When a few processes in the program are grouped together to form a communicator, each process in the communicator is again ranked from 0 to n. one-to-one communication that happens between two processes which belong to different communicators is called intercommunication. The group which initiates the communication is called the local group and the one which responds is called the remote group. Such a communicator is called 'intercommunicator'. Fig 4-3 is the conceptual representation of the intercommunication. Numbers inside the circle show process' rank in the default communicator MPI_COMM_WORLD. Numbers outside the circle show the process' rank in the new communicator they are grouped into.

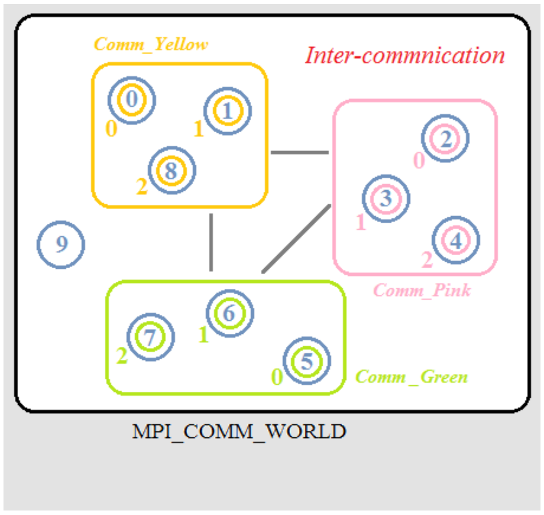


Fig 4-3: Intercommunication

By using intercommunication, the fault can be localised. When a fault is detected in the process, by not communicating with that particular group, rest of the processes can go ahead with the computation. This is a strong point when it comes to implementing a fault tolerant program. Quality implementations like MPICH and OpenMPI do not have any overhead for intercommunication as compared to normal intracommunication. It is as safe and efficient which makes it an interesting option.

There are limitations to this however. It can be used only when computation of the other groups do not depend on the results/messages of the faulty group. Each group would have to be working with an independent chunk of computation. This can be reassigned to one of the 'living' groups if there was a fault detected in one of the communicators. This factor limits the use of intercommunicators, yet it a powerful tool that can be used when there are jobs that could be parallelised.

4.2.2 Error Handling in MPI

Error handling as given in MPI specification is rather straight forward. Every communicator is associated with an error handler, a function to which the control has to be transferred, in case an error was detected by the program. For any newly created communicator, the default error handler is MPI_ERRORS_ARE_FATAL, [**22**] [**2**] which aborts the program once an MPI error occurs during the computation. When the programmer doesn't wish to do anything when an error occurs, he/she can simply let the default error handler be, and the function aborts as if the MPI-Abort was called by the process which invoked the handler. Most of the programmers let it stay this way, which makes the program susceptible to halt in every error. The program has to be started all over again, which means all the precious computation done till then is lost.

A little more flexible way is to set it to MPI_ERRORS_RETURN in which case, the handler returns the error code to the programmer and he/she has to decide how to handle it. In any communicator, the handler can be set as required by calling the function MPI_Errhandler_set:

```
int errCode;
errCode = MPI_Errhandle_set( MPI_Comm communicatorName,
          MPI_Errhandler errHandler);
```

With MPI_ERROR_RETURN, the program will return the error code when the error occurs. The returned error code is implementation specific and its meaning could be learned by calling the routine MPI_Error_string. MPI also defines the Error Classes to which the error code belongs. This can be said to give a general classification of the error. One can know the class of an error code by calling the routine MPI_Error_class.

```
char errMessage[MPI_ERR_STRING_BUFSIZE];
int errMsgSize, errClass;
MPI_Errhandler_set(MPI_FIRST_COMM, MPI_ERRORS_RETURN);

errCode = MPI_Scatter(sendData, sendCount, sendType, recvData,
            recvCount, recvType, root,  MPI_FIRST_COMM);

if (errCode!= MPI_SUCCESS)
{
   MPI_Error_class(errCode, &errClass);
   MPI_Error_string(errClass, errMessage, &errMsgSize);
   fprintf(stderr, "MPI_Scatter error class: %s\n", errMessage);

   MPI_Error_string(errCode, errMessage, &errMsgSize);
   fprintf(stderr, "MPI_Scatter error code: %s\n", errMessage);

      /*.. Do the clean-up...*/

   fprintf(stderr, "MPI_Scatter exiting..\n");
   MPI_Abort(MPI_FIRST_COMM, errCode);
}
```

The main purpose of this error handler is to give the programmer a chance to do the necessary things before the program aborts. It may be anything like ending a file, save the status i.e. checkpoint so that one can start over at the place it was aborted, etc. The program need not essentially abort, if the programmer is sure of the implications of the error and has a complete track of what part of the program it can affect. He/she can also decide to proceed with the computation by taking necessary action when the control returns to the program. This is another powerful tool which can be used when there is little dependency between the processes in the program, like in an embarrassingly parallel program.

Note that, though MPI standard defines only two handlers, implementations are at liberty to define and implement their own error handlers. [**22**] FT-MPI defines extra error handlers which gives it a unique feature of handling process crashes. The MPI forum is working on releasing another version of MPI, the MPI-3 in which error handling and fault tolerance will be major areas of interest. There is working group dedicated to this topic [**23**]. Extending MPI is also another important area in which research is going on.

4.3 Programming Model for Fault Tolerance

In an MPI program, all the processes including the rank 0 process can be involved in the computation and there need not be any process dedicated to keep track of all the processes. All the processes could do input/output functionalities. The results could be accumulated using all-to-all function. Other collective communication functions can also be used. This adds to making the computations faster as there will be an extra process working. However, this makes the program fragile to an extent because the moment one of the processes crashes or there is a communication error, the whole set up has to be shut down and restarted if not check pointed. Besides, latency of the functions like all-to-all increases exponentially as the message size increases. This would effectively slow down the program for bigger message sizes.

In a typical server-client connection pattern one can see that the loss of a client does not affect the server or any other client connected to the server. Server keeps servicing the other clients who are alive. The crashed client is just neglected.

The same idea could be applied to make the MPI program similarly fault tolerant. A system could be designed such that –

- All the worker processes are connected to the rank-0 server process, which starts and wraps up the MPI program.
- Only one-to-one communication with the server is allowed for the workers
- Work is broken down into smaller chunks called work items
- Server maintains list of work items, their status and status of each worker
- Server distributes work item to each worker, collects results and upon the completion of all the work items, computes the final output
- Worker receives the work item, computes the result and sends it back to the server. It stores the state of present computation only.
- Workers do not talk to each other. They are not aware of other workers.

Fig 4-4 shows the concept of the server worker model for fault tolerance. One can see that the error would be localised and a process crash does not affect the working of the server or any other process in the system. Hence it is more fault-tolerant than the usual way in which all the processes are used for the computation.

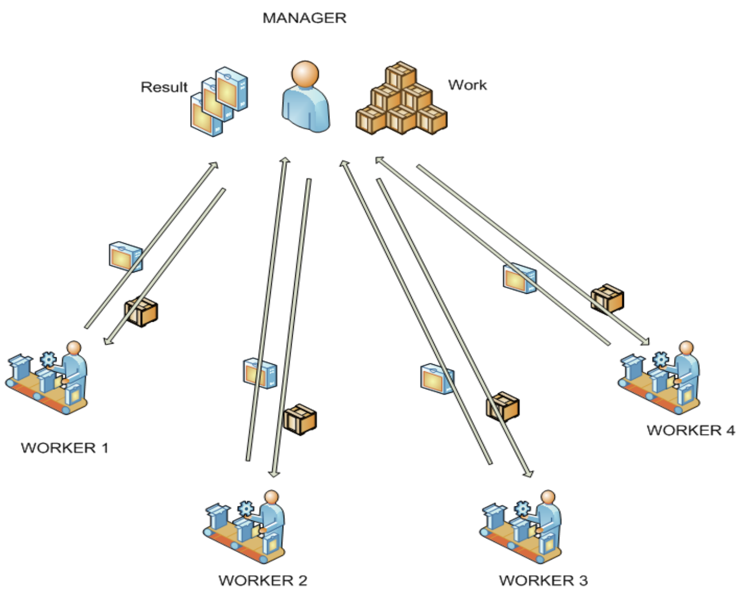


Fig 4-4: Concept of Server Worker programming module

If a worker process crashes, there are two ways in which it could be handled-

Since the server has the details of the work assigned to each worker, the work that was assigned to the crashed worker could be re-assigned to one of the living workers in the system and it has to be computed from the scratch.

The work item that was assigned to the crashed process could as well be neglected, to carry on with other work at hand with all the other workers alive. This can be done only when there is no dependency on the work item lost. Monte Carlo is one of the embarrassingly parallel algorithms which can afford this luxury. Hence this was adopted for the programming in this project.

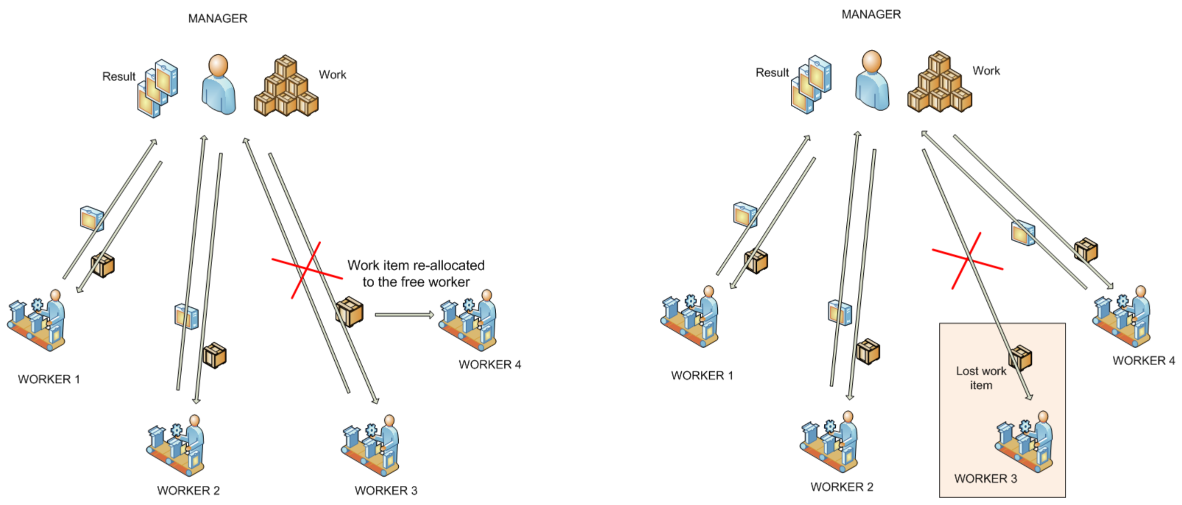


Fig 4-5: (a) work item reassigned to living worker; (b) work item considered lost

4.4 The Prototype Program

This section describes the concept of pi calculation, the random number generator and the algorithm used. All the issues faced during the implementation time are also mentioned in the respective sections with the methods used to overcome.

4.4.1 Monte Carlo Numerical Integration

The objective of the project was to write a prototype program for numerical integration. Calculation of the value of π using Numerical Integration was chosen for the prototype. Main reason for choosing π calculation was its simplicity. Thus it would give us more space to concentrate on making the program fault tolerant, which was the prime focus of the project. Though it was quite a simple problem that could be solved using Monte Carlo Numerical Integration, it was strong enough to model other real world problems which rather made it more attractive option.

The task to be carried out was simple. π is nothing but the ratio if the circumference of a circle to its diameter. A circle inscribed in a unit square was considered as shown in the Fig 4-6.

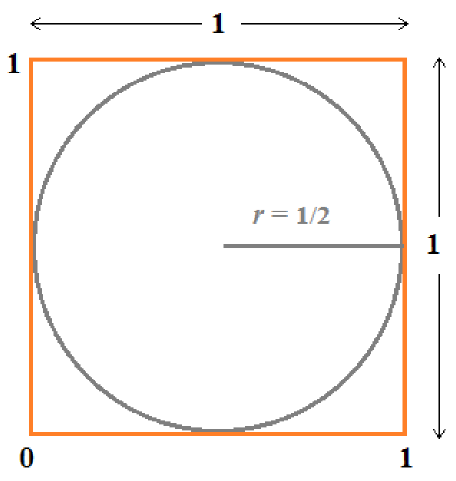


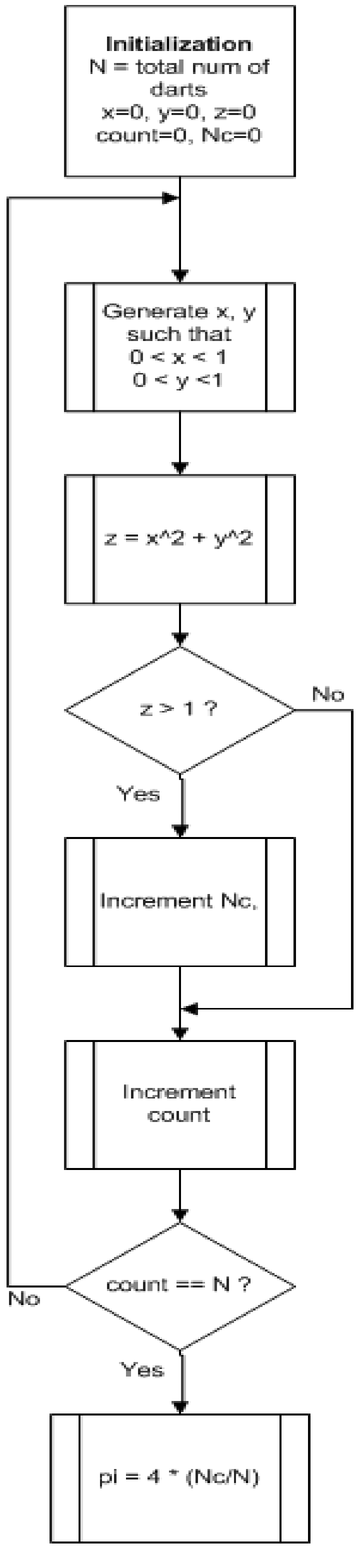
Fig 4-6: Unit square and inscribed circle

The radius of the circle will then be $\pi\left(\frac{1}{2}\right)^2$ or $\left(\frac{\pi}{4}\right)$. The ratio of the area or circle to the area of the square will be $\left(\frac{\pi}{4}\right)$. Now, if we generate thousands of random points anywhere in the unit square, the ratio of the points which fall inside the circle to the total number of points generated would give us the value of $\left(\frac{\pi}{4}\right)$. And hence value of π would nothing but the ratio multiplied by 4. Now, if a few random points are generated in the unit square, the ratio of number of points inside the circle to the total number of points generated shall give an approximate value of $\left(\frac{\pi}{4}\right)$. Hence,

$$\boldsymbol{\pi = 4\ x\ \left(\frac{no.\ of\ points\ inside\ the\ circle}{total\ no. of\ points\ generated}\right)}$$

--- eq. 4.1

The logical flow is as represented in the flow chart



x and y are the co-ordinates inside the square. They are randomly assigned a number between 0 and 1. These random numbers are generated using a random number generator function. Nc is the number of points which are inside the circle. To test if the point thus generated is inside the circle, each value is squared and added. This is nothing but finding the distance of the point from the centre of the circle whose coordinates are (0,0). If the value thus obtained is lesser than 1, it is inside the circle. Finally, the value of π is calculates using the--- eq. 4.1

This is the concept of numerical integration using Monte Carlo. As the number of points generated increases, the accuracy of the result also increases. i.e., in Fig 4-7, instance (b) would give better approximation of π than instance (a). However, the error percentage (E) and the number of points generated (N) are related by E $\alpha \left(\frac{1}{\sqrt{N}}\right)$as seen in --- eq. 3.12. Hence the increase in the points generated has to be significant for it to show up in the accuracy.

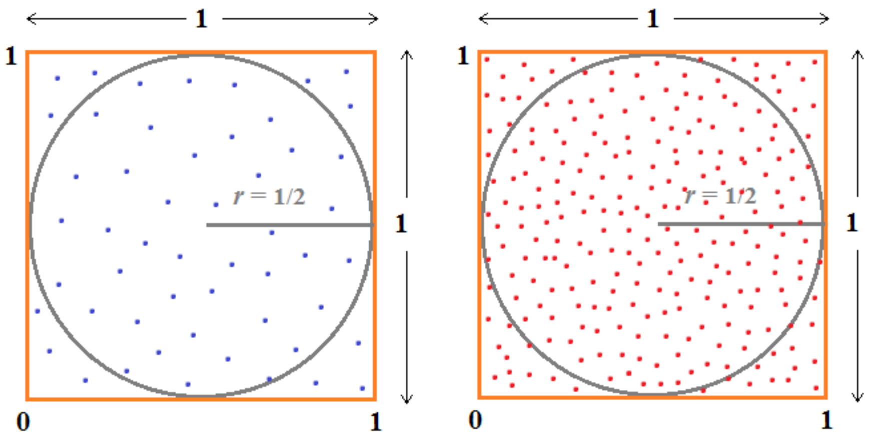


Fig 4-7: number of darts directly proportional to accuracy (a) and (b)

One can see now how easily it can be parallelised. The work of generating N random numbers could be easily divided between p processes and a speed up of p folds could be easily achieved as there is little or no dependency between the processes. There is absolutely no communication required between the participating during the execution time. This is possible due to the embarrassingly parallel nature of the algorithm. This is the property we are exploiting in this project in designing a fault tolerant Monte Carlo algorithm. This is discussed in the section 4.4.4. However, the accuracy has a major dependency on the quality of the random numbers generated. This is discussed in the following section 4.4.2.

4.4.2 Choose the RNG

The quality of the results of any Monte Carlo computation is highly dependent on the quality of the Random Number Generator used in the calculation. The usual rand() function in C cannot be used as it gives the same random number sequence in all the processes, which would turn an n processes parallel program into an (n-1) modular redundant serial program. To be successfully parallelised, Monte Carlo simulations need highly independent random number sequences while computing. Overlapping random number stream should never be used as this might lower the quality of the results and especially in some computations like studying the electron velocity/phase etc. yield outrageous and erroneous results [**29**].

To generate independent random numbers on each of the processes, different ideas were tried out to see what works. They are-

1. Generate all the random numbers in one process and send it as a message to all the participating processes to compute. This slowed down the application terribly. Another drawback observed was, the nodes were not working independently anymore which was against the concept of the project.
2. Jump a known distance in a sequence hence ensuring independent numbers. This gave the random number we desire but the burden of computation and jumping fell on the programmer.
3. Different generators war used in each of the process. This again added to complicating the program.
4. Simply make all the processes chose distinct seed and generate random numbers. This does not yield the streams as good as a single stream yielding random numbers. The streams may overlap if the seeds are wrongly chosen.
5. A PRNG with thousands of different seeds taken from time of the day was used. However, it was similar to a true RNG. Re-generating the same stream in the same order was impossible. Storing each of the random number generated for future reference was out of question as millions of random numbers were used and it was nothing but a waste of memory and computation time for storing and maintaining a database.

What we desire is a library with a function or an API which would return highly independent non overlapping stream of random numbers on each of the nodes, without being a burden on programmer. It has to be a deterministic PRNG, i.e., the random numbers should be re-generated if there is a need, may be for testing/debugging purposes. Very important properties of the parallel random number generators are-

1. Generator should be able to deliver independent stream regardless of how many processes are in the program.
2. Every sequence in each of the processes should be as good as the random numbers generated on a single process.
3. There should not be any relation between the random numbers of any two processes in the program.
4. The algorithm used should be efficient and ensure that the random numbers are equally distributed, never repeat numbers, reproducible, portable, has a large period and can be generated rapidly.

There were three options to choose from-

1. **RANDOM_MPI:** This makes use of Linear Congruential Random number Generator or "LCRG" which works on the formula U = (A * V + B) mod C. every p-th entry of the original sequence is calculated by each process. It is available under GNU LGPL licence in C, C++, Fortran77 and Fortran90. The installation is cumbersome. The program slowed down. the random numbers generated overlapped after some time and hence the results obtained was not

great. Hence this method to generate independent random number streams was rejected.

2. **Dynamic Creator Mersenne Twister (DCMT):** It promises independent random numbers on each node with a dedicated algorithm for the same. MT helps in distributing tasks while DC acts as a constructor which creates the instances of MT on each node to produce independent streams of random numbers. The period of the MT could be programmed and DC could also be changed on the runtime. It is straight forward to use as the 'readme' document very clearly explains the installation and usage. There are few examples which are quite useful. The problem which I faced was not being able to set the limits for the numbers generated. Random numbers needed for this project had to be a floating point, ranging between 0 and 1. For that, one needs to know the highest possible value that can be generated using the RNG. Assuming word size = 32 (this can be set in the program), 2^{31}-1 which is 2147483647 did not yield the required result. Hence this method had to be abandoned.

3. **Scalable Parallel Pseudo Random Number Generators Library (SPRNG):** Finally this library was chosen to work with. It has a very good set of documentation and lots of links which helps in installation and working with. There are numerous examples provided, and they are really helpful in understanding the usage and properties of the RNG. Another attractive thing about this RNG is that six different random number generators are implemented in it and the one we want to work with can be chosen at the beginning of the program. There is no burden on the programmer or the program. The random numbers generated are fairly independent.

4.4.3 Working with SPRNG

Working with SPRNG is fairly simple. Version 2 was chosen as it was a C implementation. The header file sprng.h was added. As we use a lot of random number streams in the program, it is required that each of it be un-coordinated as much as possible. This property is called independence of the random number streams and sprng library provides it. The initialisation is done in the INIT stage (see section 4.4.4) of each of the worker. init_sprng() has to be called for initialisation with–

```
int *init_sprng(int rng_type, int streamnum,
                int nstreams, int seed, int param)
```

- rng_type = argument to set the generator type. It can be anything between 0 and 5. 4 was chosen to pick the Multiplicative Lagged Fibonacci Generator. Other generator available are Combined Multiple Recursive Generator(0), 48 Bit Linear Congruential Generator with Prime Addend(1), 64 Bit Linear Congruential Generator with Prime Addend(2), Modified Lagged Fibonacci Generator(3), and Prime Modulus Linear Congruential Generator(5).

- streamnum = argument to set the stream number. The rank of the process is used here to have a clear idea of which stream belongs to which rank
- nstreams = total number of streams generated. The world size was used for this as we generate one stream in each process
- seed = seed in sprng is used to encode the starting state. Different streams initialised with same seed will essentially have different numbers. This was defined as a macro. With same seed, every run gives the exact same stream for each of the streamnum. In case different streams are needed for different run, the value of the seed has to be changed. This ensures the property of reproduceability of the random numbers as mentioned in the section 4.4.2.
- param = this refers to the parameters of the generator. The default sprng parameters were used in this project.
- Return value = an integer pointer stream_id, which points to the stream of the random numbers generated.

After initialisation, all that has to be done is calling the function sprng() with stream_id as the argument every time a random number was needed. Hence in the RUNNING stage of the worker, this function is called. The number generated is a decimal between 0 and 1 which is what we require. Integer random numbers can also be generated in sprng using the API isprng().

4.4.4 The Algorithm

With the background of all the concepts which have been discussed till now in this report, the final prototype is described in this section. The high level design of what the final project looks like is shown in the Fig 4-8.

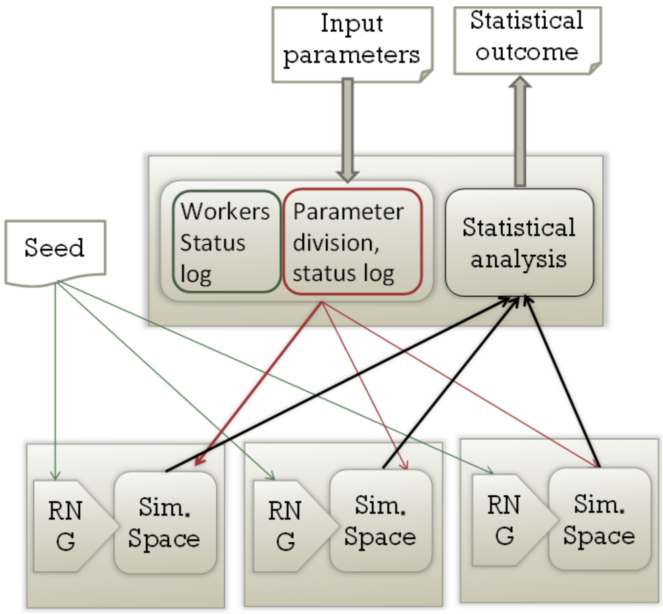


Fig 4-8: High Level Design of the prototype program

There are two parts in the system, server and worker. Rank-0 process is made the server and rest of the processes are workers. The seed is given to each of the worker during the start of the program (it can be defined as a macro as it has been done now). The same seed is given to all the workers. Seed value can be changed if different streams of random numbers are needed.

The server process does not do any calculation. It is the only process which communicates with the outer world. It is responsible to take the parameters the user gives, and give the result to the user. Inside the system, it is responsible to break down the total work into small pieces called work items and maintain the status log of all the work items. Distribute the work items to all the workers alive and collect results, track each worker and maintain the status log of them which is updated often. The status of the work and workers are updated every time there is a communication event in ISSUE and COLLECT states. Fig 4-9 shows the state machine of the server and the Table 4-1 gives a detailed description of each state's entry and exit criterion and the task carried out at each state.

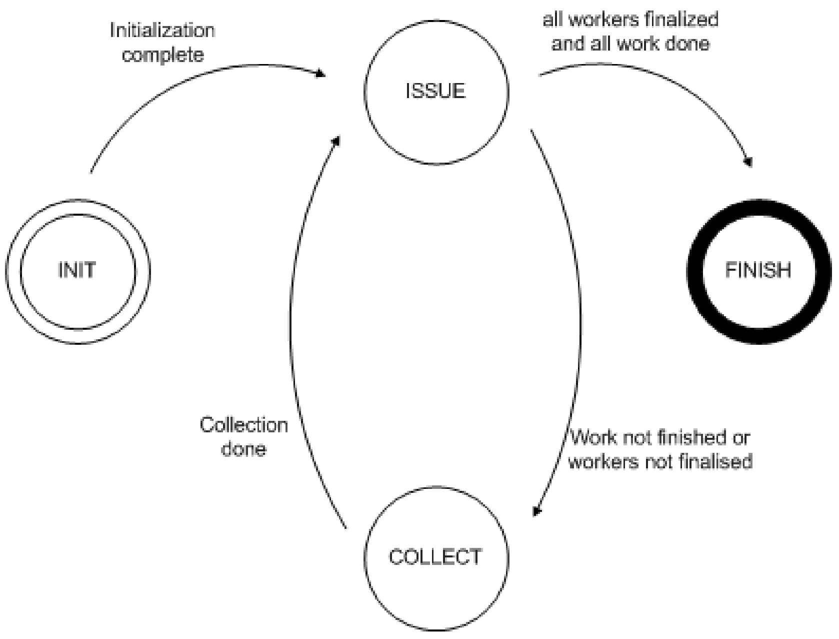

Initialization complete
ISSUE
all workers finalized and all work done
INIT
FINISH
Collection done
Work not finished or workers not finalised
COLLECT

Fig 4-9: Server state machine

State	Entry	Exit	Task
INIT	Default first state, when the execution starts	When the initialisation is complete	Create the intercomm for all the workers Set the error handler for each intercomm
ISSUE	When the INIT is complete	when issuing work to processes is complete	Send work to living processes Update the work and worker status Check for workers alive and work not done
COLLECT	When ISSUE is complete if there are living workers or some undone work	When collection of the result is complete	Receive results from the workers If not received, check for timeouts Update error condition if timed out

State	Entry	Exit	Task
FINISH	When all the work is complete or all the workers are either dead or finalised	This is the final state and the execution culminates	Release all the communicators Calculate the final result and the time elapsed Print the result and the time taken

Table 4-2: Server state machine

Only the workers do all the calculation necessary. They keep waiting for the work item from the server in a loop with the exit condition of a receive timeout. Value of the receive timeout is a tricky thing to be set, as it should be long enough to run till the server finalises but not too long after the server finalises. On receiving the work items, workers proceed to compute the result. At any time each worker has information of only a small amount of the total work. Neither the work nor the result is stored in the worker. Upon any error in the communication or if the receive timeout occurs, the worker goes into the ERROR state and finalises. The worker is not re-spawned or re-initialised after that in the run. Fig 4-10 shows the state machine of a worker and

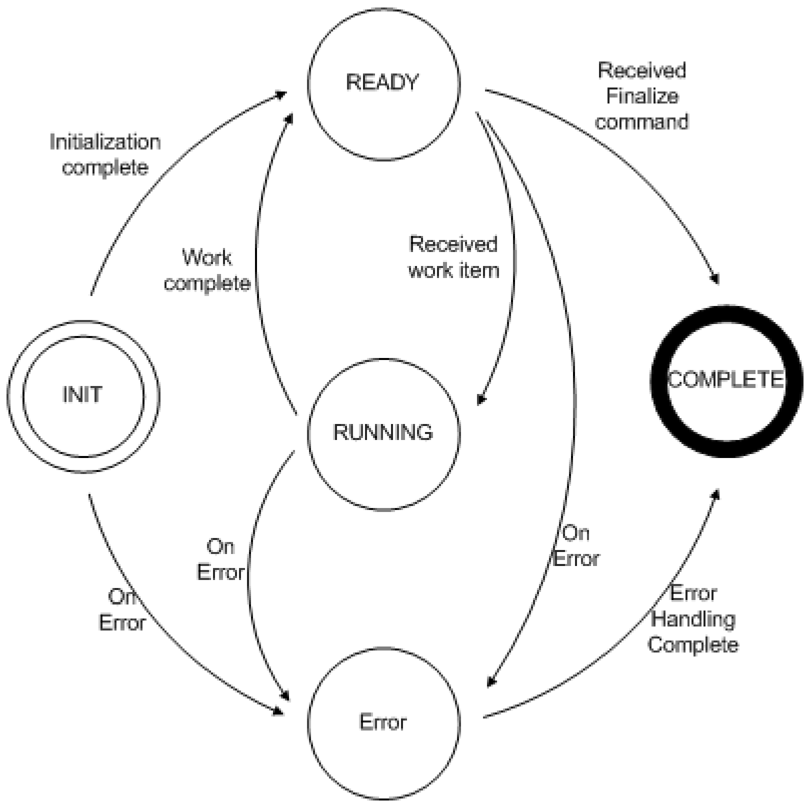


Fig 4-10: Worker state machine

State	Entry	Exit	Task
INIT	Default first state, when the execution starts	When the initialisation is complete	Create the intercomm with server Set the error handler for the intercom Initialise SPRNG
READY	When the INIT is complete	After the work is received from the server	Wait on the work to be received and check for time out If work is not received till the timeout, trigger an error case

State	Entry	Exit	Task
RUNNING	When work item is received	When the result is sent to server	Compute the result for the received work item Send the result back to server Trigger error case if the communication fails
ERROR	When error occurs on any state	After the error handling is complete	Print the error case
COMPLETE	When the processes receives the end command from server In case of an error, after the error is handled	No exit, this is the final state of the run	To release the communicator and proceed to finalise

Table 4-3: Worker state machine

4.4.5 Program for Error Immunity

For achieving the error immunity in the system, intercommunicators were created between the server and each of the workers. The error handler for each intercommunicator was set to MPI_ERROR_RETURN as discussed in the sections 4.2.1 and 4.2.2. This would make sure the program does not abort automatically and the control returns to the application where it can be handled.

When there is a problem with the MPI_Send, the sending process will instantly know because the return value will not be MPI_SUCCESS. Hence this situation could be handled at the sender end gracefully. However, the receiver will not know that the send has failed and it will wait for the message indefinitely. To avoid this MPI_Irecv is used instead of MPI_Recv. MPI_Irecv is a non-blocking receive. The program returns before the complete message has been received and it can be checked later to see if the message reception is complete. A receive timeout is set, after which the receiving process decides that the sending is failed and goes into the error handling phase where the required clean-up is carried out.

When there is a problem in the receiving end, the sender still thinks the message is sent while the receiver instantly knows that the error has occurred through the return value of the MPI_Irecv. Hence it is easy to handle at the receivers end however it is a tricky situation at the sender's end. In the project, the algorithm is written such that each 'send' is acknowledged by a 'receive'. If the sender does not receive the next message, it again waits till the receive time out and then goes into the error condition. At the server end, when server sends the work item, the acknowledgement is the result update from the worker. At the receiver end, when it sends the result, the acknowledgement is the next work item or a Finalise command from the server. Both of them have a logical receive timeouts values.

Fig 4-11 shows how the implicit acknowledgement works in the program. The solid lines represent the actual message transfer; the dotted line shows what triggers the message transfer. If a solid line does not happen due to a fault in the system, the trigger is not created for the next message transfers, thus no further message transfers happen. Error condition is executed at both the ends. Worker immediately finalises sensing the error and the server marks the corresponding intercommunicator as faulty and stops any further communication with that intercommunicator.

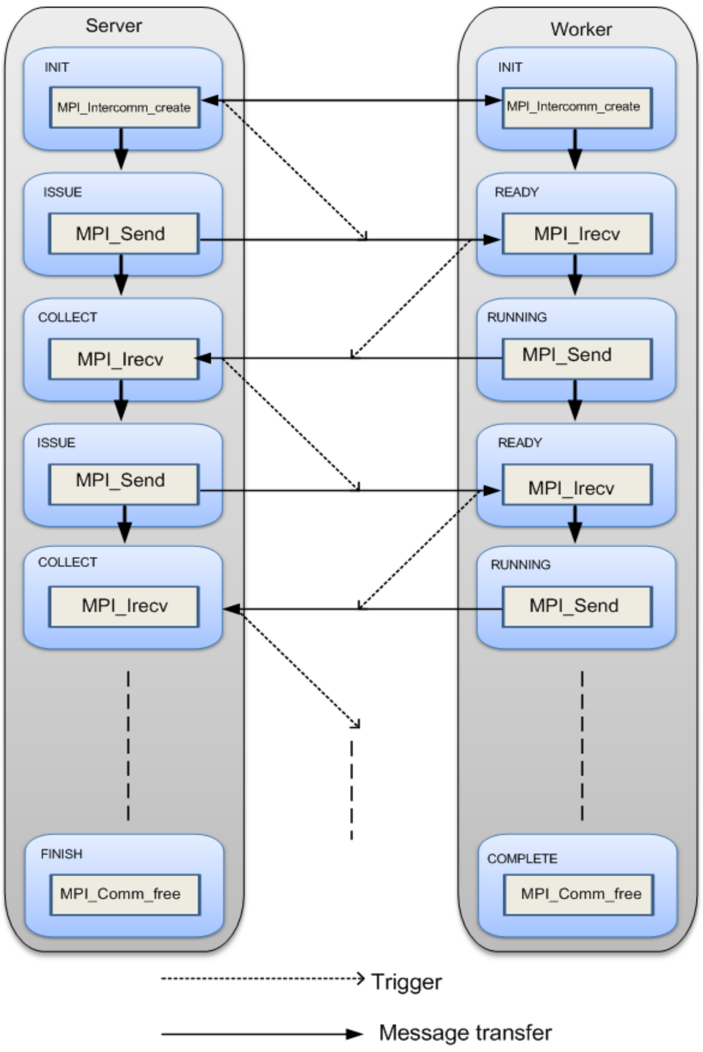


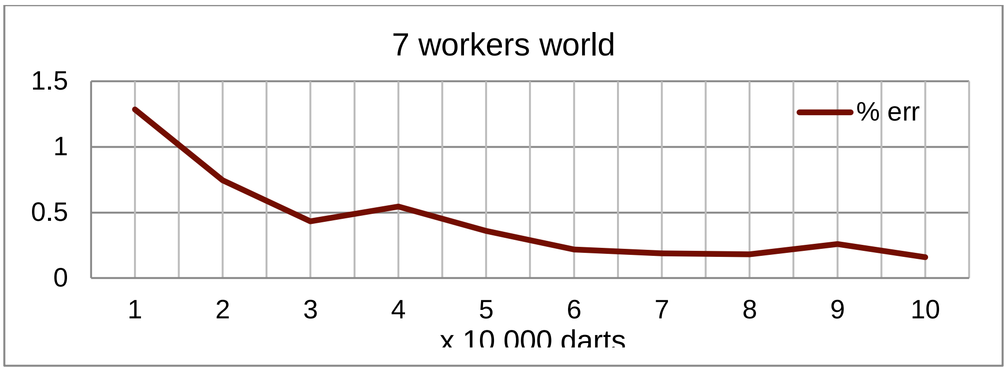
Fig 4-11: Nature of Implicit Acknowledgement in the program

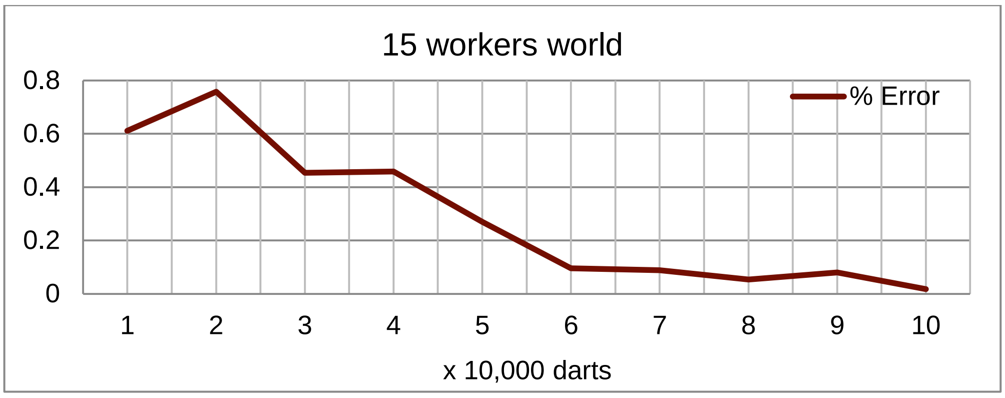
Since the Monte Carlo algorithm is inherently fault tolerant, upon an error, the corresponding intercommunicator can be released and the respective process can be assumed to be dead. The work item with which it was working at the moment of death can be labelled as lost. Due to the virtue of inherent fault tolerance of Monte Carlo algorithm, there is no need for the lost work items to be reassigned, as it does not make noticeable difference to final results. The results are discussed in Chapter 5.

Chapter 5. Results and Analysis

Experiments were carried out with the program with various inputs to test if the project objectives were achieved. To prepare for the fault tolerance test, initially it was made sure that the RNG and Monte Carlo were working exactly as desired in the project. The project was later tested for fault tolerance in application domain at both program level and algorithm level. The assumptions made during the design time regarding the accuracy and computation time were tested to achieve the best trade off.

5.1 Test for the RNG





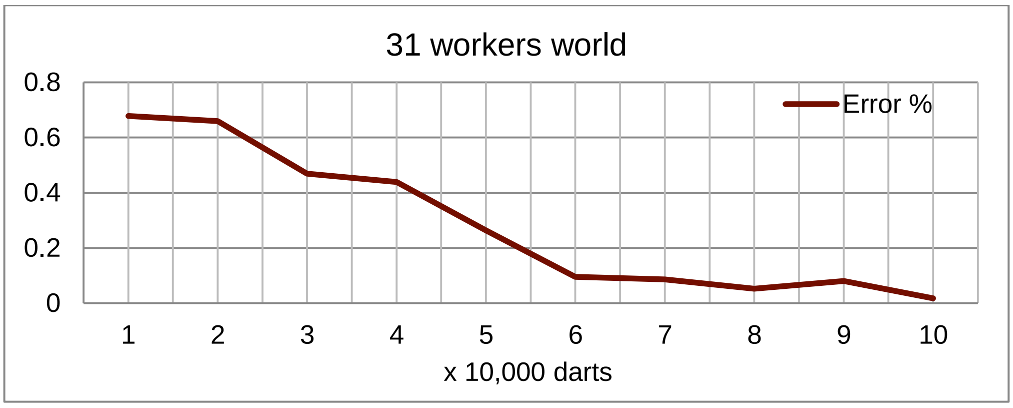


Fig 5-1: Work division for RNG testing

As the total number of darts (random numbers generated) increases, the error factor decreases and hence the accuracy increases and so does the computation time in a serial program. By making it a parallel program, the work is divided between n processors, and hence the computation time is reduced by the factor of n. In case of a single processor the random numbers generated are not correlated or overlapped. This property has to be maintained while sharing the work amongst many processors. The problems faced due to the use of serial RNG in the parallel system were discussed in the section 4.4.2. Hence the most important factor before proceeding with other test was to make sure that the sprng library generated independent streams of random numbers.

To test the nature of random number stream in each of the workers, a work division test was conducted. The number of darts was kept a constant, and was run on a world with 7 workers, 15 workers and 31 workers, with each worker having its own independent stream of random numbers. The behaviour of the error percentage variation over the increase of number of darts did not change across the three (Fig 5-1), confirming the independent stream of random numbers. If there were to be any overlap or correlation among the streams, the results would not vary with the number of darts almost exponentially as seen in the Fig 5-1. The time taken to compute the same result reduced drastically compared to each other, as shown in Table 5-1.

No of darts	7 workers	15 workers	31 workers
1000 darts * 100 work items	30.100588	14.092136	8.080371
1000 darts * 1000 work items	286.639903	134.352388	66.218469

Table 5-1: Computation Time

Thus it is quite evident that the work load sharing is done and there is no redundancy of work in the system as discussed earlier in section 4.4.2.

5.2 Error percentage variation

The next step is to prove that the assumption we make while developing the project –i.e., 'Monte Carlo is fault Oblivious', holds for the fault tolerant program designed. For this, the effect on the resulting error percentage due to the increase or decrease of the total number of darts should be known.

The measurement is not straightforward. One trial per dart number does not give the correct result right away as the readings vary in every trial even when all parameters are kept unchanged. This is because of the high randomness involved in Monte Carlo. The Monte Carlo numerical integration yields only an approximation of the actual answer, and all the readings are scattered around the actual answer, as discussed in the section 3.3.5. Hence to compare them, 20 trials were conducted for each case, and the mean was calculated by –

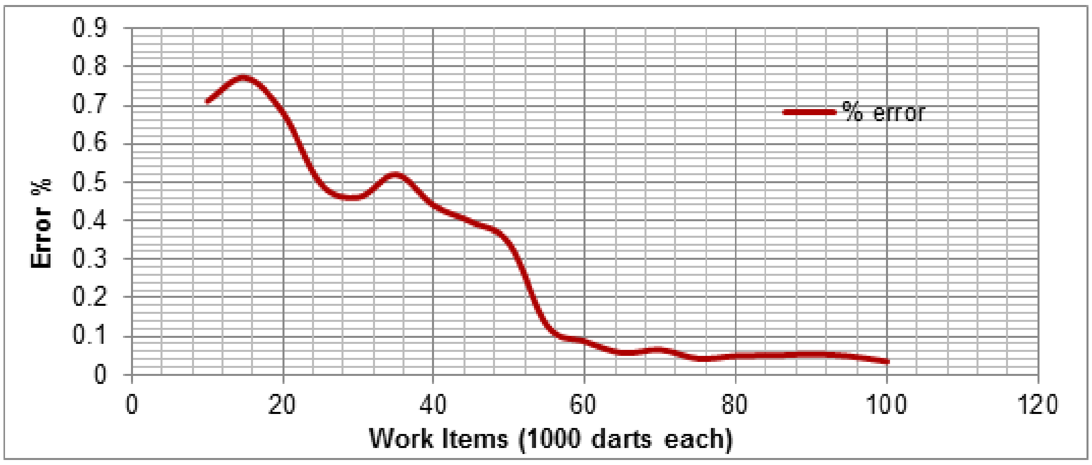
$$\mu = \frac{1}{N}\sum_{i=1}^{N} X$$

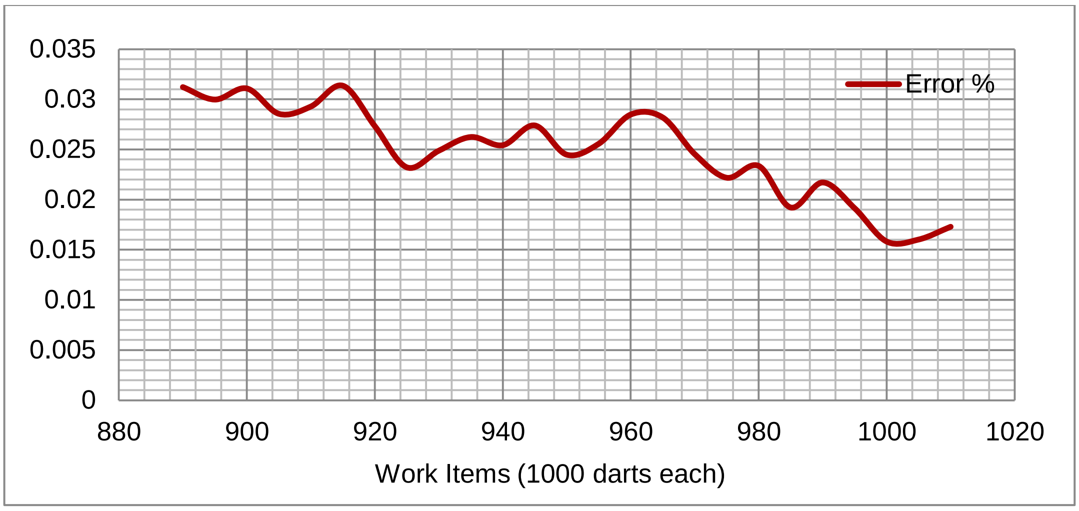
--- eq. 5.1

Where N = number of trials, X = result for each trial. With the obtained mean, the percentage error was calculated by -

$$\%\ \text{error} = \left(\frac{|\pi - \mu|}{\pi} * 100\right)$$

--- eq. 5.2





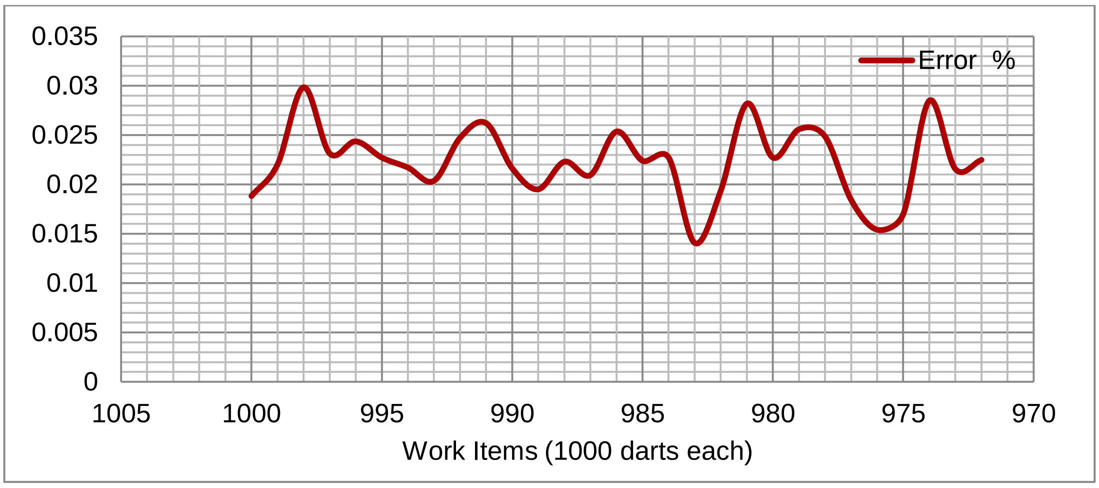


Fig 5-2: percentage error variation (a), (b), and (c)

Percentage error thus obtained was plotted to understand the behaviour of the curve. It is evident that the error percentage increases with the decrease in the number of darts from Fig 5-2(a) and Fig 5-2(b), and it is not seen so clearly in Fig 5-2(c). Work items here is nothing but a tiny chunk of the total work to be done i.e., if generating 100,000 random numbers is our target, it is divided into 100 work items with 1000 darts in each, making it easy to be distributed to all the worker processes (see section 4.3).

In Fig 5-2(a), the number of work items (W) was set to 100 and it was decreased by the decrease interval (I) of 10 every time and the trials were taken. i.e., for the first 20 trials, the work items were 100. For the next 20 trials it was reduced to 90, and so on, till only 10 remained. The ratio of decrease interval to the total number of darts is

$$\frac{I}{W} = \frac{10}{100} = 10\%$$

Since the ration is very high (10%) the range of the variation is very wide. The values vary from as high as 0.7% to as low as 0.09%.

In the Fig 5-2(b), the total number of work items is set to 1000 and the decreasing interval was still set to 10. The ‘I/W’ ratio for this case is 1%. Though the variation of the mean is now confined to the range 0.015 to 0.035. In Fig 5-2(c) however the value of I was decreased to 1 for the W of 1000 hence the ratio ‘I/W’ is now 0.1%. The variation in the error percentage is drastic over the range 0.015 to 0.03, but the range is still lesser than what it was for I/W = 1%.

Thus, we can deduce three important results from this experiment –

1. The approximation given by the Monte Carlo integration gets closer to the actual answer as the number of darts increases
2. As the ‘I/W’ ratio decreases, the range in which the mean of error percentage varies also grows smaller
3. For the ‘I/W’ less than 0.1%, the range of variation almost remains constant.

This project exploits these results, the third one in particular to achieve fault tolerance.

5.3 Fault Tolerance Test

There two aspects to the fault tolerance achieved by the project. One is the program fault tolerance, which deals with the fault tolerance of the project at the API and application level. Here the project is tested for error immunity. The second is the algorithmic fault tolerance, where the project is tested for the effective fault tolerance provided by the algorithm. The following sections present details regarding the both.

5.3.1 Program Fault Tolerance Test

The fault tolerant test was conducted by inducing faults at application level (API level). Errors were simulated by giving wrong arguments in the API. The processor in which the wrong argument was given would instantly know the issue and hence retire to the error condition. The processor which waits on or send the message will know it through the timeout. This is exactly as discussed in the section 4.4.5. Table 5-2 gives details about the four cases tested and actions taken by server and worker for fault tolerance in each of the scenarios.

Error		Action	
Server	**Worker**	**Server**	**Worker**
Send – ok	Receive – err	After send, Server waits in receive mode for the communication from the worker for 3 heart beats. If not received, intercommunicator is marked as dead and there is no further communication with that intercom.	Worker instantly knows that receive operation has failed. Retires to Error condition and finalizes.
Send – err	Receive – ok	Server instantly knows that send operation has failed. The intercommunicator is marked as dead and there is no further communication with that intercom	Worker waits in receive mode till the receive timeout occurs. If there is no action from the server, it retires to error condition and finalizes.
Receive – ok	Send – err	Server waits in receive mode for 3 heart beats. If not received, intercommunicator is marked as dead and there is no further communication with that intercom	Worker instantly knows that send operation has failed. Retires to Error condition and finalizes.
Receive – err	Send – ok	Server instantly knows that receive operation has failed. The intercommunicator is marked as dead and there is no further communication with that intercom	After send, Worker waits in receive mode for the next communication till the receive timeout occurs. If there is no action from the server, it retires to error condition and finalizes.

Table 5-2: Faults check in MPI program

5.3.2 Algorithm Fault Tolerance Test

Consider a world of 1 server and 31 workers. Workers would be working on one work item at any given time during the run of the program. Work item is a small fraction of the entire work, with 1000 darts. Upon the death or crash of a worker, the work item it was having at the time of death is lost. The server stops communicating with that worker and updates the world to be of 30 workers. From that moment on, rest of the work items are divided between the remaining 30 workers alive. Measure of the accuracy loss due to the loss of the work item and the increase in the computation time is done in the following section.

Monte Carlo Numerical integration yields an approximate result every time a trial is taken. With the loss of a work item it is impossible to exactly say how much of accuracy is compromised. To overcome this problem, 20 trials each was taken, by crashing one processor at a time till only 1 server and 1 worker remained in the world. The mean values were tabulated and mean deviation and variance was calculated for the range. This was compared with the mean deviation and variance of the 20 trials taken with 1000 work items with 1000darts in each, without any processor crash.

Mean is the average value of all the values obtained in the trial and standard deviation is the intervals next to the mean value around with 68% of the values are scattered. It is as shown in the Fig 5-3. The μ is the mean value, σ is the deviation and 68% of the values are scattered between $(\mu - \sigma)$ and $(\mu + \sigma)$. Note that mean is just the average of the values obtained and does not necessarily be the actual value. For instance, the mean of 20 trials taken can be 3.145 with a standard deviance of 0.05. The actual value of π, 3.1415 and lies in the region of μ and $(\mu - \sigma)$. The calculation of the standard deviation and variance was discussed in the section 3.3.5. 'Error Analysis of Monte Carlo Numerical Integration'.

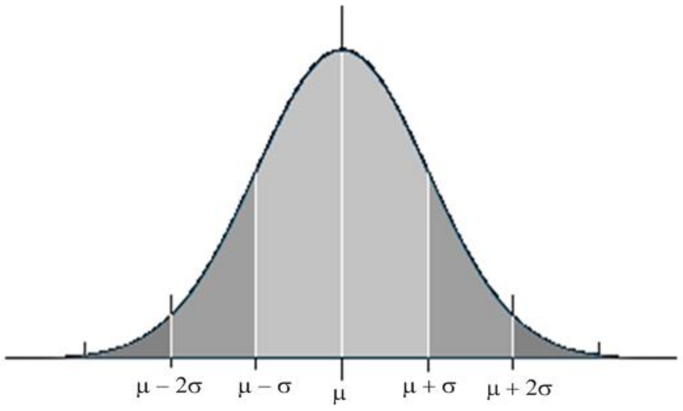
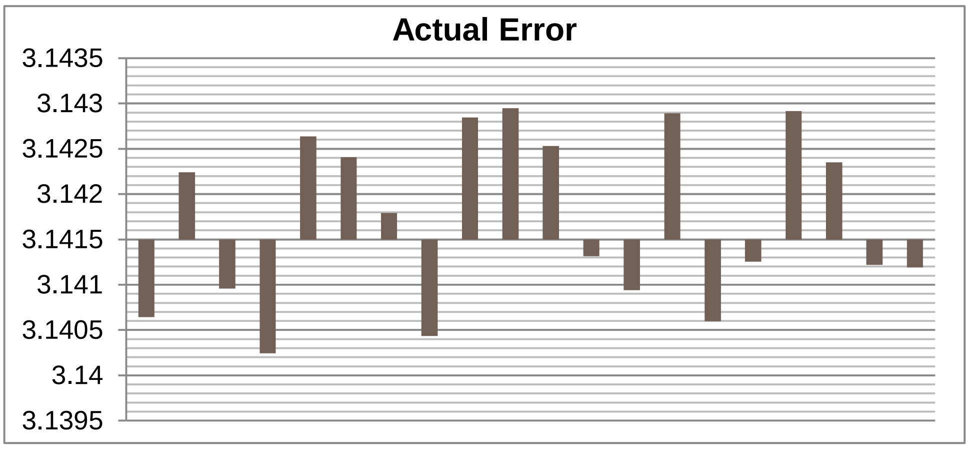


Fig 5-3: mean and standard deviation

For the 20 trials taken with 1000 work items the actual error varied between 0.001354278 and 0.000199334. The Fig 5-4(a) shows the actual error plotted on the base line π and Fig 5-4(b) shows the corresponding error percentage.



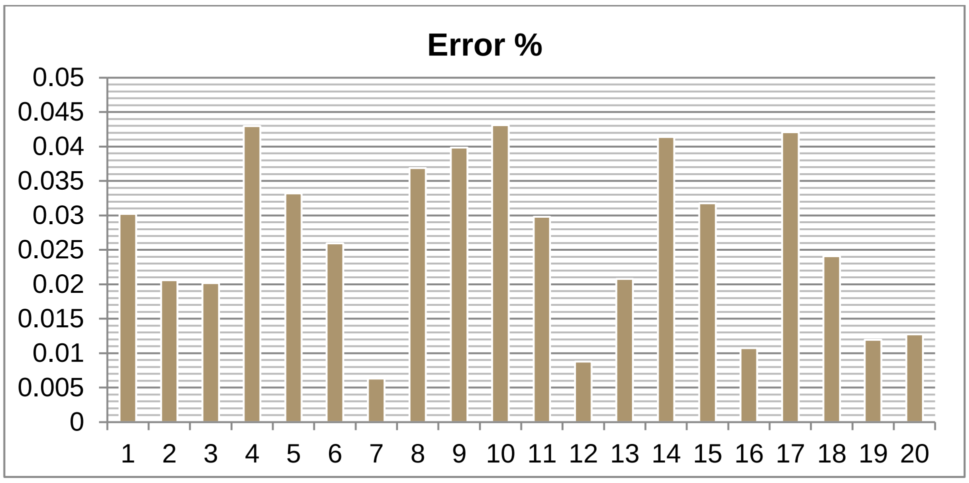


Fig 5-4: (a) Actual Error and (b)% error for 1000WI in 20 trials

The calculations yielded the following-

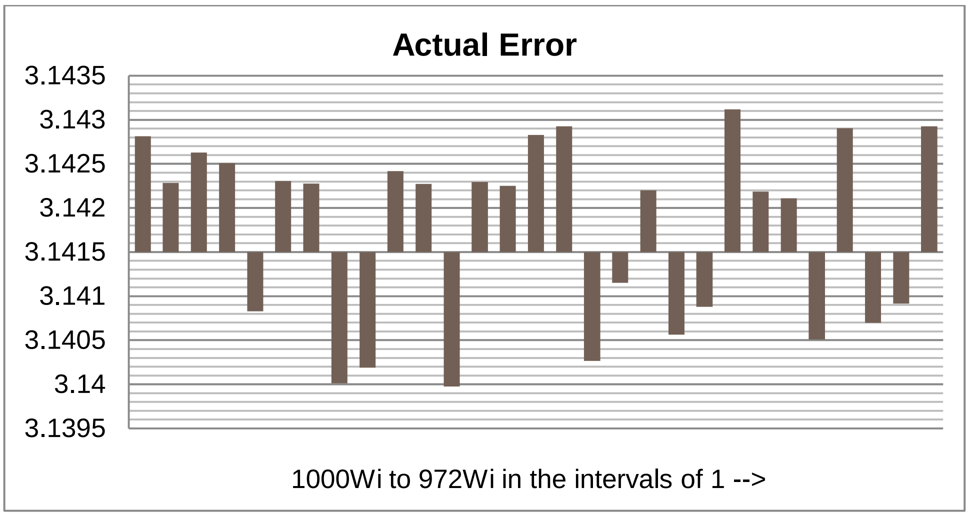
Mean μ = 3.141717786
Standard deviation σ =0.000910088
Variance σ^2 = 8.28259E-07

--- ***Case 1***

For the second set of data obtained (Fig 5-5 a and b) when the processors crashed, the calculation was carried out in the similar manner to obtain –

Mean μ = 3.141766584
Standard deviation σ = 0.001013869
Variance σ^2 = 1.02793E-06

--- ***Case 2***



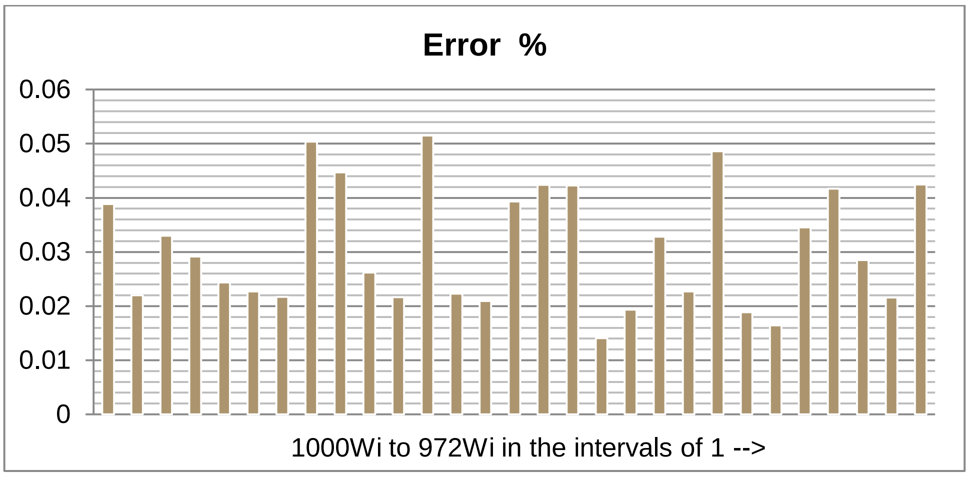


Fig 5-5: (a) Actual Error and (b)% error for decreasing Wi due to process crash

The mean value obtained for the first case is closer to the actual π value, with a difference of 0.000124786. The standard deviation is also lower, suggesting that the values are not scattered closer to the mean. For the second case, the difference between π and obtained mean is about 0.000173584. The accuracy loss due to the processor crash therefor is around $4.87978 \text{x} 10^{-5}$, which when stated in percentage is $1.55 \text{x} 10^{-3}$%. This is an extremely small value which could be neglected. Besides, in the actual scenarios, the supercomputers work with millions of such work items and therefore the accuracy loss will be a very tiny number.

No. of proc crash	Computation time
0	66.244485
4	74.263128
8	78.274515
16	80.273201
20	84.276159
24	88.305602
28	100.351111
30	132.384400

Table 5-3: Worst case computation time variation with the processor crash

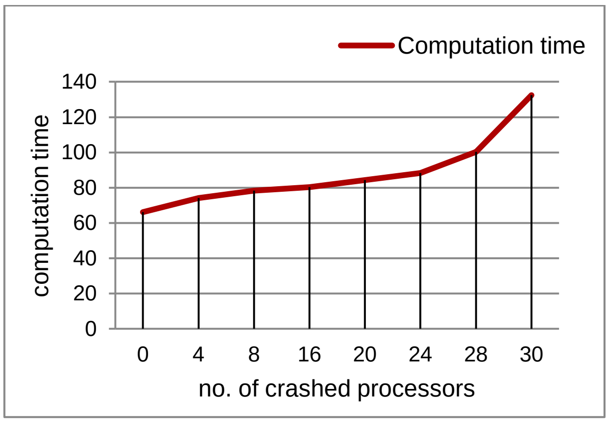


Fig 5-6: Worst case computation time variation with the processor crash

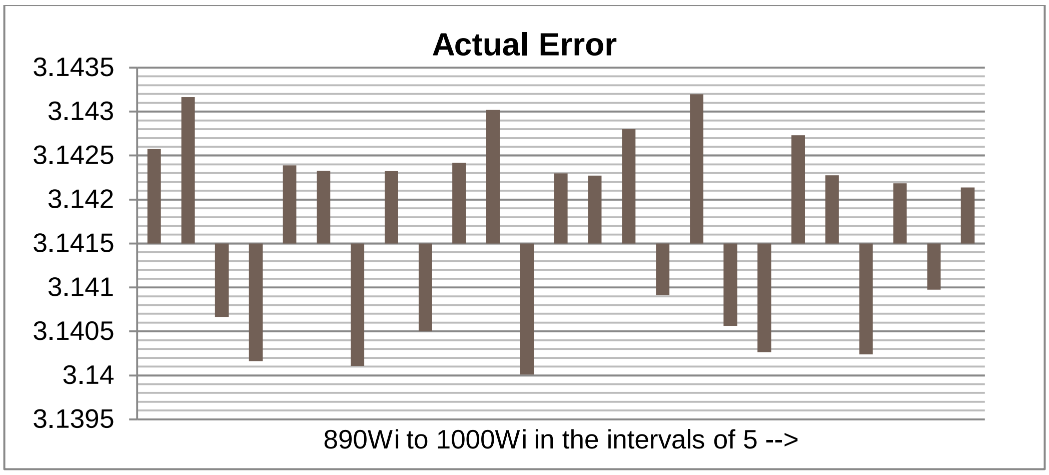
The computation time increases as the processors crash because the work that has to be done by n processors has to be now done by n-1 processors from that point on. Table 5-3 gives the worst case computation time taken by the system, i.e., the processors crash even before the computations begin. Hence the crashing processor would not have performed even a fraction of the total work and the burden falls entirely on the rest of the living processors. This is the overhead, the price to be paid for the fault tolerance. The unit of time here is seconds. The computation time with no loss is so high because of the deliberate wait that has been inserted in the prototype program. This will have to be replaced with a code to perform other computations by the processor in the real world program.

5.4 Test for Other Fault Tolerant Features

There are few features which are added in the project for extra fault tolerance and for better accuracy. The important thing here is to check if those are really of any worth as they demand other overheads, like computation time. Two such features are, sending 1 work item at a time and not re allocating the lost work item. Both are discussed in the following sections in detail.

5.4.1 Work Item per Update trade-off

Only one work item is sent to the worker at a time per send-recv communication pair. This was decided during the design phase to make sure that if there is a processor crash, not a lot of data is lost. However this increased the computation time as a lot of communications had to be carried out.



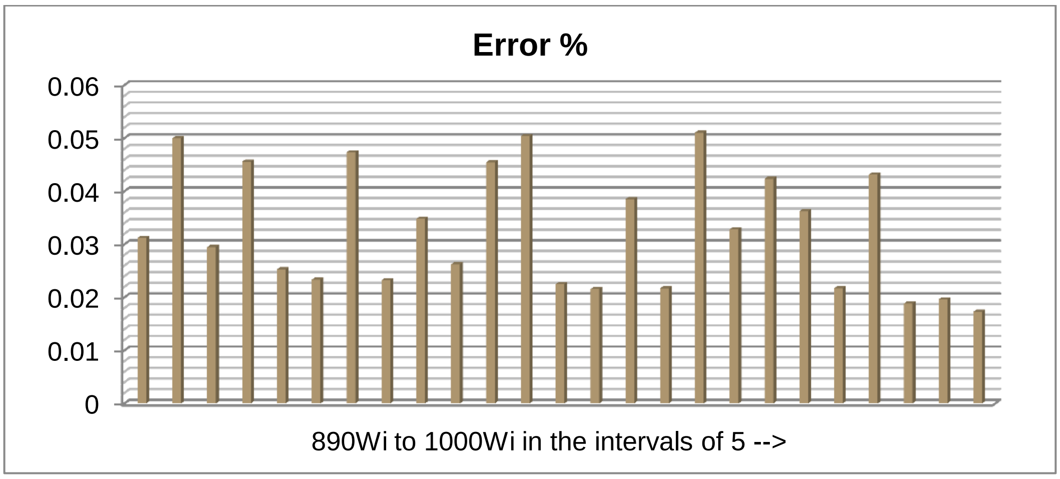


Fig 5-7: (a) Actual Error and (b)% error for decreasing Wi in 5x

The computation time can be reduced by decreasing the number of interactions between the server and worker. This could be done by sending 5 work items to a worker at a time and receiving the result of those 5 work items in one set of send-recv communication. This reduces the computation time by one fifth, which is pretty good. But for this the accuracy has to be compromised a little. To find out how much of accuracy loss would occur for this case, the readings were taken just like it was done in --- Case 2 discussed in the previous section. Effective mean, standard deviation and variance were calculated. The result obtained was-

Mean μ = 3.141700015
Standard deviation σ = 0.001082298
Variance σ^2 = 1.17137E-06

--- ***Case 3***

Compared to the Mean value of μ = 3.141766584, obtained in the --- Case 2, it can be seen that the accuracy compromise isn't very huge. But as the number increases from 5 to a bigger number, the accuracy compromise also gets bigger and bigger. However if the total number of work items increases from 1000, then the number of work items per communication can also be increased without much compromise in the accuracy.

5.4.2 Re-allocation of lost Work Item trade-off

When a processor crashes, the work item lost is technically not lost, as all the information needed for the re-allocation of that work is present with the server. As soon as the server learns about the crash of a processor, the server can re-allocate the work item which was associated with the crashed processor at the time of crash to one of the living processors. This would mean no compromise on the accuracy but the overhead would be on the computation time.

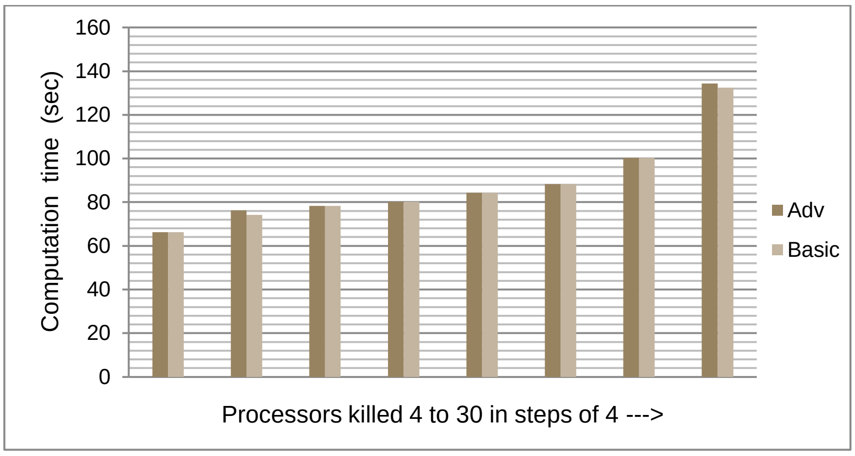


Fig 5-8: Computation times: re-allocation (Adv) and no re-allocation (Basic)

No. of proc crash	Adv	Basic
0	66.24449	66.24449
4	76.27878	74.26313
8	78.26742	78.27452
16	80.2663	80.2732
20	84.28061	84.27616
24	88.28137	88.3056
28	100.3078	100.3511
30	134.3743	132.3844

Table 5-4: Computation times: re-allocation (Adv), no re-allocation (Basic)

It is evident that as the number of crashes increases, the time difference between the Adv and Basic increases. Hence the reallocation is an option if the system is not so prone to processor crashes. Else it is not a good idea to compromise 2 seconds of computation time for an increase in the accuracy which as low as $1.55\text{x}10^{-3}\%$.

Chapter 6. Critical Evaluation

The main aim of the project was to develop a fault tolerant algorithm for highly parallel computers. The objective was to develop a prototype program using C language and MPI on the university supercomputer BlueCrystal.

Though in the beginning FT-MPI was favoured for the implementing the project, due to all reasons stated in section 4.1.1, it was given up. After a series of tests and comparisons, OpenMPI was chosen for the project (section 4.1.2). The library was already installed in BlueCrystal Phase 1 and hence there was no need for configuration. This completes the step 1 of the objectives set.

The project was designed after a lot of brain storming to meet all the objectives which were set during the initial days. To achieve the fault tolerance in the program, it had to be made immune to all the factors which caused a typical parallel MPI program to abort. At application level, two reasons for the same were identified–

1. Fail-Stop process failure i.e., processor crash/death in the communicator
2. Error in the MPI calls – wrong argument, timeout etc.

The second issue could be handled in the program but to address the first issue, support was needed from the API layer. The MPI implementation had to return the control back to the application layer (the program) so that the programmer could handle the fault. However, none of the implementations as of now supports this feature, except for FT-MPI. Though FT-MPI was an implementation of MPI specification 1.2 (the latest is ver. 2), it was decided go ahead with it for the message passing. A considerable amount of time was spent on understanding the documentation and searching for help to install it on the BlueCrystal. However due to a bunch of unresolved installation issues, the idea of using FT-MPI for the project was given up. As the other implementations abort immediately when the processor crash is detected, developing a project which can withstand processor failures was impossible. Hence, an assumption that, "failing of few of the processes in the message passing system does not cause the aborting of rest of the processes" was made, and process crash was simulated for the project. The basis of this assumption were-

- FT-MPI claims to support such a feature already
- MPI ver3 which is due to be released has this as its most important feature. There is a group which is already working to realise this

To deal with the processor crash, a server-worker programming model was designed using intercommunicators. Since only server talks to the workers through intercommunicators, death of a worker (processor crash) does not harm the working of either the server or other workers. The server always has a copy of work assigned to each and every worker. Hence crash of a processor does not harm the computation, as it could be reassigned. The use of intercommunicators successfully blocks the crashed processor and the programming model assures that no data is lost. Though this is based on the assumptions made, it can be adopted to the programs in FT-MPI or MPI ver3 implementation (the next version of MPI specification is MPI-3, which is yet to be released, claims to have process fault tolerance). This would make the program immune to processor crashes.

To deal with the second category of errors, a pair of blocking Send and non-blocking Receive was used with implicit acknowledgement. Receive timeout was used to know if the message was received at all. Due to the simplicity of the chosen domain of integration, (unit circle inscribed in a unit square), the wait cycles had to be inserted in the program. If the domain of integration were to be complicated or multidimensional, the wait cycles could be replaced with other calculations. This is saved for the future work in the project. However, with this programing technique, the program was made immune to the second category of errors. Thus step 2 of the objectives was successfully completed.

To bring in fault tolerance in algorithmic level was step 3 in the list of objectives. Monte Carlo, one among the seven dwarfs of parallel computing algorithms [**24**], which is widely applied for solving problems in finance, astrophysics, nuclear physics, etc., was considered. For the prototype, numerical integration was chosen, and for clarity and simplicity, calculating π (whose value is known) was used. To get the right result using the algorithm, there was a high dependency on the random number generator.

A parallel random number generator was needed, which was highly independent and non-correlated. The numbers produced by each stream had to be unique, not produced by any other stream at any point of the run to make the work division successful. Another requirement was, for better approximation of the value by a Monte Carlo Numerical Integration, the numbers generated had to have less interval space, i.e., the random numbers had to be concentrated around the area of interest and not equally distributed. A Gaussian RNG or an exponential RNG which would produce highly independent data streams in parallel was required for the project. Programming for an RNG would take a lot of time and effort. Besides, it was out of the scope of the project. Hence all the available options were considered and compared to decide which suited the best and yielded high quality independent random number streams (section 4.4.2). Finally, sprng, a freely downloadable library for parallel random number generation was used for the project (section 4.4.3). The RNG algorithm used was Multiplicative Lagged Fibonacci Generator. Though not ideal for this application, it did really well. The results obtained were reasonably good as the streams were fairly independent. The results would have been much better i.e., higher accuracy could have been achieved with lesser number of random numbers, if the parallel RNG had used Gaussian or exponential algorithm. However this was out of scope to be developed during the course.

Step 4 of the objectives was to check for the application response to processor crash/fail-stop process failure, compile the results and analyse them to draw a logical conclusion. The final project prototype developed was tested thoroughly (Chapter 5) and was found to handle process crash (taking into account the assumption made), error at the application and API levels successfully. The accuracy compromise for achieving this was measured and it was found to be as low as 4.87978×10^{-5} or 1.55×10^{-3}%. The program took longer and longer than usual time to compute the result as more and more number of processors crashed but it wouldn't abort until all the work is completed. Though this is not desirable, it is preferred because a programmer would rather let the computation take longer and complete, than let it abort and start all over again at the occurrence of every error.

Thus all the objectives set out for the project were met successfully. The work done is detailed in Chapter 4. The results are tabulated and analysed in Chapter 5.

Chapter 7. Future

The project was just a prototype to what can be done for fault tolerance in MPI programming in the application end by a programmer. This has a lot of scope for further exploration. With high performance computing becoming the buzz word of the near future, this area of research will get more prominence than what it already has today. This chapter gives some possible improvements for the project and samples of what could be done in future on the foundation of this project.

7.1 Possible Improvements

The first thing which strikes is the use of MPI2 features like 're-spawn process' to make the program more fault-tolerant. If there is a redundant processor maintained to keep track of the server proceedings, upon crash of the server, it could be replaced or re-spawned. This redundant processor could be a dedicated processor which performs no other computation if the MPI_COMM_WORLD has s lot of nodes. It could also be one of the worker processors if less number of nodes is involved in the computation.

If the installation of FT-MPI is successful anytime in future, the program could be ported to FT-MPI from OpenMPI which would make it sustain Fail-Stop process failures. Error Handlers have to be modified accordingly. BLANK failure mode could be used on a worker death and REBUILD could be used in case the server fails (refer section 3.2.5). The simulation of the process crash will not be necessary at that time. Instead programmer can ssh into one of the nodes and kill the process using the following script in BlueCrystal-

```
qstat JOB -an1
#grabbing the nodename
node=`qstat -an1 1587392 |grep master|
      awk '{print $12}'|awk -F + '{print $1}'`
ssh $node
#getting the pids
ps -ef |grep P-Gadget | awk '{print $2}'|sort
#chose pid to kill
#kill pid
```

The random number generator is another important area for improvement. A Gaussian or an exponential random number generator is ideal for Monte Carlo numerical integration. The sprng library which was used in the project does not have any of these. And other freely downloadable libraries which provided Gaussian RNG did not support parallelisation. The solution was to develop a parallel scalable Gaussian RNG. This could not be done in the course of the project as it demanded a lot of effort. Besides, it was not related to making the program fault tolerant, it was required only to increase the accuracy of the result obtained. Hence it can be taken up as an add-on or an improvement factor to make the prototype more efficient.

7.2 Future Work

Once an implementation of MPI ver3 is released, the project can be made immune to Fail-Stop process failures, just as suggested in the previous section for FT-MPI. Depending on the features and support the MPIs provide, the prototype could be remodelled to meet the requirements.

Real world problems in finance and nuclear physics could be solved using the program so that it has enough computations to fill the wait times in the prototype. The results thus obtained can be used to analyse the performance of the project in a real world scenario. This would increase the credibility of the project.

Most important future work would be to modify the prototype code to use it as an abstraction layer over the MPI and expose few APIs to the user. This would create a platform for the programmer who wants a fault tolerant Monte Carlo application.

Bibliography

[1] *website: "http://www.top500.org/lists/2010/11"*.

[2] MPI 2 Forum:, "MPI: A Message-Passing Interface Standard. Document for a Standard Message-Passing Interface," University of Tennessee, 1995.

[3] James S. Plank and Kai Li and Michael A. Puening, "Diskless Checkpointing," in *Plank97*, 1997.

[4] Charng-Da Lu, "Scalable Diskless Checkpointing For Large Parallel Systems," University of Illinois, Urbana-Champaign, Dissertation 2002.

[5] Graham E. Fagg et al., "Fault Tolerant Communication Library and Applications for High Performance Computing," *Los Alamos Computer Science Institute Symposium*, pp. 27 - 29, 2003.

[6] Rob T. Aulwes et al., "Network fault tolerance in LA-MPI," *volume 2840 of Lecture Notes in Computer Science (LNCS)*, pp. 344 - 351, September-October 2003.

[7] Al Geist and Christian Engelmann, "Development of naturally fault tolerant algorithms for computing on 100,000 processors," 2002.

[8] Mootaz Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," June 1999.

[9] Soulla Louca, Neophytos Neophytou, Arianos Lachanas, and Paraskevas Evrepidou, "MPI-FT: Portable fault tolerance scheme for MPI," *Parallel Processing Letters 10(4)*, p. 371–382, 2000.

[10] R. Gupta et al., "CIFTS: A Coordinated Infrastructure for Fault-Tolerant Systems," *ICPP 2009 IEEE*, 2009.

[11] "website: "http://www.mcs.anl.gov/research/cifts/index.php"".

[12] Elmootazbellah N. Elnozahy and James S. Plank, "Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery," *IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, Vol 1 No. 2*, pp. 97 - 108, April-June 2004.

[13] John H. Halton, *A Retrospective and Prospective Survey of the Monte Carlo Method.*: Society for Industrial and Applied Mathematics, Jan., 1970.

[14] Paul Coddington, "Distributed and High-performance computing Case study - monte carlo Simulation of the ising model," University of Adelaide, July - October 2000.

[15] Yahya M. Masalmah and Yu (Cathy) Jiao, "Scalable Parallel Algorithms for High Dimensional Numerical Integration," Oak Ridge, Oct 2010.

[16] Jeffrey S Rosenthal, "Parallel computing and Monte Carlo algorithms," *Far East Joural of Theoretical Statistics*, pp. 207 - 236, 2000.

[17] Jan Tobochnik, and Wolfgang Christian Harvey Gould, *Introduction to Computer Simulation Methods: Applications to Physical Systems*, third edition ed.: Addison-Wesley, 2006, Chapter 11- Numerical and

Monte Carlo Methods.

[18] Henry Neeman, "Parallel & Cluster Computing, Monte Carlo ," *OU Supercomputing for Education and Reasearch*, 2008.

[19] S., Sahin, C., Thandavan, A. Branford, Weihrauch C., Alexandrov V. N., and I. T. Dimov, "Monte Carlo methods for matrix computations on the grid," *Future Generation Computer Systems-the International Journal of Grid Computing Theory Methods and Applications, 24 (6).*, pp. 605-612, 2008.

[20] V.N. Alexandrov and Z. Zlatev, "Using Parallel Monte Carlo Methods in Large-Scale Air Pollution Modelling," in *International Conference on Computational Science*, 2004, pp. 491-498.

[21] E.I. Atanassov V.N. Alexandrov, I. Dimov, S. Branford, A. Thandavan, and C. Weihrauch, "Parallel Hybrid Monte Carlo Algorithms for Matrix Computations," in *International Conference on Computational Science (3)*, 2005, pp. 752-759.

[22] "website: "http://sprng.cs.fsu.edu/",".

[23] website, "https://www.acrc.bris.ac.uk/phase1_user_guide/user_guide.htm,".

[24] website, ""http://mvapich.cse.ohio-state.edu",".

[25] William Gropp and Ewing Lusk, "Fault Tolerance in MPI Programs," *International Journal of High Performance Computing Applications, vol. 18, no. 3,* pp. 363-372, 2002.

[26] MPI message-passing interface standard, "The MPI message-passing interface standard," p. Section 7.2, May 1995, http://www.mpi-forum.org.

[27] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra, *MPI—The Complete Reference Volume 1, The MPI Core, 2nd edition*. Cambridge, MA: MIT Press, 1998.

[28] *website: "https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/FaultToleranceWikiPage".*

[29] website, ""http://sprng.cs.fsu.edu/"".

[30] Krste Asanovic et al., "The Landscape of Parallel Computing Research: A View from Berkeley," *Technical Report No. UCB/EECS-2006-183*, December 18 2006.

[31] Charng-da Lu and D.A. Reed, ""Assessing Fault Sensitivity in MPI Applications," Supercomputing," in *Proceedings of the ACM/IEEE SC2004 Conference*, vol. vol., no., pp. 37, 2004.

Appendices

Appendix - A. FatoAl.h

```
#ifndef fatoal_H_
#define fatoal_H_

#undef BASIC //define if the lost work item need not be re allocated

/*------------ Macros --------------*/

#define PI_ACCURATE           3.1415926535897932384626433
#define MAX_PROC_NUM          32// 31workers + 1server
#define MAX_INTERCOMM_NUM     MAX_PROC_NUM-1
#define SEED                  35791270
#define GEN_TYPE              4

#define DARTS_PER_WORK_ITEM   1000
#define NUMBER_OF_WORK_ITEM   1000
#define TOTAL_DARTS           DARTS_PER_WORK_ITEM*NUMBER_OF_WORK_ITEM

//tags
#define TAG_WORK_ID           501
#define TAG_RESULT            502
#define TAG_IC_CREATE         503

//time out values
#define W_IRECV_TIMEOUT       100
#define RETRY_COUNT           10
#define LISTEN_TIMEOUT        16
#define S_IRECV_TIMEOUT       3

//work id
#define FINALISE      65536 //a very big number
#define NULLWORKID    -1

/*------  Globals ------*/
int gnWorldSize;

/*------  functions  -------*/

static void server(void);
static void worker(void);

//server end routines
void UpdateWorkerWorkState(int, int);
void InitialiseWorkers(void);
void InitialiseWorkItems(void);
int GetNextWorkItem(void);
void ResultLog(int, int);
void CheckForTimeOut(int nIntercommId);
void PrintFinalResult(void);

//worker end routines - none

/*------ enums ------*/

typedef enum{
        INITS =1,
        ISSUE,
        COLLECT,
        FINISH
}eSrever_State;

typedef enum {
        INIT =1,
        READY,
        RECEIVED,
        RUNNING,
        COMPLETED,
        ERROR
}eWorker_State;
```

```
typedef enum{
        IC_ALIVE=1,
        IC_DEAD
}eIC_State;

typedef enum{
        WORK_NOT_DONE=1,
        WORK_IN_PROGRESS,
        WORK_LOST,
        WORK_DONE
}eWork_State;

/*--------  structures  ---------*/
typedef struct
{
        unsigned int nIntercommId;
        eWorker_State eWorkerState;
        int nCurrentWorkId;
        int nRecvFailureCount;
        eIC_State eIntercommState;
        int nNumWorkItems;
}tWorker;

typedef struct
{
        int nWorkId;
        int nIntercommId;
        eWork_State eWorkState;
}tWorkItem;

#endif /*fatoal_H_*/
```

Appendix - B.   FatoAl.c - Server code

```
/*---------------------------------------------------------------------------------------------
function name  : server
functionality  : server part of the server-worker program
called by      : main, if the processor rank is 0
function calls : InitialiseWorkers()
                 InitialiseWorkItems()
                 UpdateWorkerWorkState()
                 ResultLog()
                 CheckForTimeOut()
                 PrintFinalResult()
--------------------------------------------------------------------------------------------*/
void sever()
{

    MPI_Comm worker_comm[MAX_PROC_NUM];
    MPI_Status status;
    MPI_Request req[MAX_INTERCOMM_NUM];
    eSrever_State serverState;

        int i, j, count, over=0;
        int retVal=0, intercomError=0, nMyCommRank;
        int nNextWorkId, nResultMsg[MAX_INTERCOMM_NUM], flag[MAX_INTERCOMM_NUM]={0};
        double dStartTime, dEndTime, dTimeElapsed;

    dStartTime = MPI_Wtime();
    gnCurrentSize = gnWorldSize;
    printf("Server - current world size: %d\n\n", gnCurrentSize);
    serverState = INIT;

    while(!over)
    {
        switch (serverState)
```

```
{
 case INITS:

        //initialise workers and work items
        InitialiseWorkers();
        InitialiseWorkItems();

        //create intercommunicators and set error handlers
        //printf("server in INITS\n");

        for ( i = 0; i < gnWorldSize-1; i++ )
        {
                intercomError = MPI_Intercomm_create( MPI_COMM_SELF, 0, MPI_COMM_WORLD,
                        i+1, TAG_IC_CREATE, &worker_comm[i]);
                if(intercomError)
                {
                        printf("Server - intercom Error = %d\n",intercomError);
                }
                else
                {
                        MPI_Comm_rank( worker_comm[i], &nMyCommRank );
                        printf("server - my comm rank in communicator '%d' is '%d'\n",
                        i, nMyCommRank);
                }
                MPI_Comm_set_errhandler( worker_comm[i], MPI_ERRORS_RETURN );
        }
        serverState = ISSUE;
        break;
 case ISSUE:
        //Dispatch work to all the workers available
        printf("server in ISSUE\n");
        for ( i = 0; i < gnWorldSize-1; i++ )
        {
                if(atWorker[i].eWorkerState == READY)
                {
                        nNextWorkId = GetNextWorkItem();
                        //printf("Server - next work %d\n",nNextWorkId);
                        /*if(i == 2) //to generate error
                                retVal = MPI_Send(&nNextWorkId, 1, MPI_INT,
                                0, 11, worker_comm[i]);
                        else*/
                                retVal = MPI_Send(&nNextWorkId, 1, MPI_INT,
                                0, TAG_WORK_ID, worker_comm[i]);
                        if (MPI_SUCCESS == retVal)
                        {
                                if(nNextWorkId == FINALISE)
                                {
                                        printf("Server - sent FINALISE to ic %d\n",i);
                                        UpdateWorkerWorkState(i, FINALISE);
                                }
                                else
                                {
                                        printf("Server - sent work %d to ic
                                         %d\n",nNextWorkId, i);
                                        UpdateWorkerWorkState(i, nNextWorkId);
                                }
                        }
                        else
                        {
                                printf("Server - session error to comm: %d,
                                retval: %d\n", i, retVal);
                                UpdateWorkerWorkState(i, NULLWORKID);
                        }
                }
                else
```

```
                        {
                                //printf("Server - worker %d is in the state %d\n",
                                 i, atWorker[i].eWorkerState);
                        }
                }// end for
                printf("server - work pending: %d, workers alive: %d\n",
                        gnWorkNotDone, gnWorkersAlive);
                if((gnWorkNotDone ==0)&&(gnWorkersAlive ==0))
                {
                        serverState = FINISH;
                        break;
                }
                serverState = COLLECT;
                break;
        case COLLECT:
                //collect results
                //printf("server in COLLECT\n");
                sleep(1);
                for ( i = 0; i < gnWorldSize-1; i++ )
                {
                        if(atWorker[i].eWorkerState == RUNNING)
                        {
                                retVal = MPI_Irecv(&nResultMsg[i],1,MPI_INT,
                                0,TAG_RESULT,worker_comm[i],&req[i]);
                        }
                }
                sleep(1);
                for ( i = 0; i < gnWorldSize-1; i++ )
                {
                        if(atWorker[i].eWorkerState == RUNNING)
                        {
                                MPI_Test(&req[i], &flag[i], &status);
                                if(flag[i])
                                {
                                        printf("Server - ic %d sent result = %d\n",
                                        i, nResultMsg[i]);
                                        ResultLog(i, nResultMsg[i]);
                                        UpdateWorkerWorkState(i, NULLWORKID);
                                        gnWorkDone++;
                                }
                                else
                                {
                                        //check for time out
                                        CheckForTimeOut(i);
                                }
                        }
                }
                serverState = ISSUE;
                break;
        case FINISH:
                printf("server in FINISH\n");
                //free intercoms
                for ( i = 1; i < gnCurrentSize; i++ )
                {
                        MPI_Comm_free( &worker_comm[i-1] );
                }
                //print the time elapsed
                dEndTime = MPI_Wtime();
                dTimeElapsed = (dEndTime - dStartTime);
                printf("time elapsed (in sec): %lf\n", dTimeElapsed);

                //print result
                PrintFinalResult();
                over = 1;
                break;
        }//end of switch
    }//end of while(!over)
}
```

Appendix - C.   FatoAl.c - Worker Code

```
/*----------------------------------------------------------------------------------------------
function name  : worker
functionality  : worker part of the server-worker program
called by      : main, if the processor rank is not 0
function calls : init_sprng()
                 sprng()
---------------------------------------------------------------------------------------------*/
oid worker()
{

        int nMyRank, nMyCommRank;
        int retVal, intercomError;
        int done =0, flag =0;
        int nWaitCount=0, i;
        int nWorkId,  nWorkResult=0;

        MPI_Request req;
        MPI_Status status;
        MPI_Comm server_comm;
        eWorker_State workerState = INIT;

        // for random number
        int nLowLim= -1, nHighLim =1, nCircle=0;
        double x, y, z, pi_current;
        int *stream_id;

    while(!done)
    {
        switch(workerState)
        {
                case INIT:
                        MPI_Comm_rank( MPI_COMM_WORLD, &nMyRank );

                        //printf("worker: %d start\n", nMyRank-1);

                        /* create intercommunicators for communication with manager only */
                         intercomError = MPI_Intercomm_create( MPI_COMM_SELF, 0,
                                MPI_COMM_WORLD, 0, TAG_IC_CREATE, &server_comm );
                        if(intercomError)
                        {
                                printf("worker: %d, intercom Error = %d\n",
                                nMyRank-1, intercomError);
                        }

                        MPI_Comm_set_errhandler( server_comm, MPI_ERRORS_RETURN );

                        //initialise sprng
                        stream_id = init_sprng(GEN_TYPE,nMyRank,
                                gnWorldSize,SEED,SPRNG_DEFAULT);

                        workerState = READY;
                        break;
                case READY:
                        retVal = MPI_Irecv(&nWorkId, 1, MPI_INT, 0,
                                 TAG_WORK_ID, server_comm, &req);
                        while(1)
                        {
                                sleep(1);
                                MPI_Test(&req, &flag, &status);
                                if(flag==1)
                                {
                                        workerState = RECEIVED;
                                        nWaitCount=0;
                                        break;
                                }
                                else if(nWaitCount == W_IRECV_TIMEOUT)
```

```
                    {
                            //printf("worker: %d, receive timeout \n", nMyRank-1);
                            workerState = ERROR;
                            break;
                    }
                    nWaitCount++;
            }
            break;
    case RECEIVED:
            if(nWorkId == FINALISE)
            {
                    //printf("worker: %d, received msg to finalise\n",nMyRank-1);
                    workerState = COMPLETED;
            }
            else
            {
                    //printf("worker: %d, received work- %d\n", nMyRank-1, nWorkId);
                    workerState = RUNNING;
            }
            break;
    case RUNNING:
            printf("worker: %d, starting the work- %d \n",nMyRank-1, nWorkId);
            nCircle=0;
            for (i = 1; i <= DARTS_PER_WORK_ITEM; i++)
            {
                    // get random numbers between 0, +1
                    x = (double)sprng(stream_id);
                    y = (double)sprng(stream_id);

                    // z = x^2 + y^2
                    z = (x * x) + (y * y);

                    // if it is inside the curve, increment nCircle
                    if (z <= 1.0)
                            nCircle++;
            }
            nWorkResult = nCircle;

           retVal = MPI_Send(&nWorkResult, 1, MPI_INT,
                      0, TAG_RESULT , server_comm);

            if(retVal)
            {
                   printf("worker: %d, MPI_Send error - retval = %d\n",
                            nMyRank-1, retVal);
                    workerState = ERROR;
                    break;
            }

            printf("worker: %d, sent result %d\n", nMyRank-1,nWorkResult);
            workerState = READY;
            break;
    case COMPLETED:
            MPI_Comm_free( &server_comm );
            done = 1;
            break;
    case ERROR:
            // when send or receive fails, mark intercommunicator as dead & exit
            printf("worker: %d, error case.. free the inter-comm\n", nMyRank-1);
            MPI_Comm_free( &server_comm );
            done = 1;
            break;
    default:
            printf("Worker: %d, unknown case\n",nMyRank-1);
            break;
            
        }
    }
}
```