

Using Message Passing Interface(MPI) to Parallelise a Simple Genetic Algorithm(SGA) Running on Beowulf Cluster



UNIVERSITY of LIMERICK
O L L S C O I L L U I M N I G H

Department of CSIS

FINAL YEAR PROJECT - 2017

By: **Abdul Halim**

ID: 13029096

Bachelor of Science in Computer Systems (LM051)

Supervised by: **J. J. COLLINS**

Abstract

The aim of this project was to leverage Message Passing Interface(MPI) API for the parallelisation of a machine learning paradigm running on a distributed-memory model infrastructure. This High Performance Computing (HPC) architecture is a Beowulf cluster based on commodity hardware and open source software. MPI defines a set of library routines and environment variables to facilitate execution of tasks in parallel across computing nodes using communication and synchronisation constructs. The Machine Learning paradigm is a Simple Genetic Algorithm (SGA), an adaptive heuristic search and optimisation algorithms based on evolutionary ideas of natural selection and genetics, used for function optimisation in this work. Empirical results are presented on the performance gain using the Beowulf cluster. The findings demonstrate the challenges faced by programmers when trying to balance the communication overhead in MPI and computational complexity when parallelising a program using MPI.

Acknowledgement

First of all, I would like to thank my supervisor J. J. Collins for introducing me this challenging project and providing excellent guidance and support throughout. Without these support and guidance this project would have been nearly impossible to show any substantial outcome.

I want to thank Dr. Norah Power for giving me the opportunity to study this course in CSIS department. I remember the day when I submitted my application for admission still unsure whether I would be accepted or not. And the day when Dr. Power met us in this department for a introduction and a confirmed of a place.

I also want to thank Mr. Tom Mulcahy of Intel Shannon for believing in me and awarding me the *Intel Shannon Paul Whelan Scholarship*. Having received this scholarship meant a lot. It kept my motivation strong throughout.

Many thanks to my family and friends that supported me doing this difficult course for four long years as a mature student. It was a very long journey but I know it is worth every single moment that I spent in this University. I hope you all can forgive me for being so distant sometimes. I hope I can make it up to you all.

Thanks to Josh, Enda, Ben, Angie, Seun, David, Ankit just to name a few that I have had the pleasure to work with as a team on many projects.

Last but not least, a BIG thanks to all CSIS academics and staff members.

All my works are dedicated to my two little stars; Lamya and Ayaan.

Declaration

I declare that this is my work and all contributions from other persons have been appropriately identified and acknowledged.

Contents

Abstract	i
Acknowledgement	ii
Declaration	iii
List of figures	x
List of tables	xi
1 Introduction	1
1.1 Summary	1
1.2 Objectives	3
1.3 Contributions	4
1.4 Methodology	4
1.5 Overview of the Report	5
1.6 Motivation	6
2 Background	8
2.1 Parallel Computing	8
2.1.1 What is Parallel Computing	8
2.1.2 Why Parallelism	8
2.1.3 Sequential vs. Parallel execution	9
2.1.4 Types of Parallelism	10

2.1.5	Flynn's Classical Taxonomy	11
2.1.6	Classification of Parallel Computers	11
2.1.7	Parallel Computer Memory Architecture	12
2.1.8	Limitation of Parallel Programming	14
2.1.9	Application	17
2.2	OpenMP and MPI	17
2.2.1	OpenMP	17
2.2.2	MPI	24
2.2.3	Hybrid Model	33
2.3	Beowulf Clusters	33
2.3.1	Architecture	34
2.3.2	Hardware	34
2.3.3	Networking	35
2.3.4	Software Infrastructure	35
2.4	Simple Genetic Algorithm (SGA)	36
2.5	Parallel GA	44
2.5.1	Classification of Parallel GAs	45
2.5.2	Master-slave Model - Global population	45
2.5.3	Distributed Model - Coarse grained	46
2.5.4	Cellular Model - Fine grained	46
3	Design and Implementation	48
3.1	Introduction	48
3.2	Setting up the Beowulf Cluster	49
3.2.1	Cluster components	49
3.2.2	Topological design	51

3.2.3	Virtual nodes	51
3.2.4	Physical nodes	51
3.2.5	Setup & Configuration	52
3.3	Object-Oriented GA	59
3.4	Issues with MPI support for Object-oriented paradigm	62
3.5	Open Source Procedural GA	63
3.6	Parallelising Procedural GA using MPI	66
3.6.1	Parallel GA model	66
3.6.2	Initialising MPI environment	67
3.6.3	Creating MPI derived datatype for genotype	67
3.6.4	Partitioning of data	69
3.6.5	PGA v1	70
3.6.6	PGA v2	73
3.6.7	PGA v3	76
3.6.8	Issues with Serialising Dynamic Data	79
3.7	Testing	83
3.7.1	Test Configuration	83
3.7.2	Tests Outcome	85
3.8	Discussion	90
4	Empirical studies	91
4.1	Introduction	91
4.2	Experiments using PGA v1	94
4.2.1	Experiment 1: Varying number of processes	94
4.2.2	Experiment 2: Varying population size	95
4.3	Experiments using PGA v2	97

4.3.1	Experiment 3: Varying number of processes	97
4.3.2	Experiment 4: Varying population size	99
4.4	Experiments using PGA v3	102
4.4.1	Experiment 5: Varying number of processes	102
4.4.2	Experiment 6: Varying population size	104
5	Conclusions	107
	Appendices	113
A	Project plans	114
B	Git logs	116
C	Project source codes	117
C.1	params.hpp	117
C.2	sga_proc.hpp	117
C.3	sga.cpp	118
C.4	pga_proc.hpp	121
C.5	pga.cpp	122
C.6	pgav2.cpp	125
C.7	sga_proc.cpp	129
C.8	pga_proc.cpp	149

List of Figures

2.1	Sequential processing where instructions (t1, t2, ..., tN) of a problem are executed by a single processors sequentially. (Barney, 2016 <i>a</i>)	9
2.2	Parallel processing where a problem is broken down into a number of sub-problems with their own set of instructions (t1, t2, ..., tN) that can be executed in parallel by different processors. (Barney, 2016 <i>a</i>)	10
2.3	A Shared Memory model where multiple CPU is sharing a common memory space. (Barney, 2016 <i>a</i>)	13
2.4	A Distributed Memory model where multiple CPUs have their own memory spaces but connected together with network bus to exchange messages. (Barney, 2016 <i>a</i>)	13
2.5	A Hybrid Memory model where multiple shared-memory systems are interconnected together by a network communication to exchange messages. (Barney, 2016 <i>a</i>)	14
2.6	Amdahl's Law shows that speedup increases both as a function of the percentage of parallelism and the number of parallel units available. (RTC Magazine, 2013)	15
2.7	Comparison of speedup of fixed-sized and scaled-sizes parallel portion of a program.	16
2.8	Fork- Join model. (Wikipedia user: A1, 2007)	19
2.9	MPI program structure	26

2.10	A basic Beowulf architecture. (Atom.me.gatech.edu, 2016) . .	34
2.11	Single-point crossover	40
2.12	Multi-point crossover	40
2.13	Uniform crossover	41
2.14	Mutation	42
2.15	Master-Slave PGA model.	46
2.16	Distributed PGA model.	47
2.17	Cellular PGA model.	47
3.1	Layers in a Cluster	50
3.2	A typical Beowulf setup	52
3.3	Class diagram of Object-oriented SGA	61
3.4	Flowchart diagram of Implemented master-slave Parallel GA .	68
3.5	PGA v1 model	71
3.6	PGA v2 model	74
3.7	PGA v3 model	78
3.8	SGA output - Top part	86
3.9	SGA output - Bottom part	86
3.10	PGA v1 output - Top part	87
3.11	PGA v2 output - Bottom part	87
3.12	PGA v1 output - Top part	88
3.13	PGA v2 output - Bottom part	88
3.14	PGA v3 output - Top part	89
3.15	PGA v3 output - Bottom part	89
4.1	Experiment 1 - Runtime comparison SGA vs PGA v1.	94
4.2	Experiment 1: Speedup graph SGA vs PGA v1.	95

4.3	Experiment 2 - Runtime comparison SGA vs. PGA v1.	96
4.4	Experiment 2 - Speedup graph SGA vs. PGA v1.	97
4.5	Experiment 3 - Runtime comparison SGA vs. PGA v2.	98
4.6	Experiment 3 - Speedup graph SGA vs. PGA v2.	99
4.7	Experiment 4 - Runtime comparison SGA vs. PGA v2.	100
4.8	Experiment 4 - Speedup graph SGA vs PGA v2.	101
4.9	Experiment 5 - Runtime comparison SGA vs. PGA v3.	102
4.10	Experiment 5 - Speedup graph SGA vs. PGA v3.	103
4.11	Experiment 6 - Runtime comparison SGA vs. PGA v3.	105
4.12	Experiment 6 - Speedup graph SGA vs. PGA v3.	105
A.1	Gantt Project plans - List	114
A.2	Gantt Project plans - Graph	115
B.1	PGA Version control logs.	116

List of Tables

3.1	Virtual nodes configuration	52
3.2	Physical nodes configuration	52
3.3	Parameters alues used with test cases.	84
3.4	Genome value range for 3 variables Griewank function	84
4.1	A sample runs using population size of 5000 to determine average runtime(seconds) for PGA v1, PGA v2 and PGA v3.	93
4.2	Experiment 1 data using PGA v1	94
4.3	Experiment 2 data using PGA v1	96
4.4	Experiment 3 data using PGA v2	98
4.5	Experiment 4 data using PGA v2	100
4.6	Experiment 5 data using PGA v3	102
4.7	Experiment 6 data using PGA v2	104

Chapter 1

Introduction

1.1 Summary

This project explores the Message Passing Interface (MPI) parallel programming paradigm targeting a distributed memory architecture. A Simple Genetic Algorithm (SGA) is parallelised using MPI and deployed on a cluster. An empirically-based analysis of the performance is then presented.

From the early days, computers were traditionally programmed to execute instructions sequentially. Software Engineers have traditionally relied on Moore's Law (Sutter, 2005) to satisfy their requirement for performance for their serial programs. Moore's Law (Schaller, 1997) states that there is a doubling of the transistor count every 18th months, an effective doubling of the workload that the CPU can accommodate . However, it is becoming more difficult to dissipate the heat from chips with increasing transistor count (Kim et al., 2003). An alternative approach (Sutter, 2005) is to increase the number of cores and run them at slower clock speeds. Programmers are now faced with the challenge of parallelising their serial application in order to exploit the parallel deployment infrastructure.

Today, a smart phone in one's hand packs more computing power than some supercomputers in 1960's. Multi-core systems are everywhere. The biggest challenge programmers face today is utilising these computing power efficiently

and effectively. There are some problems that can only be addressed using parallel processing.

Parallel processing is the execution of a program instructions among multiple processors concurrently with the objective of running it in less time(Barney, 2016a). Parallelism is an important aspect in computing because we no longer live in a world where we could just expect that the Moore's Law will take care of all of the performance needs. Processor's clock speed are not getting much faster as it used to be in years before. The transistors sizes are shrunk so much that we are now facing the challenges on physical limitation of matter and laws of the universe. Increasing clock speed is no longer a viable solution for improve computing performance. In his infamous article titled "*The Free Lunch Is Over*" Herb Sutter went on a great details why it is not possible to add on more clock speed on the processors any longer(Sutter, 2005). Hardware vendors shifted their focus on adding more processing cores in a single silicon die to add more speed. But using these cores in an efficient and scalable manner can be very challenging.

Further more, many problems are so large and/or complex that it is impractical or impossible to solve them on a single machine. Parallel processing is essential for modelling and simulating real world phenomena. These problems are too complex and large to process sequentially.

Parallel solutions sought for problems that demand speed, process a huge amount of data, time-sensitive or time-critical(real time) (Barney, 2016a). In general, any problems that is computation greedy better suited for parallel implementation. But not all programs are suitable for executing in this manner. In order to execute a program in parallel it must have some regions that can be executed concurrently without affecting the end result (Sutter, 2005). There are number of challenges in parallel programming. These challenges include finding concurrency in a program, tasks decomposition & scheduling, synchronisation & communication, scaling and debugging (Sutter, 2005) . As part of learning the parallel programming paradigm it was intended to research on these challenges and explore the techniques and best practices available.

Genetic Algorithms(GAs) are adaptive heuristic search and optimisation algo-

rithms based on evolutionary ideas of natural selection and genetics(Goldberg, 1989). GAs powerful search and optimisation techniques are found to be very useful and effective for solving problems in many different disciplines. GAs have a wide range of interesting application area from machine learning to robotics, engineering design, financial and investment decision making(Konfrst, 2004). There are many aspects in GAs that can take advantages of parallel processing for improving performance and efficiency and better results. Parallel implementation of GAs promise a substantial gains in performance(Cantú-Paz, 1998).

Computer cluster is group of computers connected to a local area network that can run parallel applications(Gropp et al., 2003). Beowulf cluster is a parallel computing infrastructure that originated with the idea of processing tasks in parallel by leveraging commodity hardware and open source software(Beowulf.org, 2016).

This project is aimed at parallelising a sequential SGA to run on a distributed-memory model parallel environment - Beowulf cluster, using Message Passing Interface (MPI), and analyse the performance. MPI is a standard specification for Message-Passing parallel programming model for distributed-memory system in which data is moved from one process's address space to another through cooperative message exchange (Gropp et al., 1999). For this project a Beowulf cluster was implemented. At first, the Beowulf was simulated on a VirtualBox in a portable computer. Later on, the cluster was implemented using inexpensive commodity hardware. The popular MPI implementation MPICH4.0 was installed in both setup to provide the MPI environment to compile and run parallel GA's for this project. This involved carrying out other required installation, configuring and administrative tasks on Ubuntu Server 16.04 Linux distribution.

1.2 Objectives

The primary objectives are twofold:

1. Parallelise a Simple Genetic Algorithm using MPI for deployment on a distributed memory architecture such as a Beowulf cluster, and
2. Profile the performance when free parameters are modified such as the number of nodes in the cluster and population size of the SGA.

Since this project spans to a dimension that covers quite few challenging topics in computing, the secondary objectives of this projects are listed as below:

- Implementing and administering Beowulf Cluster
- Learning about parallel computing technology
- Learning about parallel programming techniques
- Study of Message-Passing-Interface(MPI) API
- Study of OpenMP API
- Exploring Simple Genetic Algorithms
- Exploring the parallelisation techniques of GAs

1.3 Contributions

This author is not aware of any other existing parallel Genetic Algorithm implementation using Message Passing Interface. The work done on parallelising GA in this project and carried out empirical studies on performance by varying population size on a Beowulf cluster is thought to be first in this field.

1.4 Methodology

The following methodology were adapted for this project:

1. Define the research question

The first step into this project was to identify what is the research question that this project is going to answer. There are many aspects in a parallel implementation of SGA to experiment with. We identified how the performance of a parallel GA reflects in terms of number of processes and in terms of population size.

2. Systematic Literature Review

The next step is to identify and select articles, books and Internet resources that are relevant for this project. This is how one carries out research on the topics of Parallelisation, Cluster computing, OpenMP and MPI, and Genetic Algorithms.

3. Propose a working hypothesis

4. Define parameter of the empirical study clearly specify the objectives of each test case

5. Build the prototype necessary to generate the data for the empirical study

6. Run the experiments and capture the data

7. Analyse the results to see if they answer the research question by supporting the working hypothesis.

- (a) If not, go back to step 3.

- (b) If new results are obtained, go back to step 1.

1.5 Overview of the Report

This report is broken into few major sections.

Introduction This section contains a basic overview of the project, objectives, motivations and a brief discussions on other contributions related to this project.

Background Research This section discusses the findings of existing research in the area of parallel computing, OpenMP, MPI, Simple Genetic Algorithms and Parallel Genetic Algorithms.

Design and Implementation This section describes the design and implementation part of the project under taken. It covers the steps of implementing Beowulf clusters, Parallel implementation of Simple Genetic Algorithms, product development tools and techniques.

Empirical Studies This section contains the findings and comparative analysis of performance of implemented parallel GA's on Beowulf cluster.

Discussion and Conclusions This section contains the discussions on findings in empirical studies and conclusions.

1.6 Motivation

During my Co-Operative(COOP) education placement at Intel I worked with teams on a number projects on computational intensive network packet processing. These application demand fully utilising the multi-core Intel Xeon processors for increased throughput. While in there I also came across many other projects in the area of Software Defined Networking(SDN) and Network Function Virtualisation(NFV). Even though each projects were different in their own field, one thing were common. These projects were addressing a common aspect; how to increase the data processing performance. The obvious answer to this question is parallel processing and multi-threading. Programmer must be able to write parallel programs to take advantage of these multi-core systems.

I realised the importance of the parallel programming while working on these real world projects at my COOP placements. My main motivation for this project is to learn about parallel programming.

I am also interested on Evolutionary Algorithms and their application in computation. While studying Intelligent System(CS4006) module for this course we were introduced to Genetic Algorithms. Prior to that I had no idea how the evolutionary theories can be applied to find solution that otherwise impossible to find or too costly. For that module I worked on an individual project on optimal visualisation of graph using Simple Genetic Algorithms which I enjoyed very much.

When I found this project on the FYP proposal list, I approached my supervisor to express my interests on this project. However, this project involves implementing MPI on Beowulf cluster. Neither of these terms I was familiar with. But I was equally excited to learn about cluster computing and programming using MPI. I saw this as a great opportunity to step into the High Performance Computing area. Which is exciting!

Chapter 2

Background

2.1 Parallel Computing

2.1.1 What is Parallel Computing

Parallel Computing is the simultaneous use of multiple computing resources such as processors and memory to solve a computational problem (Barney, 2016*a*). Parallel computing is based on the principle of breaking a large problem into smaller problems and solving these smaller problems concurrently on multiple processors.

2.1.2 Why Parallelism

- There are physical limit of hardware components how fast they can run
- Large application demands more memory and computation time
- There are problems that demand real-time constraint

2.1.3 Sequential vs. Parallel execution

Sequential execution

Single sequence of instructions is executed one at time by a single processing unit on a single sequence of data (Akl and Nagy, 2009). Figure 2.1 shows a problem consists of a set of instructions (t_1, t_2, \dots, t_N) that are executed by the processor in a sequential manner.

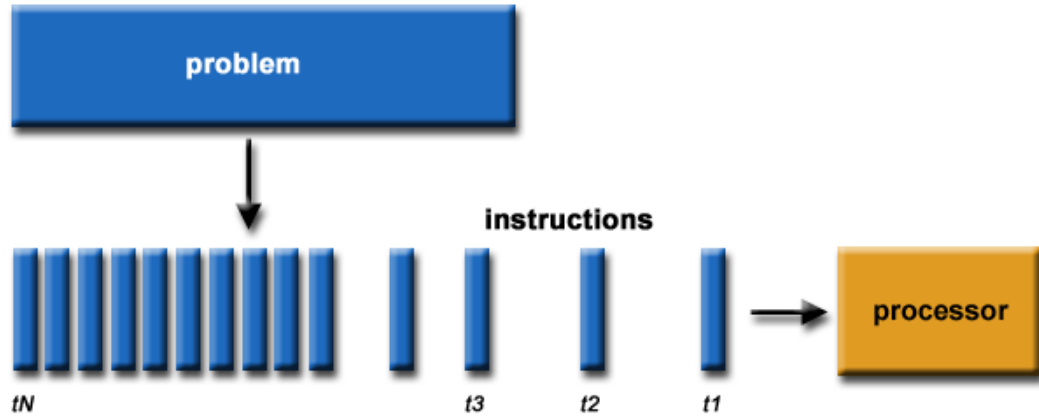


Figure 2.1: Sequential processing where instructions (t_1, t_2, \dots, t_N) of a problem are executed by a single processors sequentially. (Barney, 2016*a*)

Parallel execution

In parallel execution a problem can be broken into smaller sub-problems and two or more processors can work on it simultaneously. During this phase the processors may need to communicate with each other to exchange partial results. At the end of the computation the interim results from each processors must be combined to form the final solution to the original problem. Figure 2.2 shows a problem is broken down into sub-problems each of which then consist of a set of instructions (t_1, t_2, \dots, t_N) that are executed by different processors in parallel.

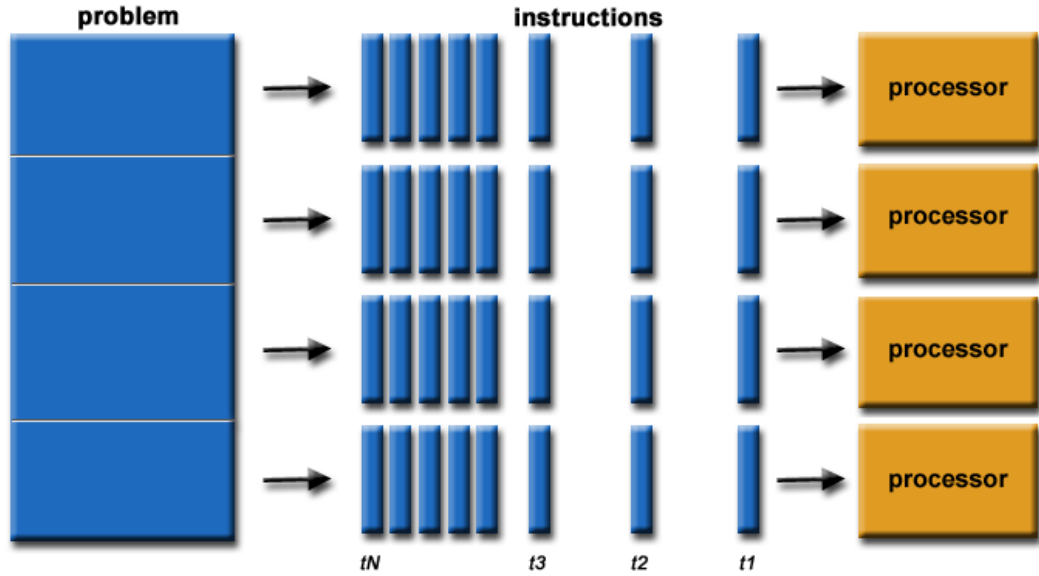


Figure 2.2: Parallel processing where a problem is broken down into a number of sub-problems with their own set of instructions ($t1$, $t2$, ..., tN) that can be executed in parallel by different processors. (Barney, 2016a)

2.1.4 Types of Parallelism

Data parallelism In Data parallelism the input data is divided among different processors. The same task runs on different parts of the data in parallel. Data parallelism is a consequence of single operations that is being applied on multiple data items

Task Parallelism In Task parallelism different tasks run on the same data. In this model, parallelism is expressed by a task graph. A task graph can be either trivial or nontrivial. The correlation among the tasks are utilised to promote locality or to minimise interaction costs. This model is enforced to solve problems in which the quantity of data associated with the tasks is huge compared to the number of computation associated with them. The tasks are assigned to help improve the cost of data movement among different tasks.

Hybrid data/task parallelism A parallel pipeline of tasks, each of which might be data parallel

2.1.5 Flynn's Classical Taxonomy

This is known as one of the earliest classification of computers and programs created by Michael J. Flynn (Barney, 2016*a*). Flynn classified programs and computers on instructions stream and data stream.

SISD Single Instruction Single Data

Only one instruction is being acted on by the CPU during one clock cycle and only one data stream is being used as input.

SIMD Single Instruction Multiple Data

All processing unit execute same instruction at any given clock cycle but each processing unit act upon different data input.

MISD Multiple Instructions Single Data

Each processing unit execute separate instruction stream independently on a single data stream.

MIMD Multiple Instructions Multiple Data

Each processor executes different instruction stream and all of them work on different data input stream.

This classification model distinguishes multi-processors computer architecture based on how they can be classified along the two independent dimension of Instruction Stream and Data Stream. There are further classification exist based on multiple data stream. **Single Program Multiple Data (SPMD)** is a subcategory of MIMD where multiple processors simultaneously execute same program on different data input.(Barney, 2016*a*)

2.1.6 Classification of Parallel Computers

Parallel computers can be classified based on their hardware supports.

- **PVP (Parallel Vector Processor)**

PVP systems contain a small number of powerful custom made vector processors with shared memory.

- **SMP (Symmetric Multiprocessor)**

SMP system consist of a number of processors connected through a bus system. Each processor has equal access to memory, I/O etc.

- **MPP (Massively Parallel Processor)**

MPP is very large-scale computer system with many processors and physically distributed memory space. This system requires high communication bandwidth and low latency interconnection between processors. It is a tightly couple system.

- **COW (Cluster of Workstation)**

COW is a low-cost variation of MPP which consist of multiple nodes(a complete workstation) connected by low-cost network. Nodes work together as a single integrated computing resource. This system is loosely coupled and highly extensible. Beowulf cluster falls into this category.

- **DSM (Distributed Shared Memory)**

DSM uses special hardware and software extension to support distributed coherent caches. In contrast with SMP system, DSM consist of distributed-memory but hardware and software gives an illusion of shared-memory behaviour.

(Perkowski, 2010)

2.1.7 Parallel Computer Memory Architecture

Shared Memory model Multiple processor operate independently but share the same address space. Figure 2.3 shows the shared-memory model architecture.

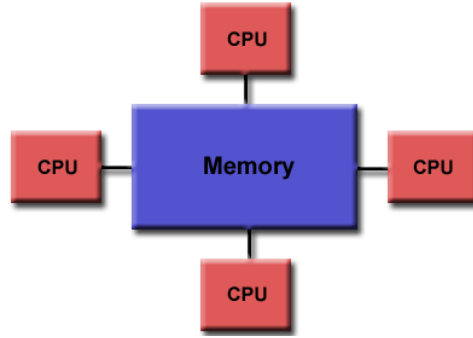


Figure 2.3: A Shared Memory model where multiple CPU is sharing a common memory space. (Barney, 2016*a*)

Distributed Memory Model Processors have their own local address space. Memory address of one processor do not map to another. A message-passing mechanism is required for data sharing. Figure 2.4 shows the distributed-memory model architecture.

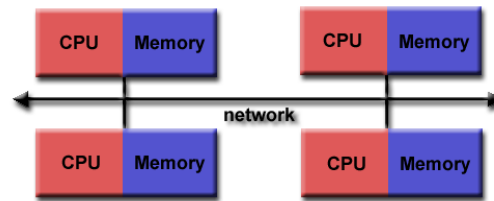


Figure 2.4: A Distributed Memory model where multiple CPUs have their own memory spaces but connected together with network bus to exchange messages. (Barney, 2016*a*)

Hybrid Model Hybrid model is the combination of shared-memory interconnected by network communication that allows distributed-memory access of data resides in another processors global space. Figure 2.5 show the Hybrid-memory model architecture.

(Barney, 2016*a*)

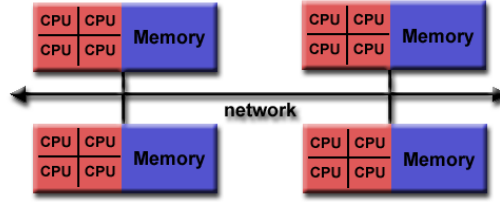


Figure 2.5: A Hybrid Memory model where multiple shared-memory systems are interconnected together by a network communication to exchange messages. (Barney, 2016a)

2.1.8 Limitation of Parallel Programming

Amdahl's Law

Amdahl's law states that a programs speedup of an algorithm on parallel computing platform is defined by:

$$S = \frac{1}{(1 - f) + \frac{f}{N}}$$

Where,

- S is the maximum speedup in execution time of entire task
- f is the percentage of the task that can be parallelise
- N is number of parallel processors

From Amdahl's Law it become apparent that there is limits to the scalability of parallelism. The speedup gain on parallelising a tasks is dependent on the ratio of paralellisable codes (Barney, 2016a). The graph in Figure 2.6 shows the limit of speed up of parallel execution of code. This Law argued that there exist a speed up limit; which is $1/(1 - f)$.

Gustafson's Law

The problem with Amdahl's law is that, it is assumed that the problem size is fixed for both serial and parallel parts. In an empirical studies by John

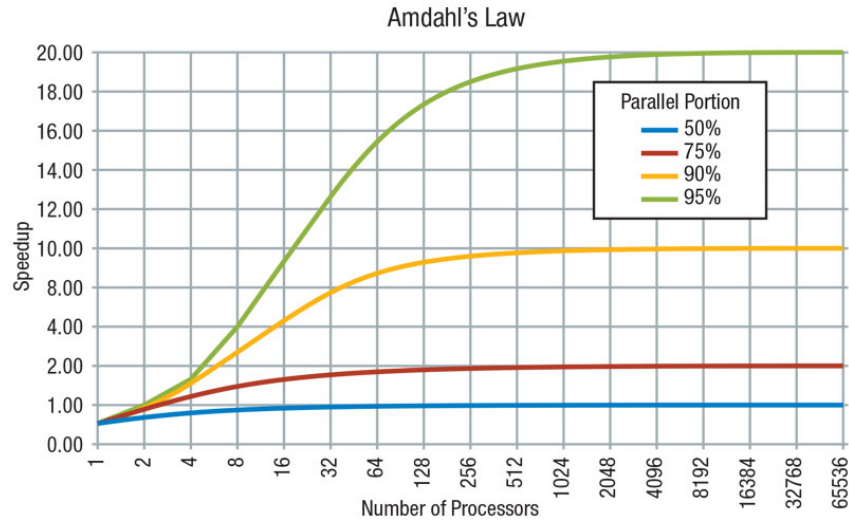


Figure 2.6: Amdahl's Law shows that speedup increases both as a function of the percentage of parallelism and the number of parallel units available. (RTC Magazine, 2013)

Gustafson (Gustafson, 1988) showed that when problem size scales up, by scaling up the number of parallel processors the execution time can be kept fixed. The limitation imposed by the serial parts of a program may be countered by increasing the total amount of computation. This is known as Gustafson's law.

Gustafson's law is formulated as:

$$S = (1 - f) + fN$$

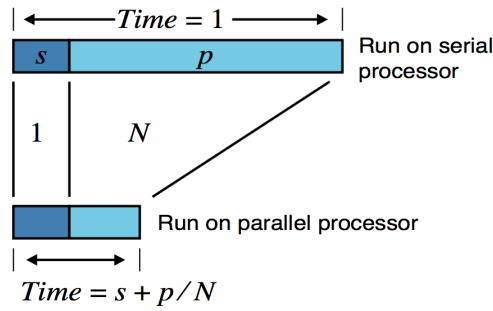
Where,

- S is the maximum speedup in execution time of entire task
- f is the percentage of the task that can be parallelise
- N is number of parallel processors

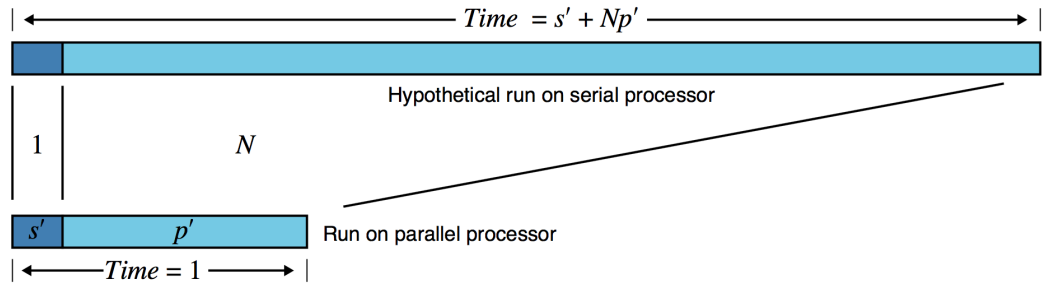
In Figure 2.7 the two models of parallel speedup contrasted and summarised in terms of gains based on fixed-size and scaled sized parallel portion in a

program. As described by Amdahl's Law, if a program with s serial portion and p parallel portion requires 1 time unit to run on a serial processors then it requires $s + p/N$ time unit on N parallel processors. Therefore, the overall speed up is $1/(s + p/N)$. It is assumed that the parallel portion p is fixed.

In Figure 2.7b shows if the parallel portion of the program increases (where serial portion is shown as s' and parallel portion as p') time requires to run this program on serial processor is $s' + Np'$ in comparison to running it on N parallel processors. Therefore, if the number of processors N is increased in proportion to the scaled up parallel portion of the program then the speedup is $s' + Np'$.



(a) A program with s serial portion and parallel portion p requires 1 time unit to run on a serial processors and $s + p/N$ on N parallel processors. Therefore, the overall speed up is $1/(s + p/N)$.



(b) A program with s' serial portion and p' parallel portion requires 1 time unit to run on N parallel processors and $s' + Np'$ unit time on a serial processor. Therefore, the overall speed up is $(s' + Np')/1$ i.e. $s' + Np'$.

Figure 2.7: Comparison of speedup of fixed-sized and scaled-sizes parallel portion of a program.

(Gustafson, 1988)

2.1.9 Application

Parallel computation many application areas, which include:

- Scientific Computing
- Simulation
- Aerospace
- Weather forecasts - Climate changes
- Astronomy - Planetary movement
- Big Data and Data mining

(Barney, 2016*a*)

2.2 OpenMP and MPI

2.2.1 OpenMP

OpenMP stands for Open Multi-Processing. OpenMP is an Application Programming Interface (API) to facilitate parallel programming on shared memory systems. (Chapman et al., 2008) It is a set of compiler directives and supporting callable runtime library routines and environment variables that extends C/C++ and Fortran to express shared memory parallelism. (Dagum and Menon, 1998)

In 1990's OpenMP was defined by OpenMP Architecture Review Board (ARB), formed by a group of leading Hardware and Software vendors to provide a common API for programming a range of SMP architectures. This was based on earlier similar work done by Parallel Computing Forum(PCF). Unlike MPI

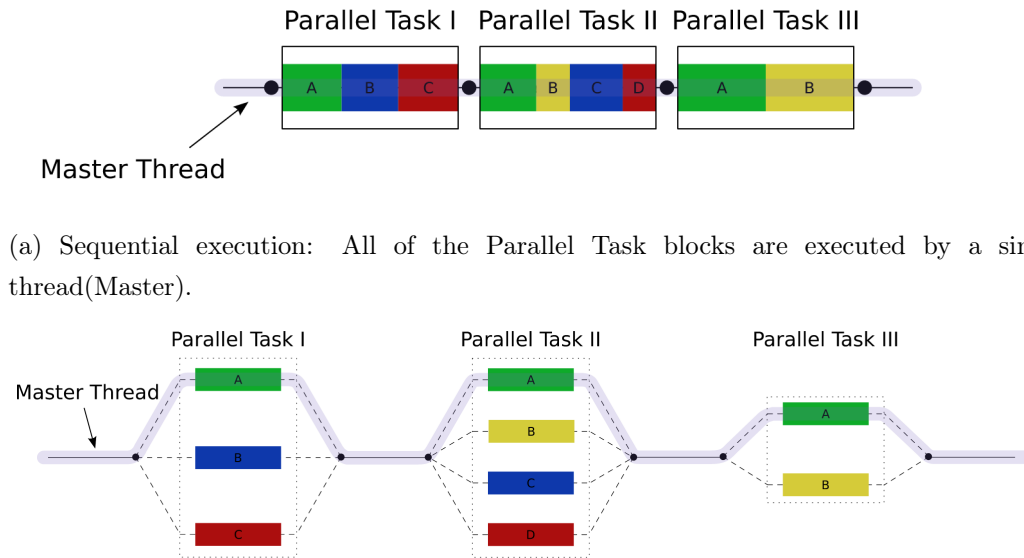
which we will discuss below, OpenMP is not a standard. Instead, it is considered as an agreement between members of ARB who shares common interest in portable, user-friendly and efficient approach to shared memory parallel programming. (Chapman et al., 2008) OpenMP API was designed to provide an easy method to thread application without requiring a programmer to know how to create threads, synchronise and destroy them or even how many to create.

OpenMP does not define a new language. It is a set of compiler directive that can be added to a sequential program to describe how a work can be shared among threads executing on different processor cores and synchronise their access to shared data if needed. These compiler directives make it easy to modify a sequential program to take advantage of shared-memory parallel architecture with a minimum modification to existing codes. OpenMP is simple to use because the complexity of parallelism is handled by the compiler and also it emphasised on structured parallel programming. (Chapman et al., 2008)

Thread based parallelism

OpenMP is based on Multithreading. Program launches as a single thread which can create other threads if required. It utilises the "fork" and "join" model of parallel execution. All OpenMP program starts as a single process which is known as master thread. The master thread continues the program execution sequentially to the point where it encounter a parallel region of the program defined by a OpenMP construct directive. (Barney, 2016c)

In Figure 2.8 shows the fork-join model of parallel execution. It shows a program containing parallel executable code fragments. These code fragments are put into three region labelled as **Parallel Task I, Parallel Task II and Parallel Task III** to show the synchronisation points. Within these points program must synchronise before advancing. Each group contain code fragments of subtasks (labelled as A, B, C, D etc.) that are independent of each other and can be executed in parallel asynchronously. These subtasks can



(a) Sequential execution: All of the Parallel Task blocks are executed by a single thread(Master).

(b) Parallel execution: At each Parallel Task block the master thread forks multiple team threads which then execute the subtasks within that parallel region concurrently. When finished the team threads then join with master thread and terminate before master thread advancing to the next parallel region.

Figure 2.8: Fork- Join model. (Wikipedia user: A1, 2007)

take advantage of concurrent execution by multiple threads. Figure 2.8a shows the program is executed by a single thread(Master thread). The Figure 2.8b shows that the same program is parallelise using fork-join model. When master thread reaches a parallel region it forks multiple team threads as required by that region. These team threads then execute the subtasks (A, B, C, D etc.) assigned to them concurrently. When the team threads complete executing the subtasks, they join back to the master thread and terminate. The master thread is then continue with rest of the program.

OpenMP Components

The OpenMP API comprised of three primary components:

- Runtime Library Routines
- Environment Variables

- Compiler directives

Runtime Library Routines

The Runtime library routines are used for setting up and identifying number of threads, thread numbers for identification, dynamically adjusting of threads, timers, nested parallelism and for locking mechanism. (Barney, 2016*c*)

Environment Variables

OpenMP provides several environment variables for controlling the execution of parallel code at runtime. These variables can be used to control the number of threads, scheduling type, binding threads to processor, controlling level of nested parallelism, dynamic thread adjustment, stack size and thread wait policy. (Barney, 2016*c*)

Compiler Directives

Compiler directives are special constructs that specify how a compiler should process its inputs. OpenMP API uses compiler directives for C, C++ and Fortran compilers for marking a parallel region, distributing works among threads, dividing up loop iterations, serialising section of codes and for synchronisation of the work. In C/C++ source code the directives appears as `#pragma` as shown at Line 2 of Listing 2.1. (Barney, 2016*c*)

```

1
2 #pragma omp <directive-name> [clauses , ...] newline

```

Listing 2.1: OpenMP directive format in C/C++

Using OpenMP compiler directives a programmer instructs the compiler which part of program to run in parallel and how to divide the work among threads that will run it. These directives are meaningful to the compilers only. OpenMP directives looks like comments in Fortran programs and `pragma` in C or C++ program. The advantage of such implementation is that a program containing

OpenMP directives will still compile and run even if a compiler is unable to interpret these directives.

The line 5 in "Hello world" program showing in Listing 2.2 is a compiler directive in C program that marks the region as a starting point of the parallel execution.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     #pragma omp parallel
6     printf("Hello, world.\n");
7     return 0;
8 }
```

Listing 2.2: A "Hello world" program in C using OpenMP compiler directive

OpenMP directives can be classified in following categories:

- Parallel region construct
- Work-sharing constructs
- Synchronisation constructs
- Data-scoped attribute clauses

Parallel region construct A parallel region is a block of code that will be executed by multiple threads. This construct forks additional threads to carry out the work within the next parallel code block. Line 5 in Listing 2.2 shows the parallel directive that specifies a parallel region (Barney, 2016c)

Work-sharing constructs A work-sharing construct specify how independent work to be assigned to one or more threads. The *omp for* and *omp do* is used to divide up loop iteration among multiple threads. The *section* clause is for assigning non-iterative consecutive but independent code blocks to multiple threads. The *single* clause specify that a code block is to be executed by

only one single thread. A code block to be executed only by master thread the *master* clause is used. Listing 2.3 shows an example of *for* work-sharing construct. (Barney, 2016c)

```
1  #include <omp.h>
2  #define N 1000
3  #define CHUNKSIZE 100
4
5  main(int argc, char *argv[]) {
6      int i, chunk;
7      float a[N], b[N], c[N];
8
9      /* Some initializations */
10     for (i=0; i < N; i++)
11         a[i] = b[i] = i * 1.0;
12     chunk = CHUNKSIZE;
13     #pragma omp parallel shared(a,b,c,chunk) private(i)
14     {
15         #pragma omp for schedule(dynamic,chunk) nowait
16         for (i=0; i < N; i++)
17             c[i] = a[i] + b[i];
18     } /* end of parallel region */
19 }
```

Listing 2.3: An example of Work-sharing "for" loop construct.

Synchronisation constructs OpenMP provides synchronisation constructs to deal with race-condition to ensure that the correct results is produced. The synchronisation constructs are used to control how the execution of each thread progress in relation to other threads. The example in Listing 2.4 shows the use of Critical directive to define a critical region a program. (Barney, 2016c)

```
1  #include <omp.h>
2
3  main(int argc, char *argv[]) {
4      int x;
5      x = 0;
6
7      #pragma omp parallel shared(x)
```

```

8      {
9      #pragma omp critical
10     x = x + 1;
11     } /* end of parallel region */
12 }
13

```

Listing 2.4: An example of Critical directive construct.

Data-scoped attribute clauses As OpenMP is shared-memory model of parallelisation, most variables are shared among multiple threads. OpenMP provides number of clauses to deal with variable scopes to define their visibility. These clauses are used in conjunction with other directives to control the scope of enclosed variable within a parallel code block. (Barney, 2016c)

Advantages & Disadvantages of OpenMP

Advantages The following are the advantages and disadvantages of OpenMP as discussed on Dartmouth College website on parallel programming. (Dartmouth College, 2011).

- Good performance and scalability
- OpenMP programs are portable
- Easier to program and debug compared to MPI
- Allow program to be parallelised incrementally

Disdvantages

- Can only be run in Shared-memory computers
- Requires compiler that supports OpenMP
- Mostly used for loop parallelisation

2.2.2 MPI

MPI stands for **M**essage **P**assing **I**nterface. MPI is a standard specification for Message-Passing parallel programming model in distributed memory architecture. In Message-passing parallel model data is moved from one process's address space to another through cooperative message exchange. MPI itself is not a library but simply provides a standard specification for writing message passing programs. This standard defines the syntax and semantics of essential library routines that facilitates developers writing platform independent message-passing programs in C, C++ and Fortran. (Barney, 2016b)

MPI standard was a result of the works of many individuals and groups to address the problems they encounter in distributed memory model parallel programming. The need for an standard arose when a number of incompatible tools were developed independently by different groups. In 1992 a workshop on standards for Message-Passing in distributed memory environment was held sponsored by Center for Research on Parallel Computing, Williamsburg. This was followed by a draft proposal of MPI1 and forming of MPI Working group. The MPI1 draft proposal was presented in following year at Supercomputing 93 conference. MPI Working group held regular meetings, public comments and open mailing lists which later on constituted as **MPI Forum** (Barney, 2016b). The current version of MPI standard is MPI 3.1 which was approved by MPI forum in June 2015. The next major version, MPI 4.0 is work in progress, which focuses on support on fault tolerance, improved support for hybrid programming models and application hints to MPI libraries to enable optimisations(Mpi-forum.org, 2016).

MPI Concepts

The Message-passing approach makes data exchange cooperative between processes. Data is explicitly sent by one process and received by another. MPI is communication protocol and semantic specifications for how its features must behave in any implementation (Gropp et al., 1999).

To discuss various MPI constructs an example of matrix multiplication program using MPI is shown in Listing 2.5. This code example is referenced in following subsections to explain some of these constructs.

General MPI Program Structure

Figure 2.9 shows a general MPI program structure. MPI header files must be included by all programs that make MPI library call. MPI library calls are only permissible after initialising and before terminating the MPI environment (Barney, 2016b). In the example program in Listing 2.5 line 28 initialises the MPI environment and line 111 terminates it.

Groups and Communicators

MPI uses groups and communicators objects to define which process can communicate with each other (Barney, 2016b). Processes can be collected into groups. Groups are implemented as ordered list of process identifiers stored in an integer array (Gropp et al., 1996). Each message is sent in a context and must be received in the same context. A group and context together form a communicator (Gropp et al., 1999). **MPI_COMM_WORLD** is a pre-defined communicator that includes all MPI processes. Line 30 in Listing 2.5 shows that communicator **MPI_COMM_WORLD** is used to query about number of available processes.

Rank

In MPI a process is identified by its *rank* in the group associated with a communicator (Gropp et al., 1999). A *rank* is a unique integer identifier. It is assigned by the system when a process is initialised. These identifiers are contiguous and start at '0'. The *rank* id's are used by the programmers to specify the source and destination of messages (Barney, 2016b). Line 29 in Listing 2.5 shows that a process is querying its rank ID within the communicator **MPI_COMM_WORLD**.

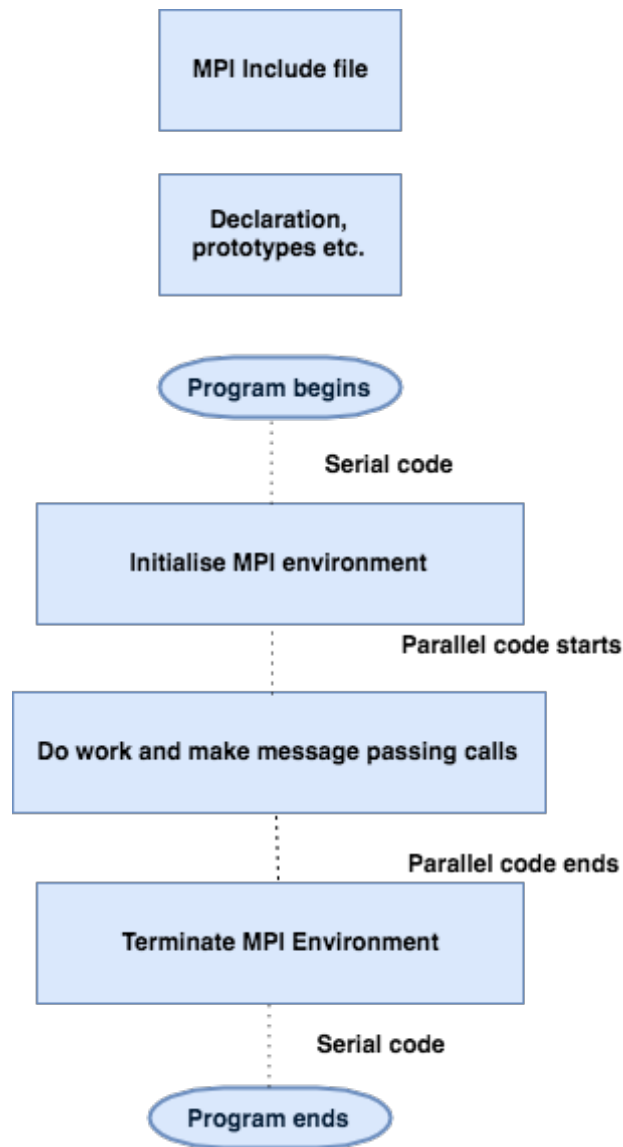


Figure 2.9: MPI program structure

Error Handling

The default action of an MPI call is to abort if there is a runtime error. However, most of the MPI function calls include a return/error code parameter. This allows a programmer to override the default behaviour of handling errors. (Barney, 2016*b*)

Point-to-Point Communication

Point-to-Point communication involves only two processes. One process performs send operation and other one performs receive operation. In MPI there are different types of send receives routines available such as:

- **Synchronous send**

`MPI_Ssend()` sends a message and blocks until the application buffer in the sending task is free for reuse and the destination process has started to receive the message.

- **Blocking send / blocking receive**

`MPI_Send()` is a basic blocking send operation. This routine returns only after the application buffer in the sending task is free for reuse. Similarly, `MPI_Recv()` is a basic blocking receive operation that receive a message and block until the requested data is available in the application buffer in the receiving task.

- **Non-blocking send / non-blocking receive**

`MPI_Isend()` is a non-blocking send operation that identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for the message to be copied out from the application buffer. A communication request handle is returned for handling the pending message status. Likewise, `MPI_Irecv()` is non-blocking receive operation that identifies an area in memory to serve as a receive buffer. Processing continues immediately without actually waiting for the message to be received and copied into the the application buffer.

(Barney, 2016*b*)

Line 59 in Listing 2.5 shows an example of blocking send using `MPI_Send()` routine my master process. There is a corresponding blocking receive call using `MPI_Recv()` at line 107.

```

1 include "mpi.h"
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #define NRA 62 /* number of rows in matrix A */
6 #define NCA 15 /* number of columns in matrix A */
7 #define NCB 7 /* number of columns in matrix B */
8 #define MASTER 0 /* taskid of first task */
9 #define FROM_MASTER 1 /* setting a message type */
10 #define FROM_WORKER 2 /* setting a message type */
11
12 int main (int argc, char *argv[])
13 {
14     int numtasks, /* number of tasks in partition */
15         taskid, /* a task identifier */
16         numworkers, /* number of worker tasks */
17         source, /* task id of message source */
18         dest, /* task id of message destination */
19         mtype, /* message type */
20         rows, /* rows of matrix A sent to each worker */
21         averow, extra, offset, /* used to determine rows sent to each
22             worker */
23         i, j, k, rc; /* misc */
24     double a[NRA][NCA], /* matrix A to be multiplied */
25         b[NCA][NCB], /* matrix B to be multiplied */
26         c[NRA][NCB]; /* result matrix C */
27     MPI_Status status;
28
29     MPI_Init(&argc,&argv);
30     MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
31     MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
32     if (numtasks < 2 ) {
33         printf("Need at least two MPI tasks. Quitting...\n");
34         MPI_Abort(MPI_COMM_WORLD, rc);
35         exit(1);
36     }
37     numworkers = numtasks-1;
38
39     /****** master task *****/

```

```

39  if (taskid == MASTER)
40  {
41      printf("mpi_mm has started with %d tasks.\n", numtasks);
42      printf("Initializing arrays...\n");
43      for (i=0; i<NRA; i++)
44          for (j=0; j<NCA; j++)
45              a[i][j] = i+j;
46      for (i=0; i<NCA; i++)
47          for (j=0; j<NCB; j++)
48              b[i][j] = i*j;
49
50      /* Send matrix data to the worker tasks */
51      averow = NRA/numworkers;
52      extra = NRA%numworkers;
53      offset = 0;
54      mtype = FROM_MASTER;
55      for (dest=1; dest<=numworkers; dest++)
56      {
57          rows = (dest <= extra) ? averow+1 : averow;
58          printf("Sending %d rows to task %d offset=%d\n", rows, dest
59              , offset);
60          MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
61          MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
62          MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype,
63              MPI_COMM_WORLD);
64          MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype,
65              MPI_COMM_WORLD);
66          offset = offset + rows;
67      }
68
69      /* Receive results from worker tasks */
70      mtype = FROM_WORKER;
71      for (i=1; i<=numworkers; i++)
72      {
73          source = i;
74          MPI_Recv(&offset, 1, MPI_INT, source, mtype,
75              MPI_COMM_WORLD, &status);
76          MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD

```

```

, &status);
73     MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source,
mtype, MPI_COMM_WORLD, &status);
74     printf("Received results from task %d\n", source);
75 }
76
77 /* Print results */
78 printf("*****\n");
79 printf("Result Matrix:\n");
80 for (i=0; i<NRA; i++)
81 {
82     printf("\n");
83     for (j=0; j<NCB; j++)
84         printf("%6.2f ", c[i][j]);
85 }
86 printf("\n*****\n");
87 printf("Done.\n");
88 }
89
90 /***** worker task *****/
91 if (taskid > MASTER)
92 {
93     mtype = FROM_MASTER;
94     MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD,
&status);
95     MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &
status);
96     MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype,
MPI_COMM_WORLD, &status);
97     MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype,
MPI_COMM_WORLD, &status);
98
99     for (k=0; k<NCB; k++)
100         for (i=0; i<rows; i++)
101         {
102             c[i][k] = 0.0;
103             for (j=0; j<NCA; j++)
104                 c[i][k] = c[i][k] + a[i][j] * b[j][k];
105         }

```

```

106     mtype = FROM_WORKER;
107     MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD)
    ;
108     MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
109     MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype,
    MPI_COMM_WORLD);
110 }
111 MPI_Finalize();
112 }

```

Listing 2.5: An example of Matrix multiplication using MPI. (Barney, 2016*b*)

Collectives Communication

Collective Communication involves all processes in a process group. There are three types of Collective Communication routines:

Synchronization: These routines are used for process to wait until all members of the group have reached the synchronization point.

Data movement: These type of routines are used to broadcast data to all processes, as well as scatter, gather or reduction of data to and from other processes.

Collective Computation: These routines are use for one process of a communicator group to collect data from other members and perform operation on those.

(Barney, 2016*b*)

Environment management

MPI provides routines for managing execution environment to initialise and terminating MPI environment. The MPI environment is used to determine how many processes are participating in a computation and their rank within the group(Gropp et al., 1996). Environment management routines include

querying rank identity as well as some other execution time related operations(Barney, 2016*b*).

Derived Data-types

MPI predefines its primitive data types to achieve portability. This is because MPI supports heterogeneous environments(i.e different nodes) where the primitive types might have different representation(Mpi-forum.org, 1995). Primitive data are contiguous. MPI also provides mechanism for user defined data structures based on sequence of these MPI primitive data types. Such data types are known as Derived data types and they are non-contiguous(Barney, 2016*b*). An example of MPI derived datatype can be seen in Listing 3.7 which is an implementation parallel GA chromosome data structure.

MPI advantages

- MPI provides a powerful, efficient and portable way to express parallel programs
- MPI is only Message-passing library that supports a standard, thus supported by almost any HPC platform
- There is little or no need for source code modification to port an application to a different platform that supports MPI standard
- Vendors can exploit native hardware features to improve performance on their own MPI implementation
- Popular MPI implementations are available both vendor and public domain

(Barney, 2016*b*)

2.2.3 Hybrid Model

MPI between nodes and Shared memory programming inside of each SMP(Symmetric Multiprocessing) node.

2.3 Beowulf Clusters

A Cluster is a independent group of computer nodes combined into an unified system through network and software systems that allows data to move between nodes(Gropp et al., 2003).

There are two main usage of Cluster computing. One is performance and the other is fault-tolerance. Cluster computers can provide high performance through parallel computation. Such parallelism is achieve by coordinating many processors for single problem. Cluster computers can be used to address three main computational constraints: Realtime-constraints, Throughput and Memory limitation(Gropp et al., 2003). The Cluster computing is dominating the HPC worldwide. The most recent[November, 2016] list of top performing Supercomputers shows that 86.4% of the Top 500 Supercomputers are Clusters based (Top500.org, 2017).

The Beowulf project defines Beowulf as scaleable performance clusters based on commodity hardware, on a private system network with open source software infrastructure(Beowulf.org, 2016). The main purpose of a Beowulf cluster is to achieve performance through parallel computations. This is accomplished by running programs across a number of nodes simultaneously and they may need to coordinate during program execution(Gropp et al., 2003).

In 1994, first Beowulf cluster project was started as a experimental platform for parallel computing by Thomas Sterling, Donald Becker and their team at the NASA Goddard Space Flight center. Their aim was to build a low cost HPC(High Performance Computing) system for scientific computation with commodity hardware and open source software bundle. (Gropp et al., 2003)

2.3.1 Architecture

A Beowulf Cluster consists of one master node and one or more compute nodes connected by a System Area Network. Figure in 2.10 shows a basic Beowulf architecture. The compute nodes are connected through a private network. Master node can be connected to the external network if required (Gropp et al., 2003). Beowulf differs from Cluster of Workstation (COW) for the fact that the entire system behave like a single machine. Only master node provides user interface. The compute nodes can be diskless or may contain small local storage. Jobs are initiated, scheduled and assigned by the master node.

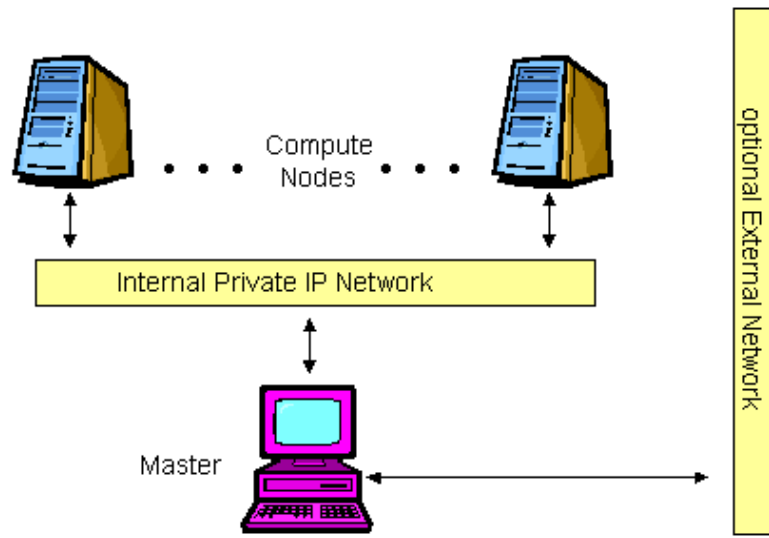


Figure 2.10: A basic Beowulf architecture. (Atom.me.gatech.edu, 2016)

2.3.2 Hardware

The main idea behind Beowulf project was to build HPC clusters using commodity hardware (Gropp et al., 2003). The commodity hardware are those that are available Commercially-Off-The-Shelf (COTS) and not build for any specific purpose. These could be from PC-compatible hardware or rack-mounted servers to some embedded systems. The commodity status refers to the ability of obtaining such hardware on the open market and leveraging cheaper large

scale productions. The biggest advantage of a Beowulf clusters is that it can be built with hardware produced by different vendors(Thiruvathukal, 2005)

2.3.3 Networking

Beowulf clusters are group of individual machines that are meant to be interconnected to perform a task or number of tasks in parallel. In order to coordinate and complete execution efficiently the nodes in a cluster must be able to communicate with one another. Therefore, networks are very important components of a Beowulf cluster(Gropp et al., 2003).

A Beowulf cluster with small number of nodes commodity networking technologies like Gigabit Ethernet are widely used. However, as the number of nodes increases network latency becomes a bottleneck for overall cluster performance. To overcome this, different network topology such hypercube topology and high-end networking hardware are used(Thiruvathukal, 2005).

2.3.4 Software Infrastructure

Operating Systems Linux is widely used on Beowulf clusters because it is open source, reliable and cost free. Another alternative is FreeBSD which is also open source. Open source nature of these Operating Systems are attractive to Beowulf community as it allows the kernel to be customised for specific needs and setup. (Gropp et al., 2003)

Middleware Coordination and communication between the processing nodes in cluster is key requirement. Beowulf utilises Message-passing model of parallel architecture. Both MPI and PVM(Parallel Virtual Machine) API can be deployed in a Beowulf systems. However, having a standard API and wider portability MPI become a popular choice for parallel programming in Beowulf. (Gropp et al., 2003)

Job control The Operating System only manages local jobs by itself. In distributed memory model additional job controller software is required. This job control program responsible for system partitioning, queuing jobs, how jobs start and run on remote nodes(Gropp et al., 2003). Example of such parallel job control systems are PBS(Portable Batch System), Hydra and MPD.

2.4 Simple Genetic Algorithm (SGA)

In his renowned book on Genetic Algorithm David E. Goldberg (Goldberg, 1989) describes Genetic Algorithms(GA) as search and optimisation technique which is based on the theory of natural selections and natural genetics. These are general search and optimisation algorithms that use the theories behind the evolution as a tool to find optimal or near optimal solution to difficult problems. This involves evolving a population of possible solutions to the particular problem by using operations inspired by natural genetic variation and natural selection (Murphy, 2003). It simulates the "survival of the fittest" among individual solution over consecutive generation for solving a problem. In 1960's Genetic Algorithms were pioneered by John Holland along with his colleagues and students at University of Michigan(Murphy, 2003).

Optimisation using GA

In science and Engineering there exist many problems that have solution space too large or sometimes even an infinite set. In those case it is impossible to check each individual solution and try to find the best one. One approach to finding a solution of such problems is limiting the number of possibilities and range with a given step size for lookup. The set of possible solution are known as 'search space'.

Each possible solution can be marked by its value of fitness for the problem. Using this fitness value, the algorithm determines which set of solutions are going to forward to produce new solutions. GA attempt to find the best solution based on its fitness value to a given problem. This is achieved by using

a combination of *exploitation* and *exploration*. When the algorithm has found a number of good possible solutions it exploits them by combining different parts of each solution to form a new candidate solutions. This is known as *crossover*. GA also do some experiment by creating new candidate solutions by randomly changing parts of old solutions which is known as *mutation*. (Murphy, 2003).

In his Master thesis Roderick Murphy described GA as 'weak' optimisation of searching procedure as they do not use any domain specific knowledge to do so(Murphy, 2003). Instead GA utilise some 'random' choice to guide its search by exploiting historical information to direct the search into a region of optima.

GA terminology

Population: Population is a subset of all possible solutions to the given problem.

Chromosome: A chromosome is one individual solution to the problem.

Gene: A gene is one element position of a chromosome.

Allele: Allele is the particular value of a gene in a chromosome.

Locus: The position of a gene on the chromosome.

Genotype: Genotype is the population in the computation space.

Phenotype: Phenotype is the population in the real world solution space. (Murphy, 2003)

Computer Implementation

To utilise GA one must encode solutions to a problem in a structure that can be stored in the computer. A solution to a problem is represented as string known as *chromosome*(strings) that consist of a combination of several genes(simple representation are binary 1's & 0's). If we consider a multi-variable problem, a gene can be considered as the bits that encode a particular parameter and its allowable value. A set of solutions to the problem is represented by a group of chromosomes referred to as a *population*. During each iteration of the algorithm the chromosomes in the population will undergo one or more genetic operations such as *crossover* and *mutation*. The result of the genetic operations will become the next generations of the population. This process continues until a solution is found or a certain termination condition is met. (Murphy, 2003)

Genetic Operators

Genetic Operators are responsible for altering the genetic composition of offspring. There three operators:

- Selection
- Crossover
- Mutation

Selection

Selection operator is used for selecting candidate solutions for reproduction. The probability of selection should be proportional to the fitness value of the chromosome itself. The selection process is very important for maintaining good diversity and preventing premature convergence. There are several methods of how to select the best chromosomes.

Roulette wheel selection: In roulette wheel selection parents are selected according to their fitness. The better the fitness value the higher the chances to be selected.

Rank selection: Rank selection first ranks the population and then every chromosome receives fitness from this ranking. In population with N chromosome, the worst chromosome will have rank 1, second worst will have 2 and the best one will have N.

Tournament selection: In Tournament selection K number of individuals are taken from the population then select the best out of these using their fitness value. The process is repeated for selecting next parent. (Tutorials-point.com, 2017)

Crossover

The primary purpose of crossover operator is to get genetic material from the previous generation to the next generation. This is done by selecting two parents and then exchanging some parts of their genes. This produces two new offsprings. There are different ways of exchanging genes.

Single-point crossover This is the simplest form of crossover in which a random crossover point is selected and the bits from this point get swapped with between two parents to get new offsprings. Figure 2.11 demonstrate the the single-point crossover.

Multipoint crossover In mulit-point crossover a number of crossover points are chosen and then bits segments in every second group are swapped between two parents to produce new offsprings. Figure 2.12 shows multi-point crossover.

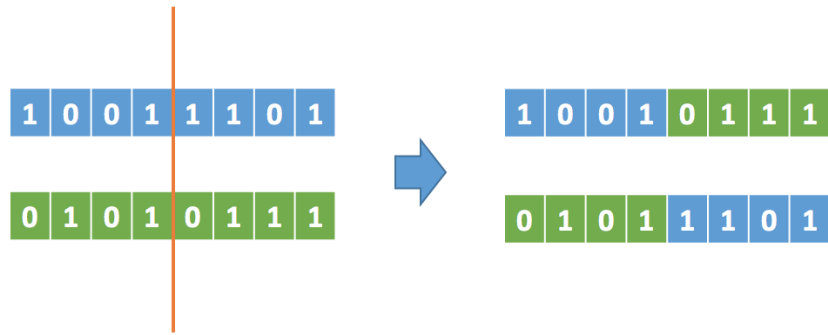


Figure 2.11: Single-point crossover

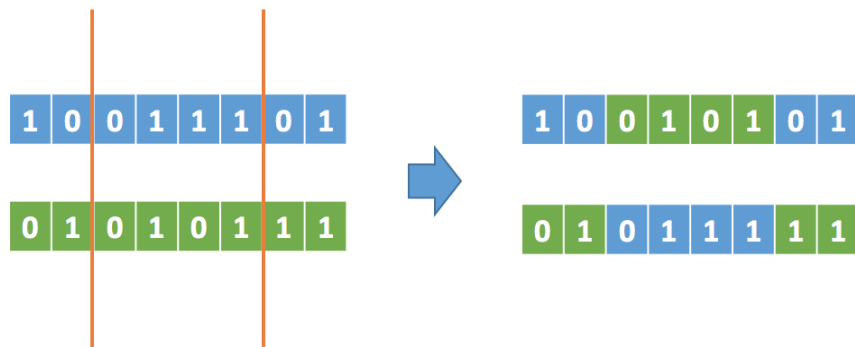


Figure 2.12: Multi-point crossover

Uniform crossover Uniform crossover is a variation of multi-point crossover. In this method random decision is made whether a bit groups will be swapped or not between two parents to produce new offsprings. Figure 2.13 shows uniform crossover.

(Murphy, 2003)

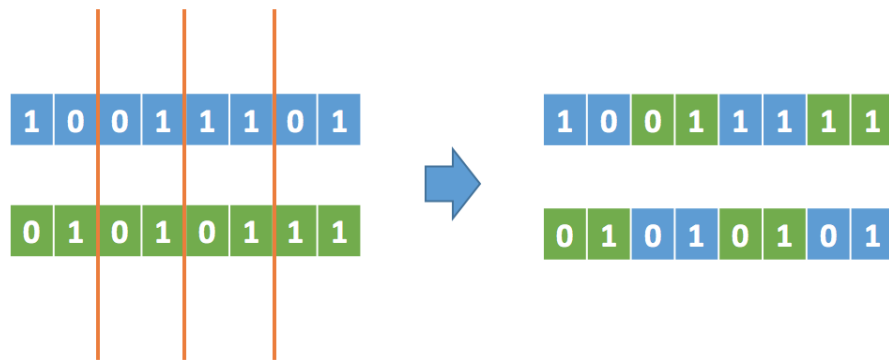


Figure 2.13: Uniform crossover

Mutation

Mutation operator create new offspring by making small tweak in chromosome. It can help creating chromosomes that would not otherwise be formed by applying selection and crossover operators alone. Mutation can allows GA to explore the search space and keep it from getting trapped in a local optimal solutions. There are several methods of mutation such as bit flipping, random resetting, swap mutation and inversion mutation(Tutorialspoint.com, 2017). Figure 2.14 shows bit flipping and Swap mutation.

Simple Genetic Algorithm

1. **Start:** Generate random population of n chromosomes (suitable solutions for the problem)
2. **Fitness:** Evaluate the fitness $f(x)$ of each chromosome x in the population
3. **New population:** Create a new population by repeating following steps until the new population is complete

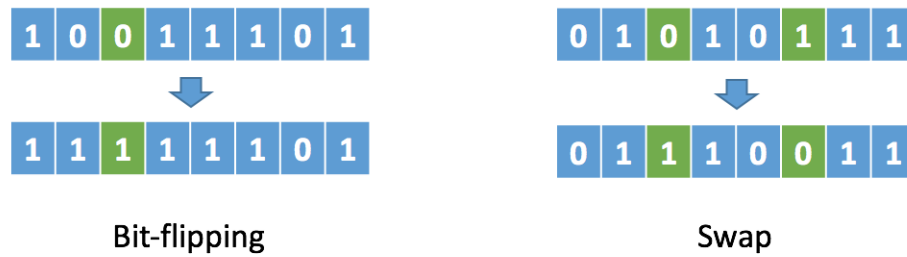


Figure 2.14: Mutation

- (a) **Selection:** Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)
 - (b) **Crossover:** With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
 - (c) **Mutation:** With a mutation probability mutate new offspring at each locus (position in chromosome).
 - (d) **Accepting:** Place new offspring in a new population
4. **Replace:** Use new generated population for a further run of algorithm
 5. **Test:** If the end condition is satisfied, stop, and return the best solution in current population
 6. **Loop:** Go to step 2

(Obitko, 1998)

Application of GA

Genetics Algorithms are very effective way of quickly finding a reasonable solution to a complex problem. GAs are most effective in search space for which little is known. Apart from general optimisation problems, Genetic Algorithms are successfully deployed on diverse range of other fields. These include tasks optimising, dynamic programming, machine learning, economics, immune systems, ecology, population genetics and social systems. (Murphy, 2003)

A more recent development is the use of Evolutionary Computation (EC) to evolve the control structures for humanoid robotics (Eaton, 2015). Writing software in a traditional manner to controls the very large number of articulation points is beyond the capability of a human programmer (Eaton, 2015). Techniques that leverage GAs have been demonstrated to effectively equip humanoid robotic platforms with abilities to simulate the mechanics of incredibly complex human movement such as kicking a soccer ball.

Advantages

Genetic Algorithms have many advantages which include:

- Does not require any derivative information (which may not be available for many real-world problems).
- Is faster and more efficient as compared to the traditional methods.
- Has very good parallel capabilities.
- Optimizes both continuous and discrete functions and also multi-objective problems.
- Provides a list of "good" solutions and not just a single solution.
- Always gets an answer to the problem, which gets better over the time.
- Useful when the search space is very large and there are a large number of parameters involved.

Limitations

There are number limitation with Genetic Algorithms that we must consider. These include:

- GAs are not suited for all problems, especially problems which are simple and for which derivative information is available.
- Fitness value is calculated repeatedly which might be computationally expensive for some problems.
- Being stochastic, there are no guarantees on the optimality or the quality of the solution.
- If not implemented properly, the GA may not converge to the optimal solution

(Pit, 1995)

2.5 Parallel GA

Once applied successfully GA's are generally able to find good solutions to a problem in reasonable amounts of time. But when the problem in hand is bigger and harder, there is an increase in time required to find adequate solutions for it. Because of this there have been many efforts to make GA's faster by parallel implementation (Cantú-Paz, 1998).

There are two possible ways of parallelising GA's.

- Data parallelism
- Task parallelism

Data parallelism Data parallelism is dividing up data in a program into multiple subsets and executing same procedure(s) on them. With data parallelism, parallelism grows with data.

Task parallelism Task parallelism involves concurrent execution of different operations on different processor in parallel on same data. Usually suitable for executing several procedures on same data where no dependencies between tasks. (Konfrst, 2004)

A hybrid parallelism by mixing both data and task parallelism is also common. Generally in GAs computation time increases with data therefore data parallelism is usually implemented for parallelising.

2.5.1 Classification of Parallel GAs

Many models of parallel GA implementation exist. These options are depend on answer to the questions such as how fitness are evaluated, how mutation and crossover operations are applied, if a single or multiple population are used, how individuals are exchanged, if selection applied globally or locally.

Luque et al (Luque et al., 2005) and Cantú-Paz(Cantú-Paz, 1998) classified parallel GA model into following categories:

- Master-slave model - Global population
- Distributed model - Coarse grained
- Cellular model - Fine grained

2.5.2 Master-slave Model - Global population

Similar to Simple GA, master-slave model consists a single global population. Usually the evaluation of fitness function is distributed among multiple processors. The main loop of the GA is executed in a master process. In this model of parallel GA selection and crossover consider the entire global population. One distinctive advantage of masters-slave model of parallel GA is that it does not alter the behaviour of the algorithm other than speeding up. This model is efficient as the objective function evaluation becomes more expensive to compute. Figure 2.15 shows the Master-slave model's message exchange

between master and slave processes where slave process evaluate a part of the global population fitness value.

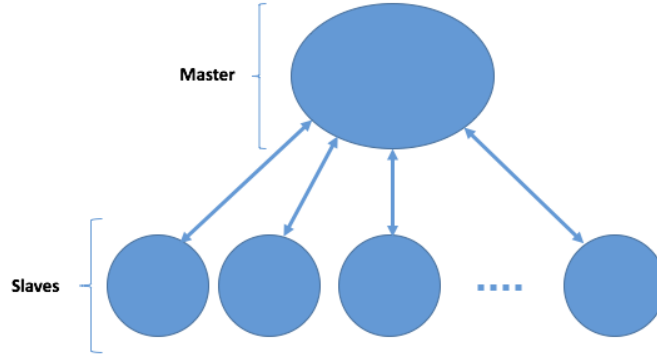


Figure 2.15: Master-Slave PGA model.

2.5.3 Distributed Model - Coarse grained

In distributed model initial population are divided into relatively isolated sub-populations. These sub-population are called *island*. This is also known as multi-population or multi-demes GAs. These islands evolve independently over time but allows individuals to be migrated to another island. The genetic operations selection, mutation and crossover are performed within the subpopulation of each island. Figure 2.16 shows the distributed model of PGA where each node independently performs all GA operation on it's local population for a generation and then the best individual is migrated to the next neighbour.

2.5.4 Cellular Model - Fine grained

Cellular or fine-grained parallel GAs works on one single population where each processing unit holds only one or two individuals hence its called fine-grained. This model structures the global population into neighbourhoods where individuals can only interact with their neighbours only. Which limits an individual only to compete and mate with its neighbouring individuals.

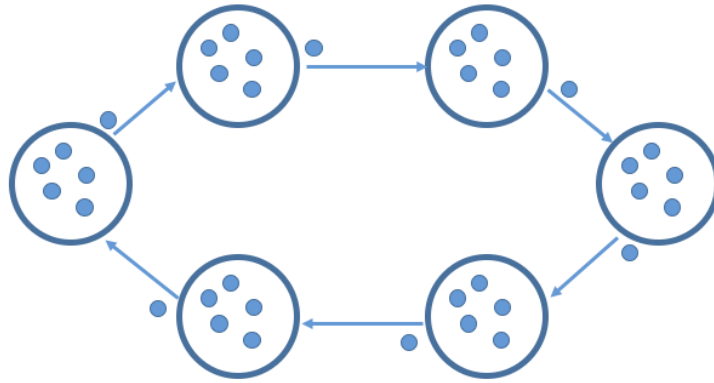


Figure 2.16: Distributed PGA model.

Therefore a good solution can arise in different area of the population distribution and can spread throughout the whole population overtime. Cellular GAs are originally designed for massively parallel machines. Figure 2.17 shows the cellular PGA model where each node holds one or two individuals and interact with all immediate neighbours for selecting other individuals to perform local GA operation to reproduce offsprings.

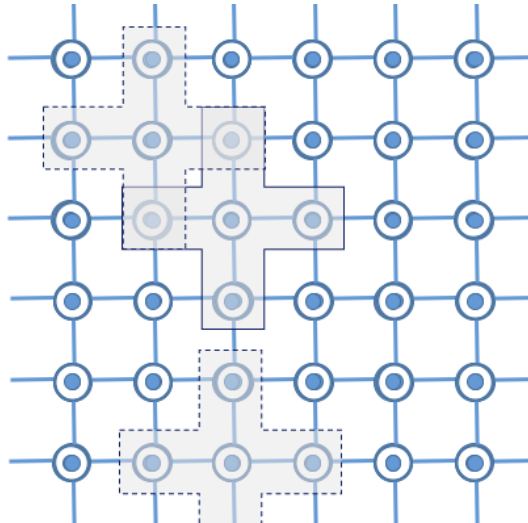


Figure 2.17: Cellular PGA model.

Chapter 3

Design and Implementation

3.1 Introduction

In this chapter all aspects of design and implementation that were done for this project are discussed. A parallel infrastructure was essential for this project for implementing the PGA using Message Passing Interface. A considerable time and effort were made to implement and configure the Beowulf cluster to facilitate the parallel infrastructure for the project. Our original idea was to use an existing Beowulf system in University's laboratory. Later it was discovered that, the most of the hardware of that Beowulf system were dismantled and no longer suitable for this project. So, the the most obvious alternative was to build a Beowulf cluster for this project. At first, a virtual Beowulf system was simulated on a dual core MacBook. Then an actual hardware based Beowulf system was implemented using multiple old PC's. After all, this is the noble idea of a Beowulf system to have it built using hardware laying around that have very little or no use at all.

Another important part of implementing a PGA, we needed to identify a suitable existing SGA implementation. Even though the Genetic Algorithm was not the centre theme of this project, it was very important to do a considerable background studies in GA to understand the underlying concepts well. It was also important to understand and identify different techniques of parallelising

GAs. Much efforts and time were put into understanding two different implementation of SGA and their suitability on parallelising using MPI in details for this project which are discussed in this chapter.

With PGA implementation, we were faced with many issues and challenges. Some of these issues were identified during PGA implementation and some were identified during empirical studies. Which resulted in implementing 3 versions of PGA.

A lot efforts were made to create prototypes using python scripts to gather experiments data and use gnuplot to visualise the results to compare and contrast to validate the hypotheses relating to PGA performance.

3.2 Setting up the Beowulf Cluster

3.2.1 Cluster components

Computer cluster is made up of various hardware and software components with complex interactions between different components. Figure 3.1 depicts the various components that form a cluster.

A Beowulf cluster contains a set of cluster nodes. A node can be a server with a specific role within the cluster or it can be a compute node. All of the nodes are connected using a dedicated Local Area Network(LAN) or System Area Network(SAN). The complexity of networking topology is depends on directly to number of nodes in the cluster.

As shown in Figure 3.1 to setup a cluster we need following hardware and software components:

Cluster Nodes Cluster nodes are stand-alone computers with one or more CPUs, own Memory unit, and Network Interface Card(NIC). A node may contain it's own local secondary storage or could be configured as disk-less workstation. For this implementation nodes were configured with their own

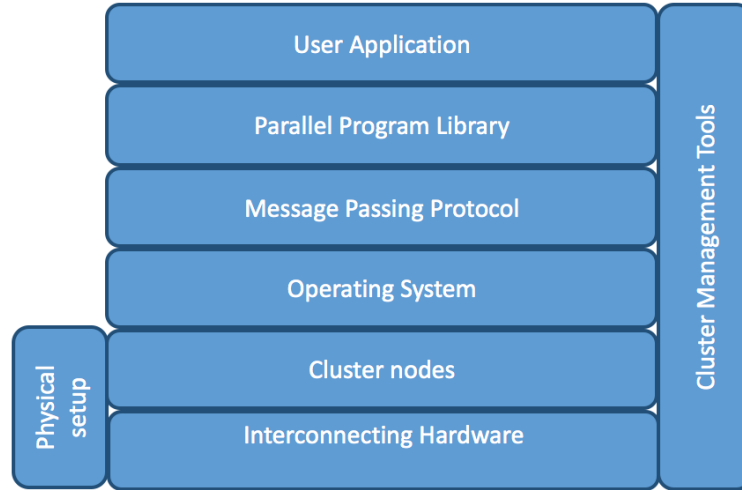


Figure 3.1: Layers in a Cluster

local storage to reduce network overhead for accessing boot-image and run-time libraries during boot up and program execution. However, Network File Sharing(NFS) was implemented to streamline program execution environments across the Cluster. The details of nodes configurations are discussed later in this chapter for both virtual and physical implementation.

Interconnecting Hardware This is networking components of the cluster that facilitates nodes communicating with one another. Since communication is the key part of cluster computing this is very important component that can define the overall performance. For this project a simple ethernet network was configured for both virtual and physical implementation. For the physical setup, Thompson Broadband Router (Model: TWG870) was used. This router has 4 Gigabit ports that facilitated to setup a Gigabit LAN for the cluster-only network.

Operating System Linux is widely used on Beowulf cluster because of its hardware support, performance, its free and open source kernel. For this project Ubuntu Server 16.04 was chosen.

Message Passing Protocol There are two typical approaches to communication between cluster nodes, one is Parallel Virtual Machine(PVM) and the other is Message Passing Interface (MPI). However, MPI has now emerged as the de facto standard for message passing on computer clusters. The objective of this project is parallelising SGA using MPI libraries therefore, MPI was chosen message passing protocol for obvious reason.

Parallel Program Library MPI has many implementation. There are two free and popular MPI libraries available, one is OpenMPI and another is MPICH. For this project MPICH 3.2 is installed on each nodes.

Cluster management tools

3.2.2 Topological design

Figure 3.2 shows a simple Beowulf cluster setup that was adopted for this project implementation. For simplicity the implemented Beowulf contains a Master node and 3 to 4 compute nodes which is extendable.

Beowulf cluster was simulated in virtual environment using Oracle VirtualBox and then actual hardware implementation was done using commodity physical hardware. In both cases the software suits and configuration was identical with minor changes.

3.2.3 Virtual nodes

Table 3.1 shows the node configuration when using simulated hardware using VirtualBox on Macbook Pro(2.6GHz Intel core i5, 16 GB RAM, 256GB SSD) host computer.

3.2.4 Physical nodes

Table 3.2 shows the node configuration of using physical components.

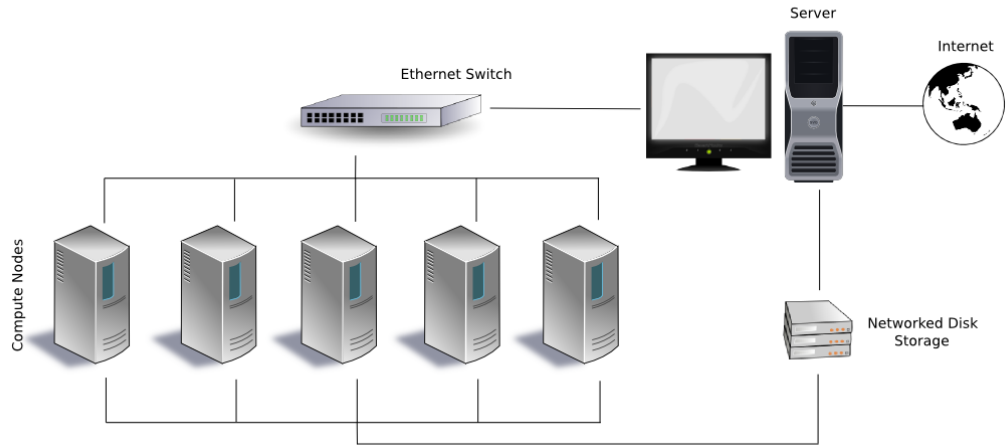


Figure 3.2: A typical Beowulf setup

Table 3.1: Virtual nodes configuration

Hostname	Role	IPv4 address	Hardware Config
master	Master	192.168.56.10/24	Single core, 2GB RAM, 10 GB HDD
node1	Slave	192.168.56.11/24	Single core, 2GB RAM, 10 GB HDD

Table 3.2: Physical nodes configuration

Hostname	Role	IPv4 address	Hardware Config
padma	Master	172.16.0.1/24	Dell Dual-core, 6GB RAM, 160 GB HDD
meghna	Slave	172.16.0.15/24	HP Dual-core, 2GB RAM, 160 GB HDD
jamuna	Slave node	172.16.0.17/24	HP Dual-core, 2GB RAM, 160 GB HDD

3.2.5 Setup & Configuration

Setup steps

- Install Operating Systems(OS) for each nodes

- Configure network settings
- Create MPI user
- Configure Network File Share
- Configure SSH password-less login
- Install MPI libraries, compiler and build tools
- Administrative tasks

Installing Operating System

Popular Linux distribution Ubuntu Server 16.04 was chosen as OS for all cluster nodes. It was installed with default settings and without any GUI or window manager. A desktop environment was not needed as it would have reduced the performance of our Beowulf system. Therefore our installation of Ubuntu Linux was a minimal system and OpenSSH server.

After a successful installation of the operating system, we needed to set up the network and, of course, the shared file system for the compute nodes. For the shared file system, we installed the NFS server on the first node, which operates as the head node. Then, we exported the locations for the shared file system from there.

Network configuration - Master node

Identify network interface card and its name

```
1 $ip link show
```

And from the output shown as below we identify the NIC names that we want to configure in following steps

```
1 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
   state UNKNOWN mode DEFAULT group default qlen 1
2  link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

```

3 2: enp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu
      1500 qdisc pfifo_fast state UP mode DEFAULT group default
      qlen 1000
4      link/ether ec:08:6b:04:62:df brd ff:ff:ff:ff:ff:ff
5 3: enp0s25: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu
      1500 qdisc pfifo_fast state UP mode DEFAULT group default
      qlen 1000
6      link/ether 00:19:d1:73:38:60 brd ff:ff:ff:ff:ff:ff

```

To configure the network configuration file `/etc/network/interfaces` need to be edited as follows:

```

1 auto lo
2 iface lo inet loopback
3
4 # The primary network interface(connected to public network)
5 auto enp0s25
6 iface enp0s25 inet static
7     address 192.168.0.14
8     netmask 255.255.255.0
9     gateway 192.168.0.1
10    dns-nameserver 89.101.160.5 8.8.8.8
11
12 # The cluster network interface
13 auto enp2s0
14 iface enp2s0 inet static
15     address 172.16.0.1
16     netmask 255.255.255.0

```

System's host file(`/etc/hosts`) needs to be edited for each node for easier host look up.

```

1 127.0.0.1 localhost
2 172.16.0.1 padma
3 172.16.0.2 meghna
4 172.16.0.3 jamuna

```

Then scp this file to other nodes and copy it with privileged user into /etc directory.

```
1 $scp /etc/hosts meghna:~/
2 $ssh meghna
3 $sudo cp hosts /etc/hosts
4 $scp /etc/hosts jamuna:~/
5 $ssh jamuna
6 $sudo cp hosts /etc/hosts
```

Create MPI user

Create a user for running MPI jobs. This user needs to be created with same user ID and group ID on each node.

```
1 $sudo adduser mpiuser --uid 999
```

Network File System configuration

Install and setup the Network File System

```
1 # Master node
2 padma$ sudo apt-get install nfs-kernel-server
3 padma$ sudo apt-get install nfs-common
4 # And compute nodes
5 meghna$ sudo apt-get install nfs-common
6 jamuna$ sudo apt-get install nfs-common
```

Next we need to share mpiuser home directory with all other nodes. To do this the file /etc/exports on the master node needs to be edited to add the share directory.

```
1 /home/mpiuser 172.16.0.0/24(rw,sync,no_subtree_check)
```

At this point NFS server needs to be restarted to make share directory available across the network.

```
1 master:~\$ sudo service nfs-kernel-server restart
2 sudo exportfs -a
```

Now we can check and confirm from compute node if the NFS shared directory can be mount.

```
1 showmount -e padma
```

This should show the export list from master node as below:

```
1 Export list for padma:
2 /home/mpiuser 172.16.0.0/24
```

To mount this shared directory from other nodes we can execute the command below from the shell.

```
1 meghna:~\$ sudo mount master:/home/mpiuser /home/mpiuser
2 jamuna:~\$ sudo mount master:/home/mpiuser /home/mpiuser
```

And to mount the share directory at system boot up without having to enter the command manually, the following entry is added to all compute nodes in the file `/etc/fstab`.

```
1 padma:/home/mpiuser /home/mpiuser nfs
```

Configure password less communication

All MPI nodes should be able to communicate with other nodes without having to provide any password. This is achieved by configuring password-less SSH between nodes.

If ssh is not installed with during the OS installation steps we could install it by running the following command on each nodes:

```
1 $sudo apt-get install ssh
```

Next step is to generate a SSH key for 'mpiuser' on all nodes. The SSH key is by default created in the user's home directory. In our case the 'mpiuser'

home directory which is located at `/home/mpiuser`. This is actually the same directory for all nodes i.e `/home/mpiuser` on the master node since the home directory is a NFS mount. So, the SSH key for the 'mpiuser' was generated on master node, and all other nodes will automatically have an SSH key in 'mpiuser' home directory.

To generate an SSH key for the MPI user on the master node (but any node should be fine).

```
1 \ $ su mpiuser
2 \ $ ssh-keygen
```

When asked for a passphrase, this needs to be left empty (hence password-less SSH).

When done, the 'mpiuser' should have same ssh-key on its home directory which is accessible from all nodes in the cluster. Now, the master node needs to be able to automatically login to the compute nodes. To enable this, the public SSH key of the master node needs to be added to the list of known hosts (this is usually a file `~/.ssh/authorized_keys`) of all compute nodes. All SSH key data is stored in one location: `/home/mpiuser/.ssh/` on the master node. So instead of having to copy master node's public SSH key to all compute nodes separately, we just have to copy it to master's own `authorized_keys` file. There is a command to push the public SSH key of the currently logged in user to another computer. To do that we run the following commands on the master node as user "mpiuser",

```
1 mpiuser@master:~\ $ ssh-copy-id localhost
```

Master node's own public SSH key should now be copied to `/home/mpiuser/.ssh/authorized_keys`. But since `/home/mpiuser/` (and everything under it) is shared with all nodes via NFS, all nodes should now have master's public SSH key in the list of known hosts. This means that we should now be able to login on the compute nodes from the master node securely without having to enter a password,

```
1 mpiuser@master:~ $ ssh meghna
2 mpiuser@meghna:~ $ echo $HOSTNAME
```

```
3 meghna
```

The user 'mpiuser' should be now be able to login on node1 via SSH. At this stage login to other nodes also be checked.

Install MPI libraries, compiler and build tools

On each node we needed to install C/C++ libraries and headers files along with required build tools. After that we installed MPICH 3.2 version of MPI library using Ubuntu's default source repository.

```
1 $sudo apt-get install build-essential
2 $sudo apt-get install mpich
3 $mpicc -v \# To check MPICH version OR
4 $mpiexec --version
```

Configuring MPI Process Manager

A process manager is needed for MPI programs are to spawn and manage parallel cluster. This process manager is an external entity. In MPICH implementation of MPI library these process managers communicate with MPI processes using a predefined interface called as PMI (Process Management Interface). The process manager included with MPICH 3.2 installation is called Hydra. In order to setup Hydra, we need to create one file on the master node. This file contains all the host names of the compute nodes. This file can be put in anywhere, but for simplicity it is created in the the MPI user's home directory. This file only needs to be present on the node that will be used to start jobs on the cluster, usually the master node. Hydra uses this file to spawn process in the cluster in round robin manner. Since the MPI home directory is shared among all nodes, all nodes will have the hosts file. This file is created using following command where the file is named *hosts*:

```
1 $cd ~
2 $touch hosts
```

For this Beowulf setup the following entries are added to this *hosts* file:

```
1 padma:2
2 meghna:2
3 jamuna:2
```

3.3 Object-Oriented GA

There are many different implementation of Simple Genetic Algorithms(SGA) available. Initially the SGA implementation selected for this project for parallelising was an implementation that closely followed the suggestions of David E. Goldberg's book "*Genetic Algorithms in Search, Optimization, & Machine learning*" (Goldberg, 1989). It was implemented using C++ and took an Object Oriented approach as an abstraction mechanism. There are 37 user modifiable parameters defined to fine tune the behaviour the SGA. It was designed to allow users to change the chromosome structure(haploid or diploid), fixed or variable length chromosome, 3 different mechanism of diploid crossovers, 7 different types of encoding schemes. User could also select fitness function from 15 different implementation(even though only a few of them are actually implemented). It also allows user to select from two different models of elitism, 6 different mechanism of selections. Furthermore, it contains an implementation of Linear Congruential method for generating pseudo random numbers which is used to randomly generate probabilities for mutation and crossover.

This codebase contains over 4,500 lines of code. About 2 weeks time was spent on compiling and running this SGA to understand different parameters and process flow of the algorithm it had implemented. Which was well beyond the planned schedule for this project that is shown in Project plans in Appendix A.

After 2 weeks of studying and experimenting with this SGA implementation, I managed to summarise an analysis of this implementation. this code exhibit many OOD(Object Oriented Development) pitfalls that was common in early Object Oriented Development days. Inheritance was used as a means

of implementation reuse without any semantic coherence. Classes are too big and contains too many responsibilities which is against "Single Responsibility principle" of OOD. In many cases, classes violates Liskov's Substitute Principle as well where a derived classes cannot be replaced for the base classes. Figure 3.3 shows the class diagram of this implementation of SGA code using C++.

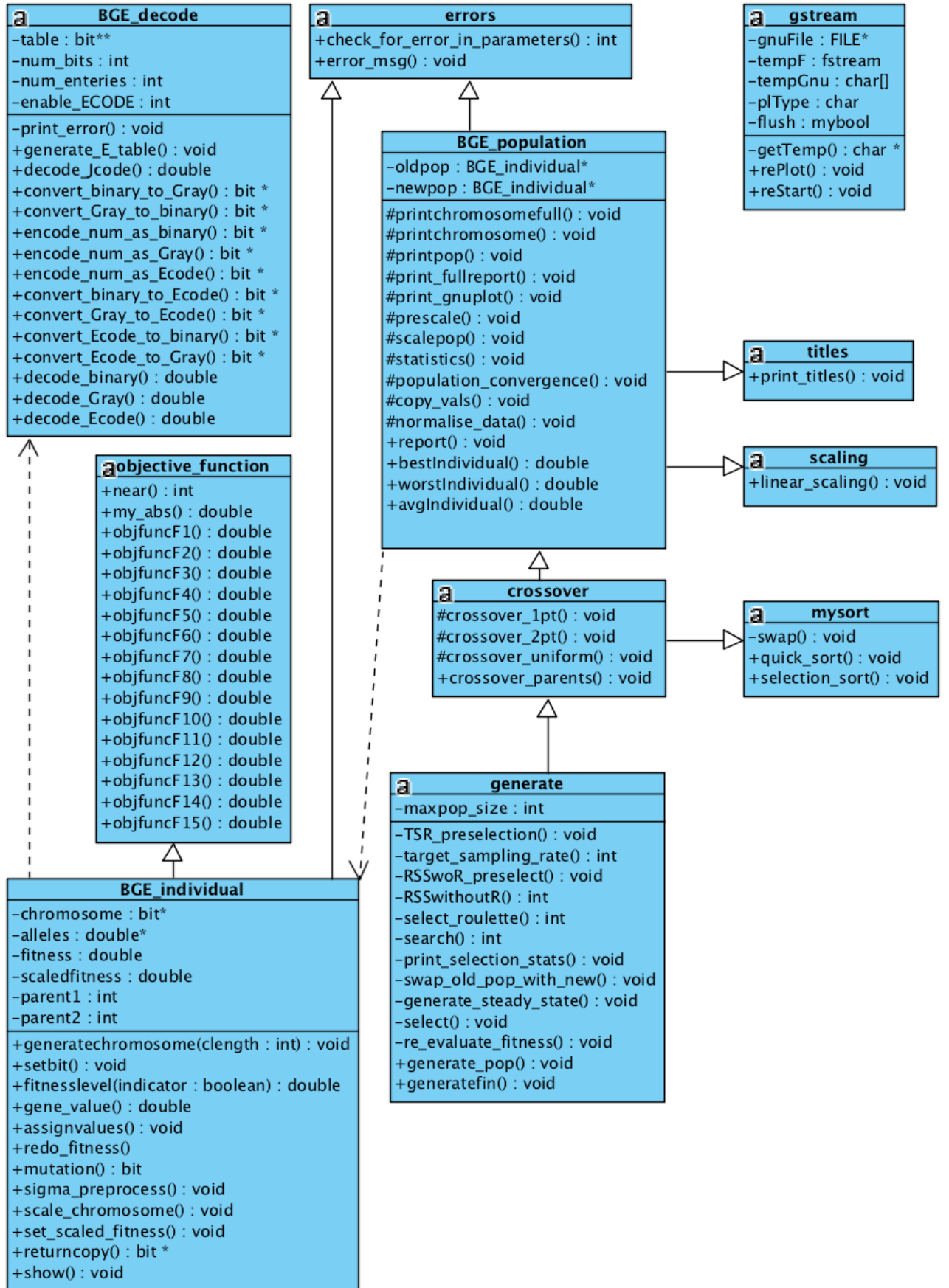


Figure 3.3: Class diagram of Object-oriented SGA

3.4 Issues with MPI support for Object-oriented paradigm

The major drawback discovered with object-oriented implementation of GAs was the MPI's lack of support for object-oriented languages. The MPICH 3.2 version of MPI implementation were used this project. With further research on this topic it was discovered none of the major MPI implementation (OpenMPI,) truly supported the object-oriented C++ other than just the compiler support.

In a conference article titled *"Object Oriented MPI (OOMPI): a class library for the Message Passing Interface"* Squyres et al discussed MPI's lack of support of C++ class libraries and binding. In that paper they suggested an Object Oriented MPI specification (Squyres et al., 1996). However, this effort have been abandoned due to the lack of support from MPI community. After doing further research on this topic in Internet discussion forums and blog posts, it is found that there are ways of doing OO MPI using C++ by using third-party libraries Boost::mpi and Boost::serialization. Boost itself is a set of libraries for the C++ programming language that provide support for tasks and structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing. Boost is a free to use library licensed under its own free, open-source license, known as the Boost Software License. An MPI program using the Boost::mpi library must initialise an mpi::environment object. This object initialises the MPI environment and enables communication among the processes. This object is initialised with the program arguments in the main program. The creation of this object initialises the MPI environment, and its destruction terminates it. The Boost::mpi requires objects to be serialised so that they are transmissible across processes. The Boost::serialization is a library for serialising C++ objects for sending/receiving data across network. To serialise a class one needs to define a template function `serialize()` in the class.

There is a little draw back with using Boost::mpi library. The Boost::mpi only implemented MPI1.1 which does not support many useful routines of MPI

standards. Some people suggested to use that for regular programs. And for advanced programs to use mainstream MPI-3 libraries with Boost::Serialization for serialising C++ objects. Considering the scope of this project this options is not taken.

3.5 Open Source Procedural GA

A procedural implementation of GA based on Zbigniew Michalewicz's book *"Genetic Algorithms + Data Structures = Evolution Programs"* (Michalewicz, 1996) available under GNU public license. The original version of this implementation was done by Dennis Cormier and Sita Raghavan. From Department of Scientific Computing at Florida State University John Burkardt made a C++ version of source code available on their department's website (Burkardt, 2017). The slightly refactored version of this source code without changing the actual algorithms is included in Appendix C. Where the original code is put into 4 separate source files shown in C.1, C.2, C.3 & C.7.

This is a simple genetic algorithm implementation where the evaluation function takes positive values only and the fitness of an individual is the same as the value of the objective function.

Each generation involves selecting the best members, performing crossover and mutation and then evaluating the resulting population, until the terminating condition is satisfied.

Behaviour of this program is modified by defining parameters listed in Listing 3.1 where POPSIZE is population size, MAXGEN is the number of generation to run, NVARs is the number of variables or genome in a chromosome, PXOVER and PMUTATION are the probabilities of crossover and mutation respectively. However this program expects the permissible value ranges for each variable to be read from a text file.

```
1 # define POPSIZE 50
2 # define MAXGENS 1000
3 # define NVARs 3
4 # define PXOVER 0.8
```

```
5 # define PMUTATION 0.15
```

Listing 3.1: Parameter definitions.

Listing 3.2 shows the data structure used for defining GENOTYPE which is a member of a population. The struct member .gene is a string of variables, .fitness is the fitness value .upper is the variable upper bounds, .lower: the variable lower bounds, .rfitness is the relative fitness and .cfitness is the cumulative fitness value. Relative and cumulative fitness values are used for elitist in a given generation.

```
1 struct genotype
2 {
3     double gene[NVARS];
4     double fitness;
5     double upper[NVARS];
6     double lower[NVARS];
7     double rfitness;
8     double cfitness;
9 };
```

Listing 3.2: GENOTYPE: Data structure of population member.

In Listing 3.3 shows the for loop that is repeated for each generation cycle and perform the GA operations on current population. The select() procedure on line 3 perform a roulette wheel selection from current population. The elitist() procedure in line 9 performing an elitist operation which finds the best and the worst member of the current population. If the current best is better than last generation's best then replace it with current best otherwise replace the current worst with last generation best.

```
1 for ( generation = 0; generation < MAXGENS; generation++ )
2 {
3     selector ( seed );
4     crossover ( seed );
5     mutate ( seed );
6     report ( generation );
7     evaluate ( );
8     elitist ( );
```



```
9 }
```

Listing 3.3: Main loop controlling the GA operations.

Fitness function

The fitness function provided with this code base is very simple one. This function was implemented to handle only three variables chromosome. It performs a very simple calculation of $fitness = x_1^2 - x_1 * x_2 + x_3$. As can be seen in source code provided in Appendix C, the original codebase is slightly modified to facilitate introducing other fitness function. The modification can be seen in Listing 3.4.

```
1 void evaluate ( int my_start , int row_count )
2 {
3     switch(FITNESS_F) {
4         case F1:
5             return f1(my_start , row_count);
6             break;
7         case F8:
8             return f8(my_start , row_count);
9             break;
10        default:
11            break;
12    }
13 }
```

Listing 3.4: Fitness function - Implementation of Griewank function

And for f8() the Griewank function was implemented as shown in Listing 3.5. The Griewank function is defined as below:

$$fitness = 1 + \sum_{i=1}^{i=n} (x_i^2/4000) - \prod_{i=1}^{i=n} \cos(x_i/\sqrt{i})$$

```
1 void f8(int my_start , int row_count)
2 {
3     int member;
4     int i;
5     double x[NVARS+1];
6     double part1 , part2;
```

```

7
8  int ends = my_start + row_count;
9
10 for ( member = my_start; member < ends; member++ )
11 {
12     for ( i = 0; i < NVAR; i++ )
13     {
14         x[i+1] = population[member].gene[i];
15     }
16
17     part1 = 0.0;
18     for(i = 1; i < NVAR; i++)
19     {
20         part1 += pow(x[i], 2) / 4000;
21     }
22
23     part2 = 1.0;
24     for(i = 1; i < NVAR; i++)
25     {
26         part2 *= cos( x[i] / sqrt(i) );
27     }
28
29     population[member].fitness = 1.0 + part1 - part2;
30 }
31 return;
32 }

```

Listing 3.5: Fitness function - Implementation of Griewank function

3.6 Parallelising Procedural GA using MPI

3.6.1 Parallel GA model

The master-slave model of parallel GA is adopted for this implementation. The operation most commonly parallelised in master-slave model is the evaluation of the fitness function as in general this requires only individuals values not the entire population so that the communication is minimum between nodes.

Parallelising the evaluation of fitness function is done by assigning a part of population to each node available in a cluster. Message passing occurs when slave nodes receive the subset of global population and returns evaluated fitness values to the master.

Figure 3.4 represent the flowchart of implemented Parallel GA. Dotted line shows Message passing communication between Master and Slaves. This flowchart shows the diagram of implemented parallel GA using the procedural SGA discussed earlier in this section. As we can see from this diagram Master process responsible for initialising first generation of population, evaluate and report on them. The next task for the master process is to dividing up population among available slave processes.

The following subsections explain various MPI directives implemented.

3.6.2 Initialising MPI environment

As with all MPI program the message-passing environment needs to be initialised prior to calling any message-passing procedure. This is done by calling `MPI_Init()` procedure on line 8 shown in the code fragments on Listing 3.6. MPI uses objects known as communicators and groups to define which collection of processes may communicate with each other. `MPI_Comm_size()` returns number of processes available for MPI communication and `MPI_Comm_rank()` returns its own unique rank id for identification within a communicator group.

```
1 MPI_Init(&argc , &argv);  
2 MPI_Comm_size(MPI_COMM_WORLD, &numNodes); // How many nodes  
3 MPI_Comm_rank(MPI_COMM_WORLD, &myId); // My Rank ID
```

Listing 3.6: Parallel GA implementation using MPI: Source code of main()

3.6.3 Creating MPI derived datatype for genotype

MPI requires data to be in a contiguous memory location for marshalling and demarshalling of data to send and receive. Any non-primitive datatype needs to be defined for MPI to handle this properly. Listing 3.7 shows the procedure

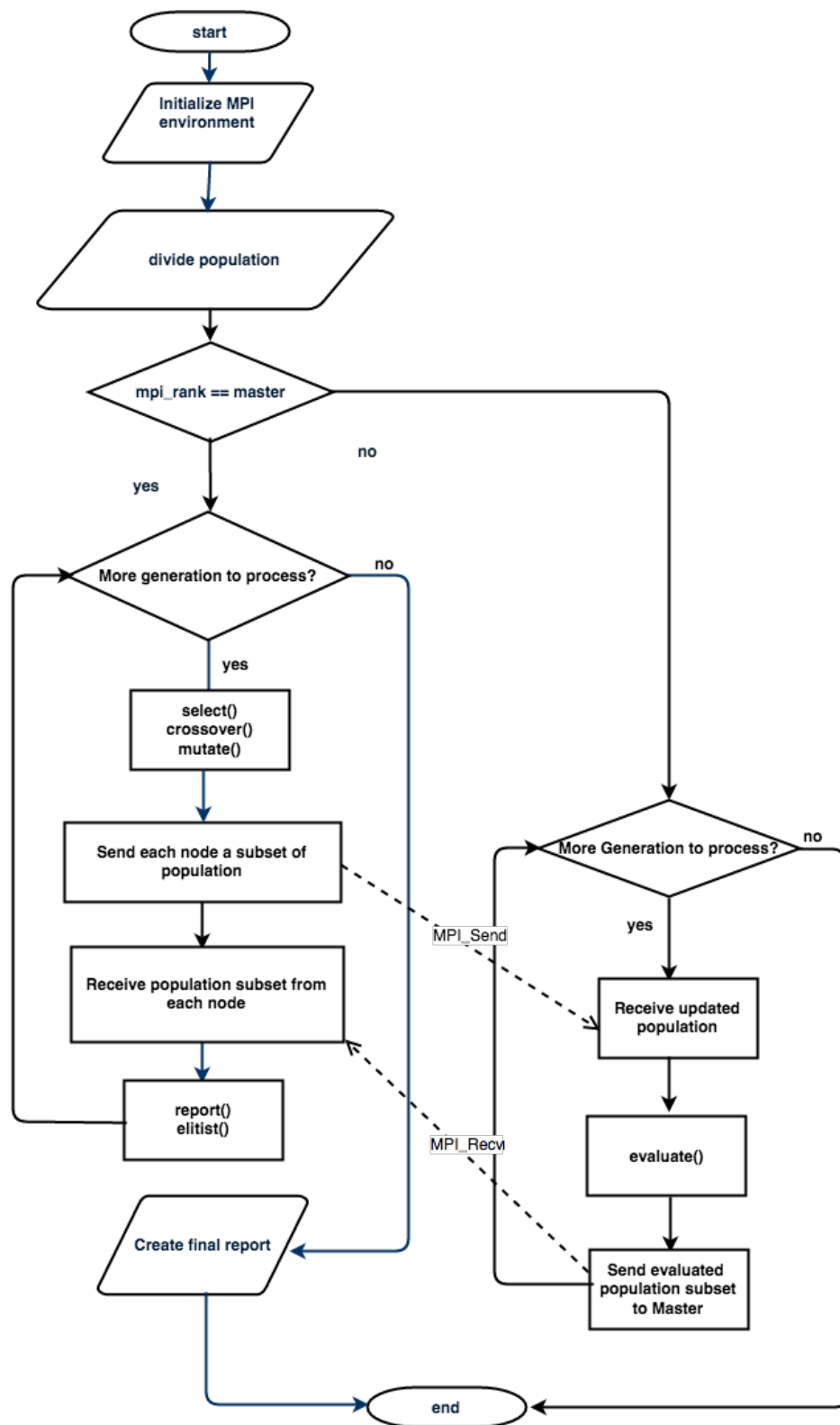


Figure 3.4: Flowchart diagram of Implemented master-slave Parallel GA

for creating a MPI derived datatype for the genotype struct of SGA. Line 14 contains the MPI routine `MPI_Type_create_struct()` that requires 3 arrays containing information about user data types, their count and their offsets values within the struct along with original data type that is being converted and an `MPI_Datatype` object where this routine will return the new type on success. In lines 7-12 the offsets values of genotype members are being stored in offsets array using C library macro `offsetof()` function.

```

1 void create_mpi_genotype_struct(MPI_Datatype& mpi_genotype){
2     const int nitems = 6;
3     int blocklengths[6] = {NVAR, 1, NVAR, NVAR, 1, 1};
4     MPI_Datatype types[6] = {MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE,
5                               MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE};
6
7     MPI_Aint offsets[6];
8     offsets[0] = offsetof(genotype, gene);
9     offsets[1] = offsetof(genotype, fitness);
10    offsets[2] = offsetof(genotype, upper);
11    offsets[3] = offsetof(genotype, lower);
12    offsets[4] = offsetof(genotype, rfitness);
13    offsets[5] = offsetof(genotype, cfitness);
14
15    MPI_Type_create_struct(nitems, blocklengths, offsets, types, &
16                           mpi_genotype);
17    MPI_Type_commit(&mpi_genotype);
18 }

```

Listing 3.7: Creating an MPI derived data type for genotype struct.

3.6.4 Partitioning of data

Following strategy shown in Listing 3.8 is applied to partition the population among available nodes. A 2D array `node_data[][]` shown in line 1 is used to hold the starting index and the row count for a given node id. From line 3 to line 11 the starting index and row counts are calculated and stored in `node_data[][]` array for each `node_id`.

```

1  int nodes_data[numNodes][2]; // Array to hold data partition
    information
2  int start_i, num_rows;
3  int chunk_size = (int)(ceil((double) POPSIZE / numNodes));
4
5  for(int i = 0; i < numNodes; i++)
6  {
7      start_i = i * chunk_size;
8      num_rows = min(chunk_size, POPSIZE - start_i);
9      nodes_data[i][0] = start_i;
10     nodes_data[i][1] = num_rows;
11 }

```

Listing 3.8: Partitioning population data among nodes

3.6.5 PGA v1

Master-Slave Communication

In master-slave model all processes run the same MPI program. But depending on its role each process executes parts of the program. The parts of the main() shown in Listing 3.9. On line 3 each process gets its own communicator ID by calling MPI_Comm_rank(). Using this ID master process executes the master_process() (line 5) and all other process executes slave_process()(line 9).

For each generation, the master_process() is responsible for performing selection, crossover and mutation operation on entire population, dividing population among all available nodes, updating all nodes with their parts of current population, collecting evaluated fitness values from each nodes, evaluate the best member for that generation, report and perform elitist operation on entire population. Where slave process is only responsible to receive a subset of population from master, evaluate their fitness values, send them back to master. Figure 3.5 shows the master and slave data decomposition and the GA operation they perform on the population data depends on their role.

```

1  ierr = MPI_Init(&argc, &argv);

```

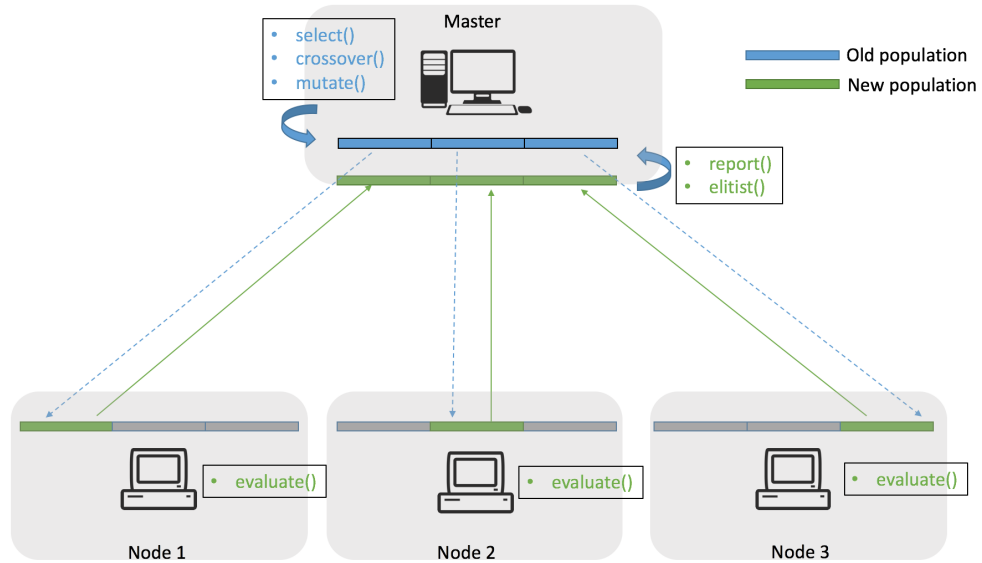


Figure 3.5: PGA v1 model

```

2 ierr = MPI_Comm_size(MPI_COMM_WORLD, &numNodes); // How many nodes
3 ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myId); // My Rank ID
4
5 if(myId == MASTER_ID) // Master Process
6 {
7     master_process(filename, seed, mpi_genotype, nodes_data,
8         start);
9 }
10 else // Slave processes
11 {
12     slave_process(seed, mpi_genotype, nodes_data);
13 }

```

Listing 3.9: Master and slave pr

Master process

In Listing 3.10 parts of the `master_process()` procedure is shown. After initialising the initial population master starts performing GA operation on them (lines 3-6). On lines 9-14 it starts sending subset of population to each nodes using `MPI_Send()` routine. As we shall see in Listing 3.11 there is a

corresponding MPI_Recv routine call (line 2) invoked by the slave process. Master then evaluate it's own parts the population subset (line 17). On lines 20-25 master starts to gather the parts of population which have their fitness value calculated by the slave nodes. Then on line 27 master perform elitist() operation using entire population.

```

1  for(int generation = 0; generation < MAXGENS; generation++)
2  {
3      selector ( seed );
4      crossover ( seed );
5      mutate ( seed );
6      report(generation);
7
8      int start_index, item_count;
9      for(int nodeId = 1; nodeId < numNodes; nodeId++)
10     {
11         start_index = nodes_data[nodeId][0];
12         item_count = nodes_data[nodeId][1];
13         MPI_Send(&population[start_index], item_count,
14         mpi_genotype, nodeId, SEND_DATA_TAG, MPI_COMM_WORLD);
15     }
16
17     /* Evaluate master's parts of the population */
18     evaluate(nodes_data[0][0], nodes_data[0][1]);
19
20     /* Collect all parts of new population from each node */
21     for(int nodeId = 1; nodeId < numNodes; nodeId++)
22     {
23         start_index = nodes_data[nodeId][0];
24         item_count = nodes_data[nodeId][1];
25         MPI_Recv(&population[start_index], item_count,
26         mpi_genotype, nodeId, RETN_DATA_TAG, MPI_COMM_WORLD, &status);
27     }
28
29     elitist();
30 }

```

Listing 3.10: The master_process() procedure.

Slave process

As we can see in Listing 3.11 the slave process starts with a `MPI_Recv()` call on line 4. After receiving the parts for the population slave process perform `evaluate()` using its own starting index and row count on line 7. When the fitness evaluation is done it start to send the population part to master process by calling `MPI_Send()` routine.

```
1 for(int generation = 0; generation < MAXGENS; generation++)
2 {
3     /* Get a subset of population from master */
4     ierr = MPI_Recv(&population[my_start], row_count, mpi_genotype
5     , MASTER_ID, SEND_DATA_TAG, MPI_COMM_WORLD, &status);
6
7     /* Perform fitness evaluation */
8     evaluate (my_start, row_count);
9
10    /* Send evaluated population to master */
11    MPI_Send(&population[my_start], row_count, mpi_genotype,
12    MASTER_ID, RETN_DATA_TAG, MPI_COMM_WORLD);
13 }
```

Listing 3.11: The `slave_process()` procedure.

3.6.6 PGA v2

Performance issue with PGA v1

The first implementation of Master-slave GA as in PGA v1 distribute evaluation of fitness function among several slave processors while master executes the GA operations(selection, crossover and mutation). This implementation explore the search space in exactly same manner as a serial GA and follows exactly the same simple GA design guidelines. It is thought to be the case that, this implementation of master-slave parallel GA results in significant improvement in performance. However, as we shall see in Chapter 4 when empirical studies were carried out the actual performance gains seemed to be worse than the serial GA. This is perhaps for the frequent MPI communication overhead

that required in each generation for evaluating fitness values back and forth to the master and slave nodes. The execution time of master-slave GAs have two components; the time used for communication between nodes and the time used in computation. So, the other reason is, master is responsible for performing all GA operations on entire population while slave nodes are mostly sitting idle during this time. With this approach slave nodes are under utilised where master node has a lot of workload

So, in this version of PGA we tried to leverage some of the GA operations as well as evaluating fitness values to the slave nodes. The movements of population data and node specific GA operations are depicted in Figure 3.6.

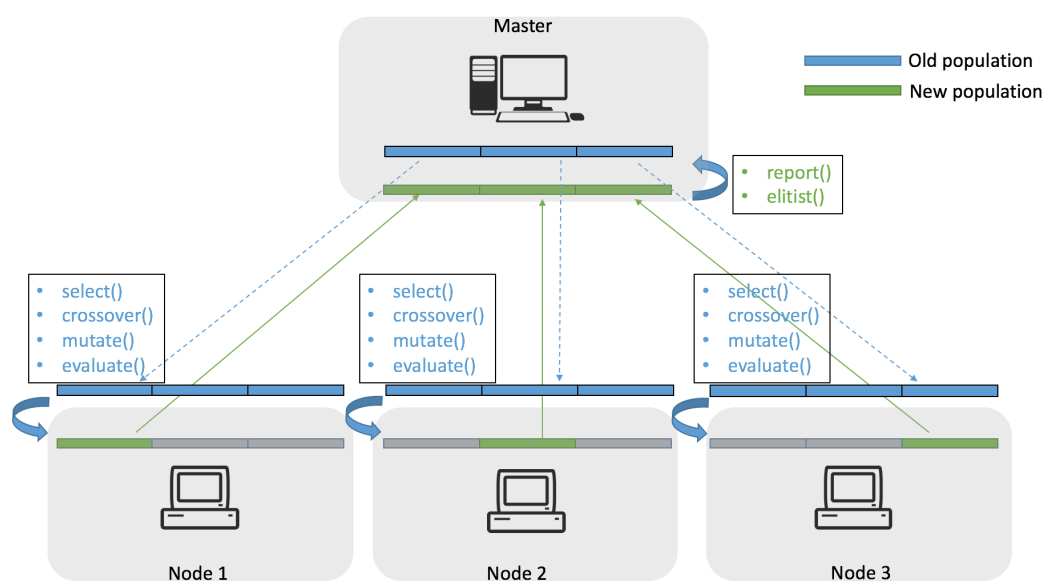


Figure 3.6: PGA v2 model

Using MPI Broadcast instead of explicit MPI Send/Recv

To reduce communication overhead MPI collective routine **MPI_Bcast()** is used by replacing explicit **MPI_Send()** and **MPI_Recv()**. To ensure proper synchronisation among all nodes in the cluster the **MPI_Barrier()** is called. When a process calls **MPI_Barrier()**, it waits until all of the nodes also call this procedure.

Some part of this implementation is shown in Listing 3.12. On lines 5-9 master node initialises initial population and perform first elitist operation(keep_the_best()). Slave nodes wait for these operation to complete because of MPI_Barrier() call on line 12. On line 15 entire population is broadcasted with MPI_Broadcast() call followed by another call to MPI_Barrier(). On lines 20-21 all nodes determine their own data partition information. Then all nodes perform GA operations and evaluate the fitness values for the sub-population(lines 22-27). On lines 32-36 each node then broadcasts its part of the population data. This followed by another synchronisation on line 38 before master node call the elitist() operation on entire population on line 43.

```

1 ierr = MPI_Comm_size(MPI_COMM_WORLD, &numNodes); // How many nodes
2 ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myId); // My Rank ID
3
4 /* Initialise first gen population in master process */
5 if(myId == MASTER_ID) // Master Process
6 {
7     initialize ( filename , seed );
8     evaluate (0, POPSIZE );
9     keep_the_best ( );
10 }
11
12 MPI_Barrier(MPI_COMM_WORLD);
13
14 /* Broadcast entire first generation from master process */
15 MPI_Bcast(population , POPSIZE + 1, mpi_genotype, MASTER_ID,
16           MPI_COMM_WORLD);
17
18 MPI_Barrier(MPI_COMM_WORLD);
19
20 /* Each process does GA ops here on individual data parts */
21 int my_start = nodes_data[myId][0];
22 int row_count = nodes_data[myId][1];
23 for ( generation = 0; generation < MAXGENS; generation++ )
24 {
25     node_selector (my_start, row_count, seed );
26     crossover ( seed );
27     node_mutate ( my_start, row_count, seed );

```

```

27     evaluate (my_start, row_count);
28
29     /* Sync message - 3 */
30     MPI_Barrier(MPI_COMM_WORLD);
31
32     for(int i = 0; i < numNodes; i++)
33     {
34         MPI_Bcast(&population[nodes_data[i][0]], nodes_data[i]
35         ][1], mpi_genotype, i, MPI_COMM_WORLD);
36         MPI_Barrier(MPI_COMM_WORLD);
37     }
38
39     MPI_Barrier(MPI_COMM_WORLD);
40
41     if(myId == MASTER_ID) // Master Process
42     {
43         report(generation);
44         elitist ( );
45     }
46
47     MPI_Barrier(MPI_COMM_WORLD);
48 }

```

Listing 3.12: PGA v2 main() procedures.

3.6.7 PGA v3

Running Time Analysis of node_selector() Procedure

During empirical study of program running time, with PGA v2 we noticed some performance gain when number of processors are increased for smaller population size(5000). We also noticed there is a drop of speedup when population size is increased.

After a some investigation into this problem it was identified that the running time of **selector()** procedure is contributing for this unexpected result. The Listing 3.13 parts of this procedure where we can see there is a nested for loop. The outer loop(line 2) on line has a range of $(0, n/p)$ and the inner loop(line

11) has a range of $(0, p)$ where n is the population size and p is number of processors. Thus this function has a running time complexity of $\mathcal{O}(n/p * n)$ which is essentially $\mathcal{O}(n^2)$.

Since, all nodes are executing this `node_selector()` procedure using entire global population, the total running time is directly dependent upon the slowest node/processor in the cluster plus communication overheard for each generation cycles.

```

1  /* Select survivors using cumulative fitness. */
2  for ( i = my_start; i < ends; i++ )
3  {
4      p = r8_uniform_ab ( a, b, seed );
5      if ( p < population[0].cfitness )
6      {
7          newpopulation[i] = population[0];
8      }
9      else
10     {
11         for ( j = 0; j < POPSIZE; j++ )
12         {
13             if ( population[j].cfitness <= p && p < population[j
+1].cfitness )
14             {
15                 newpopulation[i] = population[j+1];
16             }
17         }
18     }
19 }

```

Listing 3.13: `node_select()` procedure.

Limiting Selection to Partial Population

Based on the analysis discussed above, in this implementation the `selector()` procedure is modified so that the selection is done using nodes local subpopulation. Thus this would result in running time complexity of $\mathcal{O}((n/p) * (n/p))$ i.e. $\mathcal{O}((n/p)^2)$. Theoretically, if we could add more nodes i.e. processors we

could reduce the affect of population size increases for this procedure proportionally. We know that number of processor cannot always be matched with population size increases but for a reasonably large population size (not infinite), adding in extra nodes can reduce the running time complexity of this function significantly. Figure 3.7 shows the data partition and GA operation for master and slave nodes. All GA operation including selection is done on subpopulation.

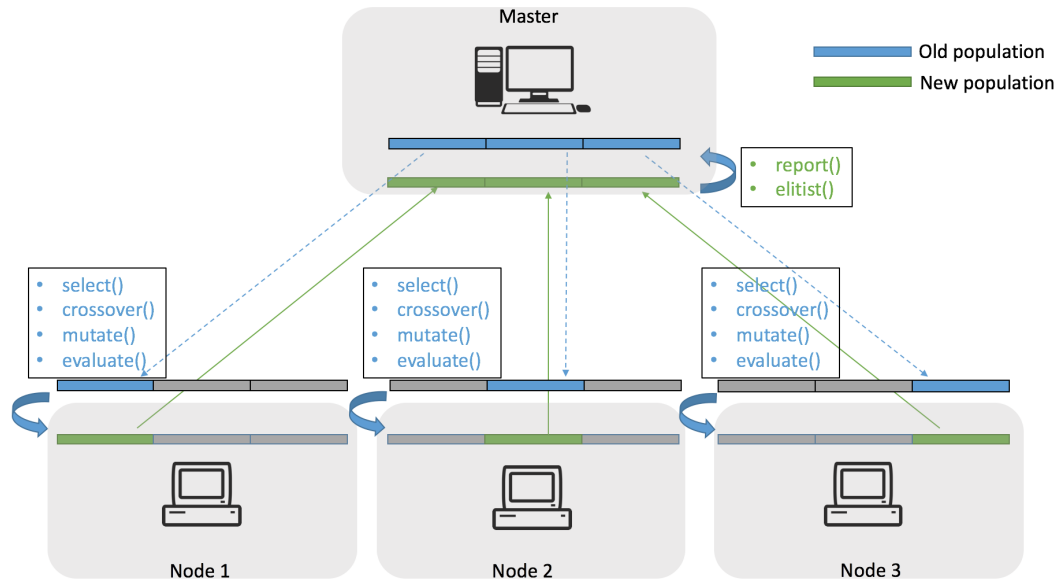


Figure 3.7: PGA v3 model

Changes to the selector functions are shown in Listing 3.14 where the range inner for loop at line 10 is limited for the local subpopulation only which gives a range of $(0, n/p)$.

```

1  for ( i = my_start; i < ends; i++ )
2  {
3      p = r8_uniform_ab ( a, b, seed );
4      if ( p < population[my_start].cfitness )
5      {
6          newpopulation[i] = population[my_start];
7      }
8      else
9      {

```

```

10         for ( j = my_start; j < ends; j++ )
11         {
12             if ( population[j].cfitness <= p && p < population[j
13               +1].cfitness )
14             {
15                 newpopulation[i] = population[j+1];
16             }
17         }
18     }

```

Listing 3.14: Changes made in node_select() procedures.

3.6.8 Issues with Serialising Dynamic Data

The SGA implementation allows population size, number of variables in a chromosome and number of generation can only be modified at compile time. To carry out the experiments we needed the PGA accept these parameters at runtime. So, an attempt was made to change these parameters at runtime with command line arguments. After a week of trials, this attempt was unsuccessful because of a number of reasons which are discussed here. Part of the problem was due to the data structure chosen to represent the chromosome which contains three static array members. The length of these arrays are defined at compile time. Population are initialised inside initialize procedure (Listing ??). Each individual is initialised with random permissible value. In this trial changes were made to the genotype data structure so that it could be initialised with dynamic arrays using malloc(). Program was compiled normally without any error. But during testing the PGA failed to produce acceptable outcome. To find the underlying issue GDB was used to debug the problem. During this debugging program was run with a population size of 5. So the best member of a generation is kept at index 5.

It was noticed(GDB output - Listing 3.6.8) that some values of population members were changing unexpectedly. As we can see from the source code given in Appendix C.7 the initialize() procedure only assign initial value for

lboud, **ubound** and **gene**. In this debugging session, the values of *i*, *j* and the best individual(`population[5]`) was monitored to catch unexpected any change. On line 42 we see that the value of `population[5].gene` is `0x626330`. But on next step on line 54 this value has changed to `0x3fe197d48c1f329c`. We know that the `initialize()` procedure only assign values for the population member from 0 to (`POPSIZE - 1`) which is 4 in this run. So, the changes in `population[5]` is unexpected. It appeared that even though no value was assigned for best individual (which is `population[5]`), its member's values were changing. After looking more closely, it was noticed that the value that was being assigned to a data member for one individual was appearing inside some other individual's data member. Which means values were being assigned into wrong memory location. This indicated that `malloc()` for initialising the population array was not correct.

```

1 847 population[j].gene[i] = r8_uniform_ab ( lboud, ubound, seed );
2 1: population[5] = {fitness = 0, rfitness = 1.6304166312761136e
   -322,
3  _ cfitness = 0, gene = 0x626330}
4 2: i = 0
5 3: j = 2
6 (gdb)
7 840 for ( j = 0; j < POPSIZE; j++ )
8 1: population[5] = {fitness = 0, rfitness = 1.6304166312761136e
   -322,
9  _ cfitness = 4.147546169663503, gene = 0x626330}
10 2: i = 0
11 3: j = 2
12 (gdb)
13
14 .....
15
16 840 for ( j = 0; j < POPSIZE; j++ )
17 1: population[5] = {fitness = 0, rfitness = 1.6304166312761136e
   -322,
18 _ cfitness = 4.147546169663503, gene = 0x626330}

```



```

19 2: i = 1
20 3: j = 1
21 (gdb)
22 842 population[j].fitness = 0;
23 1: population[5] = {fitness = 0, rfitness = 1.6304166312761136e
    -322,
24 _ cfitness = 4.147546169663503, gene = 0x626330}
25 2: i = 1
26 3: j = 2
27 (gdb)
28 843 population[j].rfitness = 0;
29 1: population[5] = {fitness = 0, rfitness = 1.6304166312761136e
    -322,
30 _ cfitness = 4.147546169663503, gene = 0x626330}
31 2: i = 1
32 3: j = 2
33 (gdb)
34 844 population[j].cfitness = 0;
35 1: population[5] = {fitness = 0, rfitness = 1.6304166312761136e
    -322,
36 _ cfitness = 4.147546169663503, gene = 0x626330}
37 2: i = 1
38 3: j = 2
39 (gdb)
40 847 population[j].gene[i] = r8_uniform_ab ( lbound, ubound, seed );
41 1: population[5] = {fitness = 0, rfitness = 1.6304166312761136e
    -322,
42 _ cfitness = 4.147546169663503, gene = 0x626330}
43 2: i = 1
44 3: j = 2
45 (gdb)
46 840 for ( j = 0; j < POPSIZE; j++ )
47 1: population[5] = {fitness = 0, rfitness = 1.6304166312761136e
    -322,
48 _ cfitness = 4.147546169663503, gene = 0x3fe197d48c1f329c}

```

```

49 2: i = 1
50 3: j = 2
51 (gdb)
52 842 population[j].fitness = 0;
53 1: population[5] = {fitness = 0, rfitness = 1.6304166312761136e
    -322,
54 _ cfitness = 4.147546169663503, gene = 0x3fe197d48c1f329c}
55 2: i = 1
56 3: j = 3

```

After this discovery, one and half week time was spent on how to allocate memory correctly for the **genotype** struct. But it seemed quite complicated to do so with three variable length array members in a struct. By researching on Internet about this, it was found that this issue is known as Variable Length Array In Struct (VLAIS). And with further research, some discussions related to this topic was found in Stackoverflow. The suggestions there were indicating that VLAIS is permissible on specific compiler i.e GCC C99 compatible only. In one discussion posted on Stackoverflow ¹ related to this topic further suggested that that variable-length arrays in structs is permissible in gcc (and the newest C-standards), but the variable array must always be the last member of the struct (so the compiler knows where all members are). In particular this means:

1. There can be only one variable-length array
2. If the struct is a member of another struct, it has to be the last member of that struct.

Another post ² also suggested that it is not possible to have more than one flexible-array member in a struct.

There is no clear information whether mpic++ compiler has support for this feature of gcc too or not. The alternative solution to this problem would be to

¹<http://stackoverflow.com/questions/21804994/using-a-struct-member-as-array-size-in-the-same-struct>

²<http://stackoverflow.com/questions/17552312/multiple-flexible-array-in-a-struct-in-c>

replace struct data structure with combination of C++ objects and vectors. Since this would have involved making a considerable changes to actual SGA implementation and possibilities of going back to the earlier problem with Object-oriented GA, a decision was made to leave these parameters to be defined at compile time.

3.7 Testing

To ensure the Parallel Genetic Algorithms implementations are producing expected outcome there is no alternative but to carry out testing. These tests were performed to validate the correctness of PGA implementation. We wanted to see if the output of the PGA program is consistent with the output of original SGA program using same parameters values and random seed.

3.7.1 Test Configuration

The common parameter values shown in Table 3.3 are used for all testing both SGA and PGA programs.

As the original SGA was implemented with Real-value encoding scheme for genome representation the range of these values are given using a input file. For this test configuration with 3 genome per chromosome the permissible values are given as listed in Table 3.4.

As discussed in section 3.2.5 the hosts information entries shown in Listing 3.15 were added for Hydra (MPI Process Manager). The entry at line 1 is master node's hostname. Which means master node also is acting as a compute node in the cluster.

```
1 padma:2
2 meghna:2
3 jamuna:2
```

Listing 3.15: The hosts file configuration for Hydra.

Table 3.3: Parameters alues used with test cases.

Parameter	Description	Value
POPSIZE	Population Size	5000
MAXGENS	Number of Generation	100
NVARS	Number of variables in a Chromosome	3
PXOVER	Probability of Crossover	0.8
PMUTATION	Probability of Mutation	0.15
FITNESS_F	Fitness function (F8: Griewank)	F8
SEED	Random value generator seed	123456789
Genome Encoding	Encoding scheme for genome representaion	Real value Encoding

Table 3.4: Genome value range for 3 variables Griewank function

Lower	Upper
-512.0	511.0
-512.0	511.0
-512.0	511.0

The PGA program was run by using following command:

```
1 $mpiexec -f ~/hosts -n 6 ./pga
```

The command **mpiexec** takes in parameter **-f** for the hosts file name and **-n** for the number of processes that needs to be created in the cluster. The example shown above will launch 6 process that will run the executable binary **pga**.

All tests results shown in this section are carried out in Hardware based Beowulf system not in simulated cluster.

3.7.2 Tests Outcome

The GA programs output for the SGA, PGA v1, PGA v2 and PGA v3 are shown below. As these tests were run with a generation size if 100 the full output is not shown to save space. Instead, outputs are shown into two split images "Top part" and "Bottom part" of the screen capture that convey most important information. We are assuming the original SGA program is producing a correct output. Then the subsequent 3 PGA programs outputs are compared against SGA's output to validate and analyse each PGA implementation.

SGA Output

Figures 3.8 and 3.9 shows the output of original SGA program compiled with the parameter configuration discussed in section 3.7.1.

PGA v1 Output

Figures 3.10 and 3.11 shows the output of original PGA v1 compiled with the parameter configuration discussed in section 3.7.1.

```

25 March 2017 10:19:39 PM

SIMPLE_GA:
  A simple example of a genetic algorithm.

  Generation      Best      Average      Standard
  number          value      fitness      deviation

      0          128.757      62.1728      26.7323
      1          128.785       72.415      25.7332
      2          130.635      79.1694      24.9169
      3          131.366      83.5163      24.6556
      4          131.366      87.5812      24.5354
      5          131.715      90.4655      23.9413
      6          131.715      92.3977      23.2717
      7          131.719       93.984      22.8934
      8          131.719      95.3116      22.7963
      9          131.719      95.8995      22.9744
     10          131.909      96.702      22.8032

```

Figure 3.8: SGA output - Top part

```

      94          132.112      104.768      24.8305
      95          132.112      105.558      24.5186
      96          132.112       105.52      24.8167
      97          132.112      105.414      24.3966
      98          132.112      105.682      24.5892
      99          132.112      105.461      24.4786

  Best member after 100 generations:

  var(0) = -509.328
  var(1) = -511.573
  var(2) = 384.602

  Best fitness = 132.112

SIMPLE_GA:
  Normal end of execution.

25 March 2017 10:19:52 PM

```

Figure 3.9: SGA output - Bottom part

PGA v2 Output

Figures 3.12 and 3.13 shows the output of original PGA v2 compiled with the parameter configuration discussed in section 3.7.1.

```

25 March 2017 10:31:22 PM

PARALLE_GA:
A simple example of a genetic algorithm.

Generation      Best      Average      Standard
number          value      fitness      deviation

      0          128.757      62.1728      26.7323
      1          128.785      72.415       25.7332
      2          130.635      79.1694      24.9169
      3          131.366      83.5163      24.6556
      4          131.366      87.5812      24.5354
      5          131.715      90.4655      23.9413
      6          131.715      92.3977      23.2717
      7          131.719      93.984       22.8934
      8          131.719      95.3116      22.7963
      9          131.719      95.8995      22.9744
     10          131.909      96.702       22.8032

```

Figure 3.10: PGA v1 output - Top part

```

      94          132.112      104.768      24.8305
      95          132.112      105.558      24.5186
      96          132.112      105.52       24.8167
      97          132.112      105.414      24.3966
      98          132.112      105.682      24.5892
      99          132.112      105.461      24.4786

Best member after 100 generations:

var(0) = -509.328
var(1) = -511.573
var(2) = 384.602

Best fitness = 132.112

PARALLEL_GA:
Normal end of execution.

25 March 2017 10:31:50 PM

```

Figure 3.11: PGA v2 output - Bottom part

PGA v3 Output

Figures 3.14 and 3.15 shows the output of original PGA v3 compiled with the parameter configuration discussed in section 3.7.1.

```

02 April 2017 09:10:37 PM

PARALLEL_GA:
A simple example of a genetic algorithm.

  Generation      Best      Average      Standard
  number          value      fitness      deviation

      0           128.757      59.5805      27.794
      1           129.737      69.2161      27.8682
      2           131.023      74.7946      27.8082
      3           131.05       78.6934      27.4979
      4           131.862      81.5396      27.1012
      5           131.862      83.7855      27.1321
      6           131.862      84.8715      27.7661
      7           131.862      86.3192      27.5143
      8           131.862      87.5834      27.1987
      9           131.946      87.6091      27.9243
     10           131.946      87.7072      28.2813

```

Figure 3.12: PGA v1 output - Top part

```

      94           132.116      95.7185      29.6399
      95           132.116      96.0822      29.6313
      96           132.116      95.3349      29.7867
      97           132.116      96.23       29.6458
      98           132.116      96.8958      29.3822
      99           132.116      95.4637      29.7365

Best member after 100 generations:

var(0) = -509.233
var(1) = -511.377
var(2) = -454.926

Best fitness = 132.116

PARALLEL_GA:
Normal end of execution.

02 April 2017 09:10:39 PM

```

Figure 3.13: PGA v2 output - Bottom part


```

02 April 2017 09:11:03 PM
PARALLEL_GA:
  A simple example of a genetic algorithm.

  Generation      Best      Average      Standard
  number          value      fitness      deviation

      0          128.757          9.5111          23.3285
      1          128.757         14.6091          27.2019
      2          128.757         16.0109          28.127
      3          128.757         20.4626          29.1089
      4          128.757         24.1669          28.6299
      5          128.757         28.5962          29.4346
      6          128.757         31.5063          29.3996
      7          128.757         34.3902           29.89
      8          128.757         28.3249          29.1682
      9          128.757         35.8326          30.4069
     10          129.787         36.0677          29.0343

```

Figure 3.14: PGA v3 output - Top part

```

      94          131.929         55.0006          19.8756
      95          131.929         56.8745          18.3887
      96          131.929         52.9122          19.0669
      97          131.929         54.6168          18.4083
      98          131.929         57.1255          17.8153
      99          131.929         57.9138          17.5553

  Best member after 100 generations:

  var(0) = 508.718
  var(1) = -510.9
  var(2) = 125.527

  Best fitness = 131.929
SIMPLE_GA:
  Normal end of execution.
02 April 2017 09:11:08 PM

```

Figure 3.15: PGA v3 output - Bottom part

3.8 Discussion

The Beowulf implementation required a further steps for administrative tasks such and allowing remote access using vncserver, configuring ddns so that the system is accessible while not working from home. Since, the development needed the cluster setup to compile and and run MPI program the Integrated Development Environment set was also configured so that it able to use the remote cluster infrastructure. The necessary details are omitted here because of space constraint and they are not the main focus point of this project.

As described in previous section in this chapter, in this project we only explored one model of parallel GA which is master-slave model. In conjunction with the tests results and experiments shown in next chapter we see that pure master-slave model of PGA(PGA v1) did not give us any speed up at all, which is understandable since the communication overhead for message-passing actually more costly than actual computation done by multiple nodes. With PGA v2 where we changed the algorithm to utilise all nodes to do the GA operations instead of just computing fitness value we started to see speedup improvements. But this gains were not as much as expected. Lastly, in PGA v3 we limited the selection space to the subpopulation that a node operates on. This implementation gives us a good speedup gains. However, as we can see from the tests result of PGA v3 the actual outcome of GA is slightly worse than that of original SGA. That is because PGA v3 is close to a distributed PGA model as discussed in Section 2.5.3 without any migration policy. Next logical step of this project would be to implement a distributed model of PGA with a migration policy. Due to the time constraint of this project we needed to stop at the PGA v3.

Chapter 4

Empirical studies

4.1 Introduction

In this chapter the runtime of three Parallel GA(PGA) implementations (discussed in previous chapter) are compared against the sequential GA implementation. Each implementation is compared and contrasted with SGA. We focused this study on to observe the program runtime by varying number of processes and by varying the population size. These experiments were carried out on a physical Beowulf cluster setup as described in Section 3.7.1 with test configurations shown in Section 3.7.1. Results may vary using other cluster configuration.

Even though we kept the free-parameter's values consistent throughout the experiments there are other constraints on which we do not have any explicit control to fine tune them. This constraints include Linux scheduler (which runs other system daemons and services), and network system overhead. Extra care was taken to ensure no other programs were running during the experiments and network activities were minimum. To further validate our experiments data statistical method of confidence interval were used.

In statistics, a confidence interval (CI) is a type of interval estimation of a random variable (in this experiment which is the program run time). It is an observed interval (i.e. calculated from the observations). In principle this

interval is different from sample to sample, that potentially includes the un-observable true parameter of interest. How frequently the observed interval contains the true parameter if the experiment is repeated is called the confidence level. If confidence intervals are constructed in separate experiments on the same population following the same process, the proportion of such intervals that contain the true value of the parameter will match the given confidence level (Cox and Hinkley, 1979). Whereas two-sided confidence limits form a confidence interval, and one-sided limits are referred to as lower/upper confidence bounds (or limits).

All of the sample runtime data gathered here are averaged from 20 runs. Due to time constraint sample size is kept 20 which means we run each program for 20 times and recorded the time taken from start to finish for each run. To calculate CI the average \bar{x} and standard deviation δ is calculated of these 20 data set. Then the CI is calculated using following formula where $z = 1.96$ for 95% CI:

$$UpperLimit = \bar{x} + z(\delta/\sqrt{20})$$

$$LowerLimit = \bar{x} - z(\delta/\sqrt{20})$$

Table 4.1 shows the data of average running time for all three implementation of PGA using a population size 5000 with their upper and lower bound with 95% confidence level. Which means that 95% of time the true value of the program runtime is within our confidence interval. After any particular sample is taken, given that exactly same configuration were used, the program runtime should be within these intervals shown in the Table 4.1 95% of the time.

Using this method all data for the experiments listed below were validated. The results shown for these experiments are the average of 20 runs.

Table 4.1: A sample runs using population size of 5000 to determine average runtime(seconds) for PGA v1, PGA v2 and PGA v3.

RUN	PGA v1	PGA v2	PGA v3
0	14.8262	5.0441	1.7067
1	14.7368	5.0157	1.9845
2	15.0571	4.9169	1.7116
3	15.0404	4.9911	1.6933
4	16.2858	4.9056	1.6935
5	14.5847	5.0745	1.7016
6	14.7812	5.0515	1.6980
7	15.4188	5.0403	1.7069
8	16.4531	4.9878	1.6974
9	16.9943	4.8959	1.7039
10	15.7324	4.9217	1.6986
11	14.4173	4.9180	1.7055
12	14.6054	5.0879	1.7063
13	14.8513	4.9155	1.7112
14	15.4737	4.9294	1.7024
15	14.4207	5.0595	1.6966
16	16.0418	5.0947	1.7095
17	16.8885	5.0685	1.6873
18	15.0276	4.9684	1.7136
19	16.4414	4.9665	1.7077
Average	15.4039	4.9927	1.7168
Standard Deviation	0.8162	0.0670	0.0618
CI (95%)	0.3577	0.0294	0.0271
Upper Limit	15.7616	5.0220	1.7439
Lower Limit	15.0462	4.9633	1.6897

4.2 Experiments using PGA v1

4.2.1 Experiment 1: Varying number of processes

Objective

Comparison of PGA v1 runtime vs. SGA runtime by varying the number of processes the work is distributed for the PGA program to see the speedup gain using a fixed population size 5000.

Table 4.2: Experiment 1 data using PGA v1

# Process(es)	SGA	PGA	speedup
1	12.8789	12.8887	0.9992
2	12.8789	12.8464	1.0025
3	12.8789	13.1183	0.9818
4	12.8789	13.1585	0.9788
5	12.8789	13.4671	0.9563
6	12.8789	15.3387	0.8396

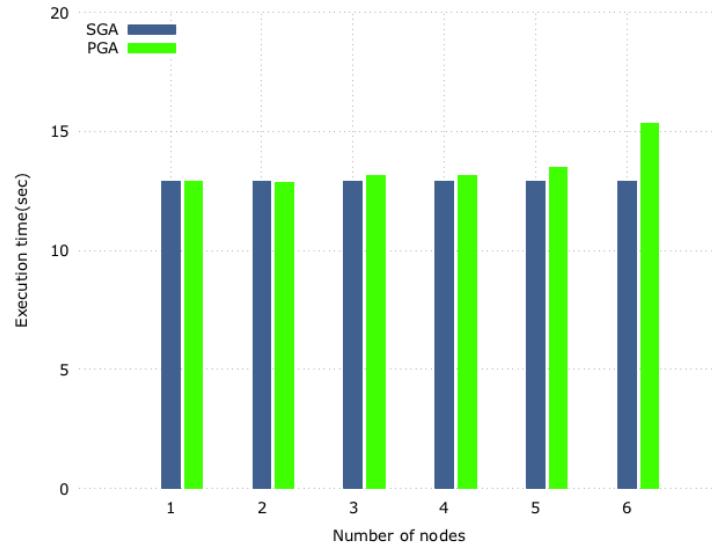


Figure 4.1: Experiment 1 - Runtime comparison SGA vs PGA v1.

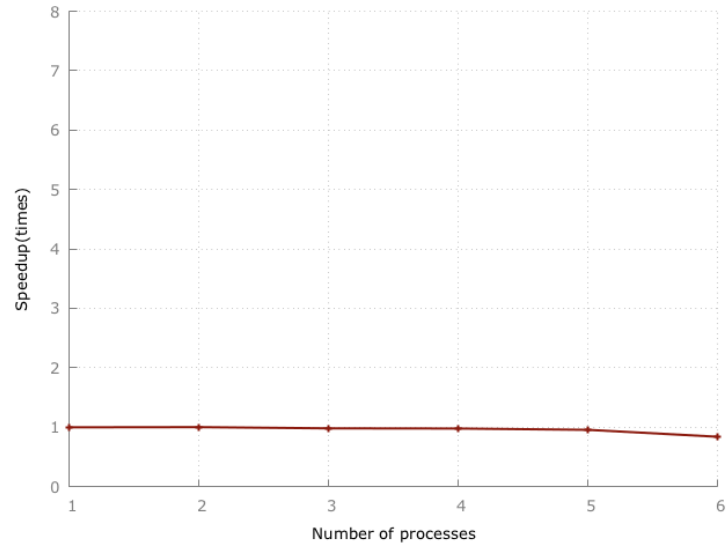


Figure 4.2: Experiment 1: Speedup graph SGA vs PGA v1.

Discussion

This experiment shows that there is no speedup gain when the number of processes were increased. That is because the communication overhead for MPI increases as the number of processes increases.

4.2.2 Experiment 2: Varying population size

Objective

Comparison of PGA v1 runtime vs. SGA runtime by varying the population size using 6 processes for PGA program.

Table 4.3: Experiment 2 data using PGA v1

# population	SGA	PGA	speedup
1000	0.57	0.6827	0.835
2000	2.1459	2.3018	0.9323
3000	4.7258	4.9238	0.9598
4000	8.2976	11.0389	0.7517
5000	12.8896	15.7361	0.8191
6000	18.4622	20.249	0.9118
7000	25.0535	26.8225	0.934
8000	32.667	36.5315	0.8942
9000	41.3425	42.1336	0.9812
10000	50.8885	51.9144	0.9802

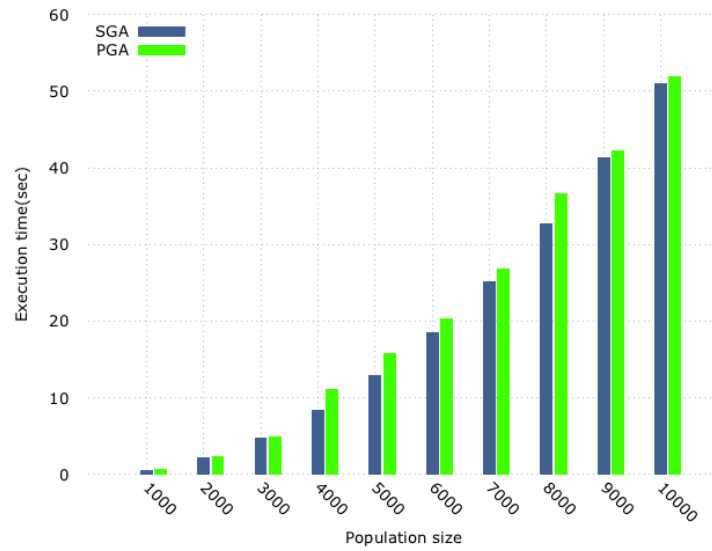


Figure 4.3: Experiment 2 - Runtime comparison SGA vs. PGA v1.

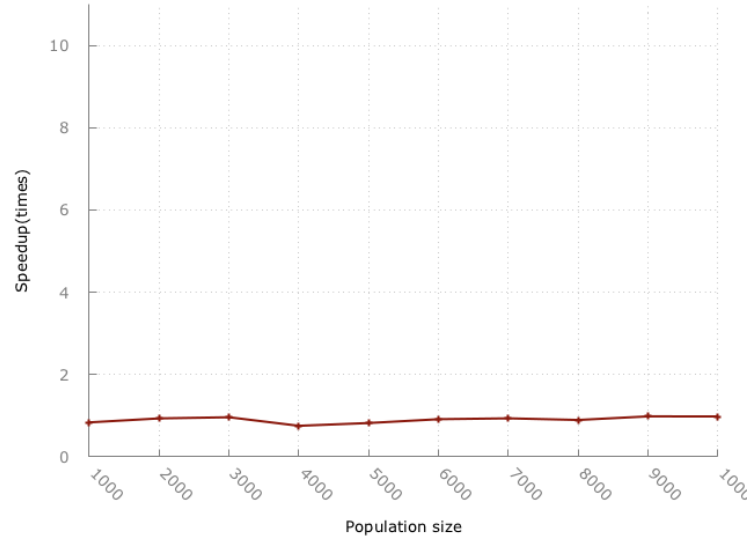


Figure 4.4: Experiment 2 - Speedup graph SGA vs. PGA v1.

Discussion

This experiment shows that there is no speedup gain as the population size were increased either. Again, this shows that the communication overhead for MPI has significant impact of overall runtime of the program. This overhead increases with population size.

4.3 Experiments using PGA v2

4.3.1 Experiment 3: Varying number of processes

Objective

Comparison of PGA v2 runtime vs. SGA runtime by varying the number of processes the work is distributed for the PGA program to see the speedup gain using a fixed population size 5000.

Table 4.4: Experiment 3 data using PGA v2

# Process(es)	SGA	PGA	speedup
1	12.8795	12.8713	1.0006
2	12.8795	8.6353	1.4915
3	12.8795	6.5845	1.956
4	12.8795	6.7115	1.919
5	12.8795	5.1471	2.5023
6	12.8795	5.0748	2.538

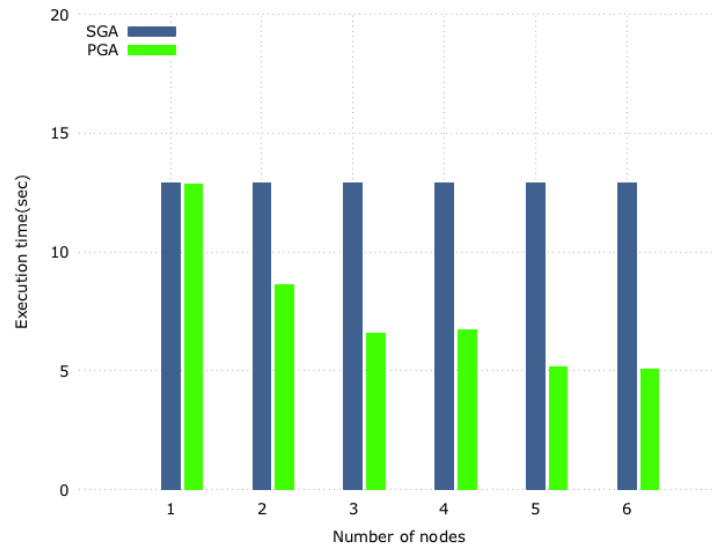


Figure 4.5: Experiment 3 - Runtime comparison SGA vs. PGA v2.

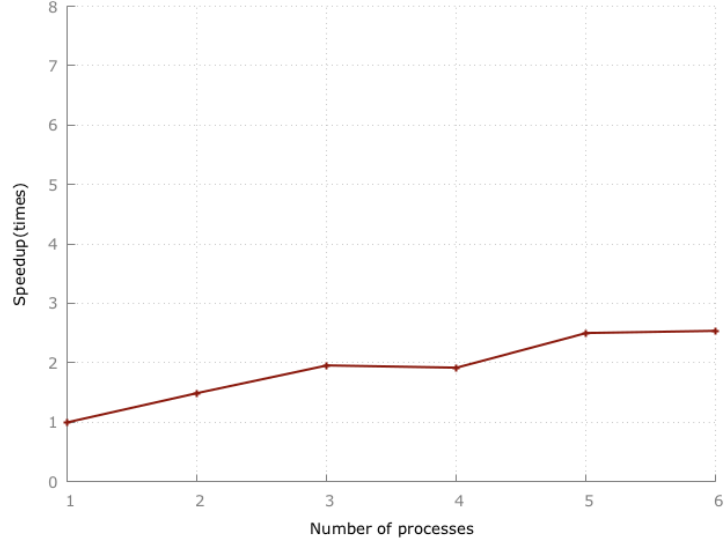


Figure 4.6: Experiment 3 - Speedup graph SGA vs. PGA v2.

Discussion

With this experiment we observed that there is speedup gain with PGA v2 as number of processes increases. However, this gain is much lower than expected with 6 processes given that the MPI collective communication routine were used for this implementation to reduce the communication overhead of PGA v1.

4.3.2 Experiment 4: Varying population size

Objective

Comparison of PGA v2 runtime vs. SGA runtime varying the number the population size using 6 processes for PGA program.

Table 4.5: Experiment 4 data using PGA v2

# population	SGA	PGA	speedup
1000	0.5698	0.4957	1.1495
2000	2.146	1.3006	1.65
3000	4.7207	2.2022	2.1437
4000	8.3287	3.4562	2.4098
5000	12.8959	4.8998	2.6319
6000	18.4703	7.016	2.6326
7000	25.0758	21.6343	1.1591
8000	32.7814	28.134	1.1652
9000	41.2779	31.5602	1.3079
10000	50.9403	44.5325	1.1439

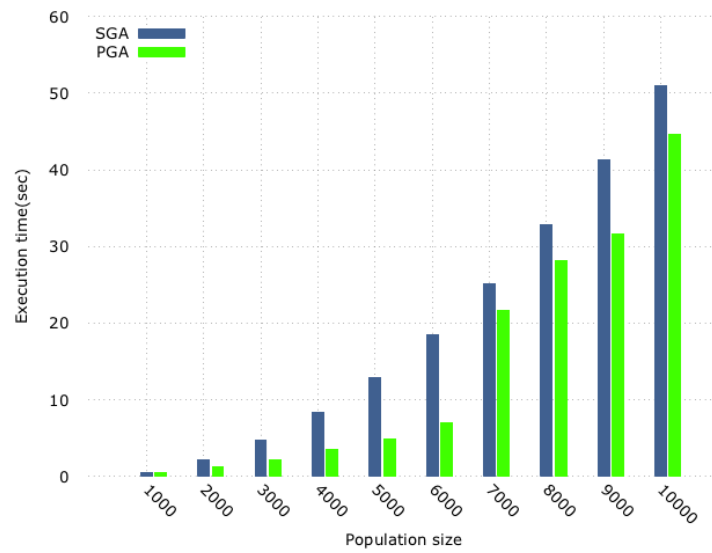


Figure 4.7: Experiment 4 - Runtime comparison SGA vs. PGA v2.

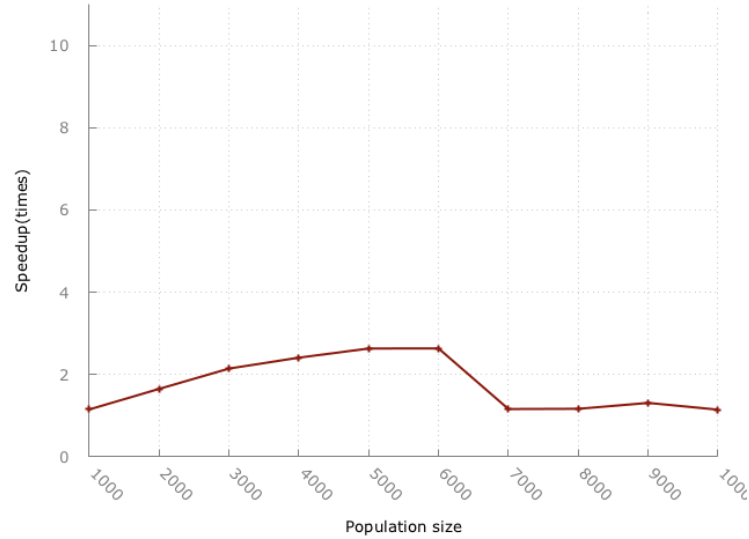


Figure 4.8: Experiment 4 - Speedup graph SGA vs PGA v2.

Discussion

Result of this experiment is very interesting as we can see from the graph shown in Figure 4.8. In Experiment 3 we saw somewhat of a steady gain in speedup for a fixed population size of 5000. But here we see as population size is increased (after 6000), there is a decrease in speedup. To find out the possible issue that may cause this behaviour, a running time analysis was carried to identify the problem that was discussed in section 3.6.7. It was identified that the selection procedure had a running time complexity of $\mathcal{O}(n^2)$. As population size is increased after a certain threshold (6000 in this experiment) the runtime to this program started to increase rapidly which degraded the overall performance of speedup of this program by parallelising with many processes.

4.4 Experiments using PGA v3

4.4.1 Experiment 5: Varying number of processes

Objective

Comparison of PGA v3 runtime vs. SGA runtime varying the number of processes the work is distributed for the PGA program to see the speedup gain using a fixed population size 5000.

Table 4.6: Experiment 5 data using PGA v3

# Process(es)	SGA	PGA	speedup(SGA/PGA)
1	12.878	13.3253	0.9664
2	12.878	3.5889	3.5883
3	12.878	3.511	3.6679
4	12.878	2.1288	6.0495
5	12.878	1.9293	6.675
6	12.878	1.7058	7.5496

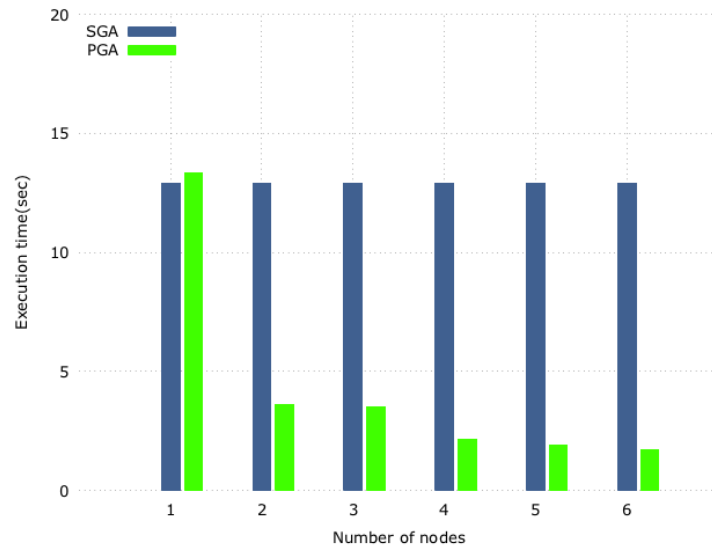


Figure 4.9: Experiment 5 - Runtime comparison SGA vs. PGA v3.

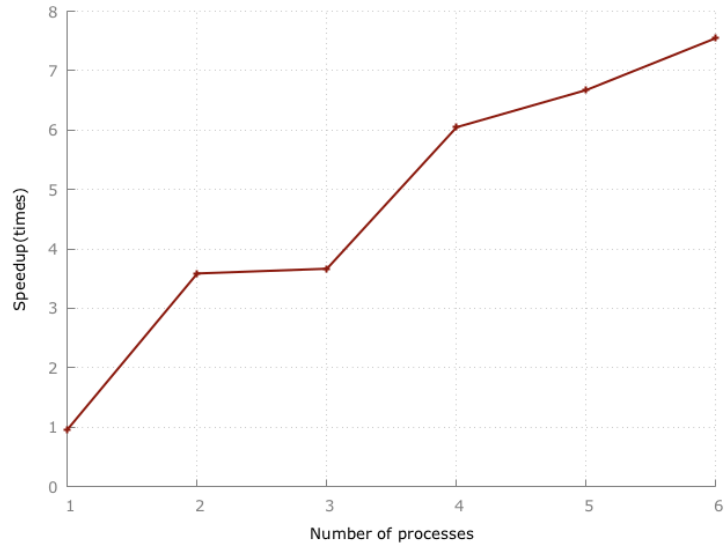


Figure 4.10: Experiment 5 - Speedup graph SGA vs. PGA v3.

Discussion

In this experiment we see a better improvement in overall speedup. But the curb of the gain is not smooth. From 2 processes to 3 processes there is not much gain but from 3 processes to 4 processes we see a sudden spike. Only explanation of this could be the fact that how MPI Process Manager creates process in the cluster. With the hosts file configuration shown in Section 3.15, Hydra (MPI Process Manager) runs 2 processes in one physical node. So, with both 3 and 4 processes we have 2 physical nodes running this program. Most likely, the underlying cache management and network overhead per node have something to do with the irregular performance variance. Running 2 processes in same physical node would have advantages of less network overheads and better caching over running a single process. This could only be confirmed by profiling this program using TAU(Tuning and Analysis Utilities) to further analyse it and find out any underlying cause should we had more time.

4.4.2 Experiment 6: Varying population size

Objective

Comparison of PGA v3 runtime vs. SGA runtime varying the number the population size using 6 processes for PGA program.

Table 4.7: Experiment 6 data using PGA v2

# population	SGA	PGA	speedup
1000	0.5694	0.3594	1.5844
2000	2.1434	0.6143	3.4894
3000	4.7145	0.9252	5.0954
4000	8.3063	1.3231	6.2777
5000	12.9011	1.6985	7.5957
6000	18.5207	2.2113	8.3753
7000	25.1309	2.7727	9.0636
8000	32.6705	3.4446	9.4846
9000	41.2867	4.2053	9.8179
10000	50.9591	5.0311	10.1288

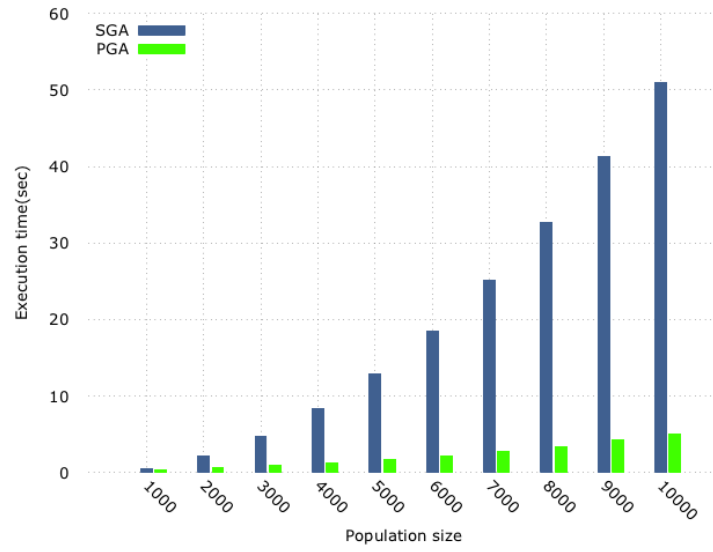


Figure 4.11: Experiment 6 - Runtime comparison SGA vs. PGA v3.

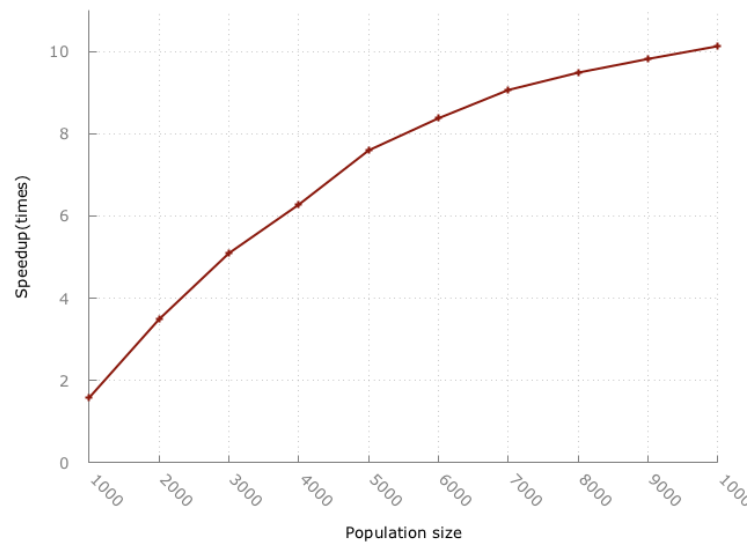


Figure 4.12: Experiment 6 - Speedup graph SGA vs. PGA v3.

Discussion

With this experiment we can see there is a consistent speedup gain with PGA v3 compared to the SGA program as population size is increased. Even though the outcome of this experiment is as expected, the results are taken as speculative until further experiments are done using different parameters. Profiling of this program also needs to be done to see a consistent and reliable outcome.

Chapter 5

Conclusions

This project was set out to implement a parallel GA using MPI and profiling the performance issue over a sequential version by varying the number of parallel processes and population size. Initially, we expected to see a substantial performance gain by parallelising the SGA. By implementing the Parallel GA many underlying issues and techniques were discovered that were unknown to me previously.

In MPI the communication cost is major concern. If a program is not carefully designed to minimise the communication overhead the overall performance can suffer a lot. As we have discovered, there are three very different models of parallel GA. Out of these three only the first model(Master-slave) implemented in this project. The experiments done in this project shown that the traditional master-slave model of PGA using MPI does not result in much performance gain over the sequential SGA because of the communication cost involved for message-passing. In general MPI programs are more suitable for computationally heavy program. If the computational cost is not greater than the communication cost then its performance gain will be negative. The subsequent versions of the PGA (v3), the GA behaviour was changed due to the fact that it is very close to *Island* model. This model of parallel GA needs a migration policy implemented. The time constraint was a major factor for this challenging project. If time permitted, the next step would be to implement a distributed model of Parallel GA with a migration policy.

This project was an exploratory experience in using MPI library and a Cluster computing environment. In this project I have had a great experience in exploring many difficult and exciting concepts in Parallelisation, High Performance Computing(HPC) and some aspect of Genetic Algorithms(GA). I have learned the advantages and limitation of MPI libraries and many underlying challenging issues involved. I gained a lot of experience in setting up a Beowulf cluster, configuring and administration of Linux Server, using remote development environment and shell scripting. Last but not least learned to use L^AT_EX to produce this report.

Bibliography

Akl, S. G. and Nagy, M. (2009), *Introduction to Parallel Computation*, Springer London, London, pp. 43–80.

Atom.me.gatech.edu (2016), ‘Basic beowulf cluster’, [image online] available: <http://atom.me.gatech.edu/scyld-doc/users-guide/images/NetTopSimple.png>. [accessed: 2 January 2017].

Barney, B. (2016a), ‘Introduction to parallel computing’, [online] available: https://computing.llnl.gov/tutorials/parallel_comp/. [accessed: 4 October, 2016].

Barney, B. (2016b), ‘Message passing interface (mpi)’, [online] available: <https://computing.llnl.gov/tutorials/mpi/>. [accessed: 23 November 2016].

Barney, B. (2016c), ‘Openmp’, [online] available: <https://computing.llnl.gov/tutorials/openMP/>. [accessed: 4 October 2016].

Beowulf.org (2016), ‘Beowulf project overview’, [online] available: <http://www.beowulf.org/overview/faq.html>. [accessed: 28 December 2016].

Burkardt, J. (2017), ‘https://people.sc.fsu.edu/~jburkardt/cpp_src/simple_ga/simple_ga.cpp’, https://people.sc.fsu.edu/~jburkardt/cpp_src/simple_ga/simple_ga.cpp. (Accessed on 28/01/2017).

Cantú-Paz, E. (1998), ‘A survey of parallel genetic algorithms’, *Calculateurs paralleles, reseaux et systems repartis* **10**(2), 141–171.

- Chapman, B., Jost, G. and Van Der Pas, R. (2008), *Using OpenMP: portable shared memory parallel programming*, Vol. 10, MIT press.
- Cox, D. R. and Hinkley, D. V. (1979), *Theoretical statistics*, CRC Press.
- Dagum, L. and Menon, R. (1998), ‘Openmp: An industry standard api for shared-memory programming’, *IEEE Computational Science & Engineering* **5**(1), 46–55.
- Dartmouth College (2011), ‘Pros and cons of openmp and mpi’, [online] available: https://www.dartmouth.edu/~rc/classes/intro_mpi/parallel_prog_compare.html. [accessed: 28 December 2016].
- Eaton, M. (2015), Introduction, in ‘Evolutionary Humanoid Robotics’, Springer, pp. 1–7.
- Goldberg, D. E. (1989), *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Massachusetts.
- Gropp, W., Lusk, E., Doss, N. and Skjellum, A. (1996), ‘A high-performance, portable implementation of the mpi message passing interface standard’, *Parallel computing* **22**(6), 789–828.
- Gropp, W., Lusk, E. and Skjellum, A. (1999), *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Massachusetts.
- Gropp, W., Lusk, E. and Sterling, T. (2003), *Beowulf Cluster Computing with Linux*, 2nd edn, The MIT Press, Massachusetts.
- Gustafson, J. L. (1988), ‘Reevaluating amdahl’s law’, *Communications of the ACM* **31**(5), 532–533.
- Kim, N. S., Austin, T., Baauw, D., Mudge, T., Flautner, K., Hu, J. S., Irwin, M. J., Kandemir, M. and Narayanan, V. (2003), ‘Leakage current: Moore’s law meets static power’, *computer* **36**(12), 68–75.

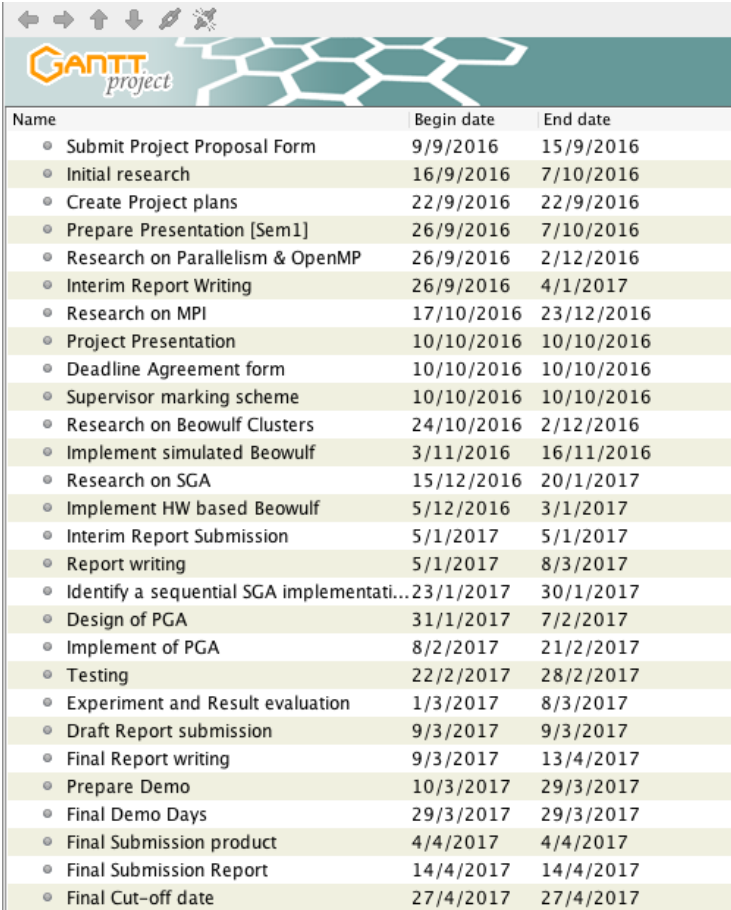
- Konfrst, Z. (2004), Parallel genetic algorithms: Advances, computing trends, applications and perspectives, *in* ‘Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International’, IEEE, p. 162.
- Luque, G., Alba, E. and Dorronsoro, B. (2005), ‘Parallel genetic algorithms’, *Parallel metaheuristics: A new class of algorithms* pp. 107–126.
- Michalewicz, Z. (1996), *Genetic Algorithms + Data Structures = Evolution Programs (3rd Ed.)*, Springer-Verlag, London, UK, UK.
- Mpi-forum.org (1995), ‘Mpi: A message-passing interface standard - type matching rules’, [online] available: <http://mpi-forum.org/docs/mpi-1.1/mpi-11-html/node37.html>. [accessed: 28 December 2016].
- Mpi-forum.org (2016), ‘Mpi: A message-passing interface standard - type matching rules’, [online] available: <http://mpi-forum.org/>. [accessed: 28 December 2016].
- Murphy, R. (2003), ‘A generic parallel genetic algorithm’, https://computing.llnl.gov/tutorials/parallel_comp/.
- Obitko, M. (1998), ‘Genetic algorithm description - introduction to genetic algorithms’, [online] available: <http://www.obitko.com/tutorials/genetic-algorithms/ga-basic-description.php>. [accessed: 2 January 2017].
- Perkowski, M. A. (2010), ‘Introduction to parallel programming’, [online] available: http://web.cecs.pdx.edu/~mperkows/CLASS_VHDL_99/050.Introduction-to-Parallel-Computing.pdf. [accessed: 28 December 2016].
- Pit, L. J. (1995), ‘Parallel genetic algorithms’, *Department of Computer Science, Master Thesis, Leiden University, Holanda*.
- RTC Magazine (2013), ‘Using opencl for network acceleration | rtc magazine’, [online] available: <http://rtcmagazine.com/articles/view/103209>. (Accessed on 03/12/2016).

- Schaller, R. R. (1997), ‘Moore’s law: past, present and future’, *IEEE spectrum* **34**(6), 52–59.
- Squyres, J. M., Willcock, J. J., McCandless, B. C., Rijks, P. W. and Lumsdaine, A. (1996), ‘Object oriented mpi (oompi): A c++ class library for mpi version 1.0. 3’.
- Sutter, H. (2005), ‘The free lunch is over: A fundamental turn toward concurrency in software’, [online] available: <http://www.gotw.ca/publications/concurrency-ddj.htm>. [accessed: 28 December 2016].
- Thiruvathukal, G. K. (2005), ‘Guest editors’ introduction: Cluster computing’, *Computing in Science Engineering* **7**(2), 11–13.
- Top500.org (2017), ‘List statistics | top500 supercomputer sites’, [online] available: <https://www.top500.org/statistics/list/>. [accessed: 2 January 2017].
- Tutorialspoint.com (2017), ‘Genetic algorithms’, [online] available: https://www.tutorialspoint.com/genetic_algorithms/index.htm. [accessed: 2 January 2017].
- Wikipedia user: A1 (2007), ‘Illustration of the fork-join model of parallel programming.’, [online] available: https://en.wikipedia.org/wiki/File:Fork_join.svg. [accessed: December 21, 2016].

Appendices

Appendix A

Project plans



The screenshot shows the Gantt Project software interface. At the top, there is a toolbar with icons for navigation and editing. Below the toolbar is the 'GANTT project' logo. The main area displays a list of project tasks in a table format. Each task is preceded by a small circular icon. The table has three columns: 'Name', 'Begin date', and 'End date'. The tasks are listed in chronological order from top to bottom.

Name	Begin date	End date
• Submit Project Proposal Form	9/9/2016	15/9/2016
• Initial research	16/9/2016	7/10/2016
• Create Project plans	22/9/2016	22/9/2016
• Prepare Presentation [Sem1]	26/9/2016	7/10/2016
• Research on Parallelism & OpenMP	26/9/2016	2/12/2016
• Interim Report Writing	26/9/2016	4/1/2017
• Research on MPI	17/10/2016	23/12/2016
• Project Presentation	10/10/2016	10/10/2016
• Deadline Agreement form	10/10/2016	10/10/2016
• Supervisor marking scheme	10/10/2016	10/10/2016
• Research on Beowulf Clusters	24/10/2016	2/12/2016
• Implement simulated Beowulf	3/11/2016	16/11/2016
• Research on SGA	15/12/2016	20/1/2017
• Implement HW based Beowulf	5/12/2016	3/1/2017
• Interim Report Submission	5/1/2017	5/1/2017
• Report writing	5/1/2017	8/3/2017
• Identify a sequential SGA implementati...	23/1/2017	30/1/2017
• Design of PGA	31/1/2017	7/2/2017
• Implement of PGA	8/2/2017	21/2/2017
• Testing	22/2/2017	28/2/2017
• Experiment and Result evaluation	1/3/2017	8/3/2017
• Draft Report submission	9/3/2017	9/3/2017
• Final Report writing	9/3/2017	13/4/2017
• Prepare Demo	10/3/2017	29/3/2017
• Final Demo Days	29/3/2017	29/3/2017
• Final Submission product	4/4/2017	4/4/2017
• Final Submission Report	14/4/2017	14/4/2017
• Final Cut-off date	27/4/2017	27/4/2017

Figure A.1: Gantt Project plans - List

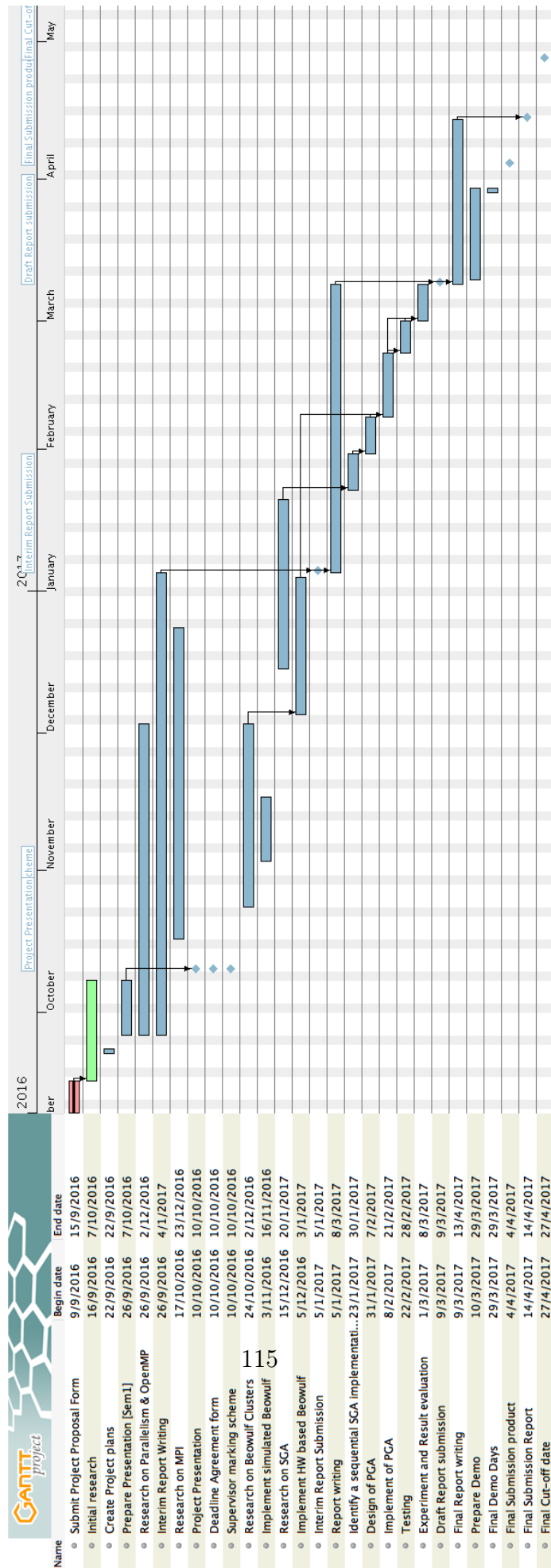
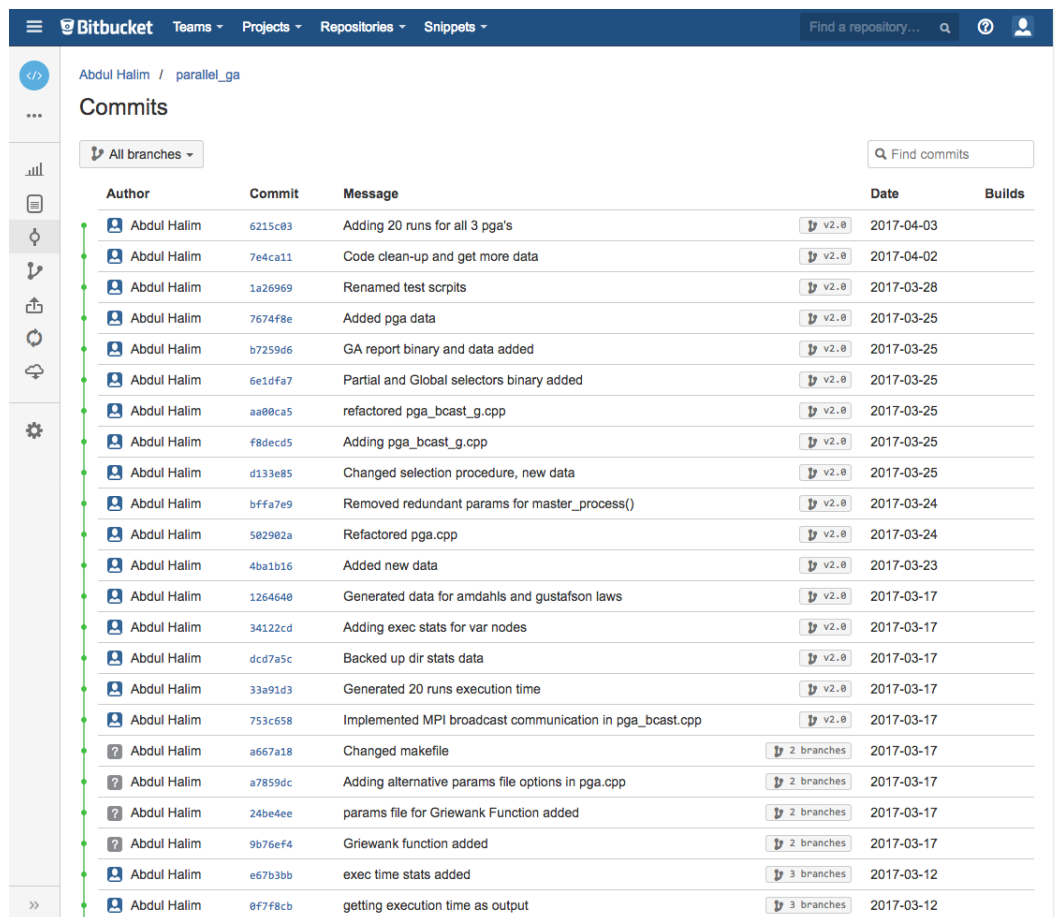


Figure A.2: Gantt Project plans - Graph

Appendix B

Git logs



Abdul Halim / parallel_ga					
Commits					
All branches					Find commits
Author	Commit	Message		Date	Builds
Abdul Halim	6215c83	Adding 20 runs for all 3 pga's	v2.0	2017-04-03	
Abdul Halim	7e4ca11	Code clean-up and get more data	v2.0	2017-04-02	
Abdul Halim	1a26969	Renamed test scripts	v2.0	2017-03-28	
Abdul Halim	7674f8e	Added pga data	v2.0	2017-03-25	
Abdul Halim	b7259d6	GA report binary and data added	v2.0	2017-03-25	
Abdul Halim	6e1dfa7	Partial and Global selectors binary added	v2.0	2017-03-25	
Abdul Halim	aa80ca5	refactored pga_bcast_g.cpp	v2.0	2017-03-25	
Abdul Halim	f8dec5	Adding pga_bcast_g.cpp	v2.0	2017-03-25	
Abdul Halim	d133e85	Changed selection procedure, new data	v2.0	2017-03-25	
Abdul Halim	bffa7e9	Removed redundant params for master_process()	v2.0	2017-03-24	
Abdul Halim	582982a	Refactored pga.cpp	v2.0	2017-03-24	
Abdul Halim	4ba1b16	Added new data	v2.0	2017-03-23	
Abdul Halim	1264640	Generated data for amdahls and gustafson laws	v2.0	2017-03-17	
Abdul Halim	34122cd	Adding exec stats for var nodes	v2.0	2017-03-17	
Abdul Halim	dcd7a5c	Backed up dir stats data	v2.0	2017-03-17	
Abdul Halim	33a91d3	Generated 20 runs execution time	v2.0	2017-03-17	
Abdul Halim	753c658	Implemented MPI broadcast communication in pga_bcast.cpp	v2.0	2017-03-17	
Abdul Halim	a667a18	Changed makefile	2 branches	2017-03-17	
Abdul Halim	a7859dc	Adding alternative params file options in pga.cpp	2 branches	2017-03-17	
Abdul Halim	24be4ee	params file for Griewank Function added	2 branches	2017-03-17	
Abdul Halim	9b76ef4	Griewank function added	2 branches	2017-03-17	
Abdul Halim	e67b3bb	exec time stats added	3 branches	2017-03-12	
Abdul Halim	0f7f8cb	getting execution time as output	3 branches	2017-03-12	

Figure B.1: PGA Version control logs.

Appendix C

Project source codes

C.1 params.hpp

```
1 #ifndef PARAMS_HPP
2 #define DEFINES_HPP
3 //
4 //  Change any of these parameters to match your needs
5 //
6 #ifndef DEFINED_BY_MAKE
7 # define POPSIZE 100
8 # define MAXGENS 100
9 #endif /* DEFINED_BY_MAKE */
10
11 # define NVARs 3
12 # define PXOVER 0.8
13 # define PMUTATION 0.15
14 # define F1 1
15 # define F8 8
16 # define FITNESS_F F8
17
18 #endif /* PARAMS_HPP */
```

C.2 sga_proc.hpp

```
1 #ifndef SGA_PROC_HPP
```

```

2 struct genotype
3 {
4     double gene[NVARS];
5     double fitness;
6     double upper[NVARS];
7     double lower[NVARS];
8     double rfitness;
9     double cfitness;
10 };
11
12 extern struct genotype population[];
13 extern struct genotype newpopulation[];
14
15 void crossover ( int &seed );
16 void elitist ( );
17 void evaluate ( );
18 int i4_uniform_ab ( int a, int b, int &seed );
19 void initialize ( string filename, int &seed );
20 void keep_the_best ( );
21 void mutate ( int &seed );
22 double r8_uniform_ab ( double a, double b, int &seed );
23 void report ( int generation );
24 void selector ( int &seed );
25 void timestamp ( );
26 void Xover ( int one, int two, int &seed );
27 void f1 ( );
28 void f8 ( );
29 #endif /* SGA_PROC_HPP */

```

C.3 sga.cpp

```

1 # include <cstdlib>
2 # include <iostream>
3 # include <iomanip>
4 # include <fstream>
5 # include <iomanip>
6 # include <cmath>
7 # include <ctime>
8 # include <cstring>

```

```

9 # include "params.hpp"
10 # include "sga_proc.hpp"
11
12 using namespace std;
13
14 struct genotype population[POPSIZE+1];
15 struct genotype newpopulation[POPSIZE+1];
16
17 int main ( );
18
19 //*****
20
21 int main ( )
22 {
23     string filename;
24
25     switch(FITNESS_F) {
26         case F1:
27             filename = "simple_ga_input.txt";
28             break;
29         case F8:
30             filename = "simple_ga_input2.txt";
31             break;
32         default:
33             filename = "simple_ga_input.txt";
34             break;
35     }
36
37     int generation;
38     int i;
39     int seed;
40
41     timestamp ( );
42     cout << "\n";
43     cout << "SIMPLE_GA:\n";
44     cout << "  C++ version\n";
45     cout << "  A simple example of a genetic algorithm.\n";
46
47     if ( NVARs < 2 )

```

```

48 {
49     cout << "\n";
50     cout << "  The crossover modification will not be available,\n
";
51     cout << "  since it requires 2 <= NVARs.\n";
52 }
53
54 seed = 123456789;
55
56 initialize ( filename , seed );
57
58 evaluate ( );
59
60 keep_the_best ( );
61
62 for ( generation = 0; generation < MAXGENS; generation++ )
63 {
64     selector ( seed );
65     crossover ( seed );
66     mutate ( seed );
67     report ( generation );
68     evaluate ( );
69     elitist ( );
70 }
71
72 cout << "\n";
73 cout << "  Best member after " << MAXGENS << " generations:\n";
74 cout << "\n";
75
76 for ( i = 0; i < NVARs; i++ )
77 {
78     cout << "    var(" << i << ") = " << population[POPSIZE].gene[i]
<< "\n";
79 }
80
81 cout << "\n";
82 cout << "  Best fitness = " << population[POPSIZE].fitness << "\n
";
83 //

```



```

84 //  Terminate .
85 //
86     cout << "\n";
87     cout << "SIMPLE_GA:\n";
88     cout << "  Normal end of execution.\n";
89     cout << "\n";
90     timestamp ( );
91
92     return 0;
93 }

```

C.4 pga_proc.hpp

```

1 # include "params.hpp"
2 # include "sga_proc.hpp"
3
4 #ifndef PGA_PROC_HPP
5
6 #define MASTER_ID 0
7
8 /* MPI message tags */
9 #define SEND_DATA_TAG 2001
10 #define RETN_DATA_TAG 2002
11 #define CONT_TAG 2003
12 #define PART_INDEX_TAG 3001
13 #define BEST_WORST_TAG 4001
14
15 enum boolean {FALSE, TRUE};
16
17 typedef struct
18 {
19     double gene[NVARS];
20     double fitness;
21     double upper[NVARS];
22     double lower[NVARS];
23     double rfitness;
24     double cfitness;
25 } genotype;
26 extern genotype population[];

```

```

27 extern newpopulation [];
28
29 /* MPI Global variables */
30 extern int myId, senderId, receiverId, numRows, numNodes,
    numRowsToReceive, ierr,
31     avgRowsPerNode, numRowsReceived, startRow, endRow,
32     numRowsToSend;
33 extern MPI_Status status;
34
35 void master_process(string& filename, int &seed,
36     MPI_Datatype &mpi_genotype, int node_data[][2],
37     clock_t start);
38 void slave_process(int &seed, MPI_Datatype& mpi_genotype,
39     int node_data[][2]);
40 void create_mpi_genotype_struct(MPI_Datatype& mpi_genotype);
41 void evaluate (int my_start, int row_count);
42 void f1(int my_start, int row_count);
43 void f8(int my_start, int row_count);
44 void node_selector(int my_start, int row_count, int &seed);
45 void node_crossover(int my_start, int row_count, int &seed);
46 void node_mutate(int my_start, int row_count, int &seed);
47 void node_evaluate(int my_start, int row_count);
48 void node_Xover (int one, int two, int &seed);
49 void print_population();
50 #endif /* PGA_PROC_HPP */

```

C.5 pga.cpp

```

1 #include <cstdlib>
2 #include <iostream>
3 #include <iomanip>
4 #include <fstream>
5 #include <iomanip>
6 #include <cmath>
7 #include <ctime>
8 #include <cstring>
9 #include <mpi.h>
10 #include "params.hpp"
11 #include "sga_proc.hpp"

```

```

12 # include "pga_proc.hpp"
13
14 using namespace std;
15
16 genotype population[POPSIZE+1];
17 genotype newpopulation[POPSIZE+1];
18
19 /* MPI Global variables */
20 int myId, senderId, receiverId, numRows, numNodes,
    numRowsToReceive, ierr,
21     avgRowsPerNode, numRowsReceived, startRow, endRow,
22     numRowsToSend;
23 MPI_Status status;
24
25 // *****
26 int main(int argc, char** argv)
27 {
28
29     string filename;
30
31     switch(FITNESS_F) {
32     case F1:
33         filename = "simple_ga_input.txt";
34         break;
35     case F8:
36         filename = "simple_ga_input2.txt";
37         break;
38     default:
39         filename = "simple_ga_input.txt";
40         break;
41     }
42
43     int seed;
44
45     /* Create MPI_Type for genotype struct */
46     MPI_Datatype mpi_genotype;
47     create_mpi_genotype_struct(mpi_genotype);
48
49     seed = 123456789;

```

```

50
51     /* Create a data structure to store data partition values for
each nodes
52     *
53     * nodes_data[node][0]: start_index
54     * nodes_data[node][1]: rowCount
55     */
56     int nodes_data[numNodes][2];
57     /* Calculate and store start-end index for each nodes */
58     int num_chunk = numNodes;
59     int start_i, rowCount;
60     int chunk_size = (int)(ceil((double)POPSIZE / num_chunk));
61
62     for(int i = 0; i < num_chunk; i++)
63     {
64         start_i = i * chunk_size;
65         rowCount = min(chunk_size, POPSIZE - start_i);
66         nodes_data[i][0] = start_i;
67         nodes_data[i][1] = rowCount;
68     }
69
70     if(myId == MASTER_ID) // Master Process
71     {
72         master_process(filename, seed, mpi_genotype, nodes_data,
start);
73     }
74     else // Slave processes
75     {
76         slave_process(seed, mpi_genotype, nodes_data);
77     }
78
79     /* Free MPI Datatype genotype */
80     MPI_Type_free(&mpi_genotype);
81
82     /* Terminate MPI */
83     MPI_Finalize();
84     return 0;
85 }

```

C.6 pgav2.cpp

```
1 # include <cstdlib>
2 # include <iostream>
3 # include <iomanip>
4 # include <fstream>
5 # include <iomanip>
6 # include <cmath>
7 # include <ctime>
8 # include <cstring>
9 # include <mpi.h>
10 # include <stddef.h>
11 # include "params.hpp"
12 # include "sga_proc.hpp"
13 # include "pga_proc.hpp"
14
15 using namespace std;
16
17 genotype population[POPSIZE+1];
18 genotype newpopulation[POPSIZE+1];
19
20
21 /* MPI Global variables */
22 int myId, senderId, receiverId, numRows, numNodes,
    numRowsToReceive, ierr,
23     avgRowsPerNode, numRowsReceived, startRow, endRow,
24     numRowsToSend;
25 MPI_Status status;
26
27 // *****
28 int main(int argc, char** argv)
29 {
30     clock_t start, end;
31     double cpu_time_used;
32     start = clock();
33
34     char hostname[MPI_MAX_PROCESSOR_NAME];
35     int length; // To store character length when required
36     ierr = MPI_Init(&argc, &argv);
```

```

37     ierr = MPI_Comm_size(MPI_COMM_WORLD, &numNodes); // How many
nodes
38     ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myId); // My Rank ID
39
40     string filename;
41
42     switch(FITNESS_F) {
43         case F1:
44             filename = "simple_ga_input.txt";
45             break;
46         case F8:
47             filename = "simple_ga_input2.txt";
48             break;
49         default:
50             filename = "simple_ga_input.txt";
51             break;
52     }
53
54     int generation;
55     int i;
56     int seed;
57
58     /* Create MPI_Type for genotype struct */
59     MPI_Datatype mpi_genotype;
60     create_mpi_genotype_struct(mpi_genotype);
61
62
63     seed = 123456789;
64
65
66     /* Initialise first gen population in master process */
67
68     if(myId == MASTER_ID) // Master Process
69     {
70         timestamp ( );
71         cout << "\n";
72         cout << "PARALLEL_GA:\n";
73         cout << " A simple example of a genetic algorithm.\n";
74

```

```

75         if ( NVARs < 2 )
76         {
77             cout << "\n";
78             cout << " The crossover modification will not be
available,\n";
79             cout << " since it requires 2 <= NVARS.\n";
80         }
81
82         initialize ( filename , seed );
83         evaluate (0, POPSIZE );
84         keep_the_best ( );
85     }
86
87     /* Sync message - 1 */
88     MPI_Barrier(MPI_COMM_WORLD);
89
90     /* Broadcast entire first generation from master process */
91     MPI_Bcast(population , POPSIZE + 1, mpi_genotype ,
92             MASTER_ID, MPI_COMM_WORLD);
93
94     /* Sync message - 2 */
95     MPI_Barrier(MPI_COMM_WORLD);
96
97
98     /* Calculate and store start-end index for each nodes */
99     int nodes_data[numNodes][2];
100     int start_i , num_rows;
101     int chunk_size = (int)(ceil((double) POPSIZE / numNodes));
102
103
104     for(int i = 0; i < numNodes; i++)
105     {
106         start_i = i * chunk_size;
107         num_rows = min(chunk_size , POPSIZE - start_i);
108         nodes_data[i][0] = start_i;
109         nodes_data[i][1] = num_rows;
110     }
111
112

```

```

113  /* Each process does GA ops here on individual data parts */
114
115  int my_start = nodes_data[myId][0];
116  int row_count = nodes_data[myId][1];
117  for ( generation = 0; generation < MAXGENS; generation++ )
118  {
119      node_selector (my_start, row_count, seed );
120      crossover ( seed );
121      node_mutate ( my_start, row_count, seed );
122      evaluate (my_start, row_count);
123
124      MPI_Barrier(MPI_COMM_WORLD);
125
126      for(int i = 0; i < numNodes; i++)
127      {
128          MPI_Bcast(&population[nodes_data[i][0]], nodes_data[i
129  ][1],
130                  mpi_genotype, i, MPI_COMM_WORLD);
131          MPI_Barrier(MPI_COMM_WORLD);
132      }
133
134      MPI_Barrier(MPI_COMM_WORLD);
135
136      if(myId == MASTER_ID) // Master Process
137      {
138          report(generation);
139          elitist ( );
140      }
141
142      MPI_Barrier(MPI_COMM_WORLD);
143  }
144
145  MPI_Barrier(MPI_COMM_WORLD);
146
147  /* Master process prints final report */
148
149  if(myId == MASTER_ID) // Master Process
150  {
151      cout << "\n";

```



```

151     cout << "    Best member after " << MAXGENS << " generations
      :\n";
152     cout << "\n";
153
154     for ( i = 0; i < NVAR; i++ )
155     {
156         cout << "    var(" << i << ") = " << population[POPSIZE].
gene[i] << "\n";
157     }
158
159     cout << "\n";
160     cout << "    Best fitness = " << population[POPSIZE].fitness
<< "\n";
161
162     cout << "\n";
163     cout << "SIMPLE_GA:\n";
164     cout << "    Normal end of execution.\n";
165     cout << "\n";
166     timestamp ( );
167
168     end = clock();
169     cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
170     printf("%f\n",cpu_time_used);
171 }
172
173 /* Free MPI Datatype genotype */
174 MPI_Type_free(&mpi_genotype);
175
176 /* Terminate MPI */
177 MPI_Finalize();
178 return 0;
179 }

```

C.7 sga_proc.cpp

```

1 # include "params.hpp"
2 # include "sga_proc.hpp"
3 // *****
4

```

```

5 void crossover ( int &seed )
6
7 // *****
8 //
9 // Purpose:
10 //
11 // CROSSOVER selects two parents for the single point crossover
12 // .
13 // Local parameters:
14 //
15 // Local, int FIRST, is a count of the number of members chosen
16 // .
17 // Parameters:
18 //
19 // Input/output, int &SEED, a seed for the random number
20 // generator.
21 {
22     const double a = 0.0;
23     const double b = 1.0;
24     int mem;
25     int one;
26     int first = 0;
27     double x;
28
29     for ( mem = 0; mem < POPSIZE; ++mem )
30     {
31         x = r8_uniform_ab ( a, b, seed );
32
33         if ( x < PXOVER )
34         {
35             ++first;
36
37             if ( first % 2 == 0 )
38             {
39                 Xover ( one, mem, seed );
40             }

```

```

41     else
42     {
43         one = mem;
44     }
45
46     }
47 }
48 return;
49 }
50 // *****
51
52 void elitist ( )
53
54 // *****
55 //
56 // Purpose:
57 //
58 // ELITIST stores the best member of the previous generation.
59 //
60 // Discussion:
61 //
62 // The best member of the previous generation is stored as
63 // the last in the array. If the best member of the current
64 // generation is worse then the best member of the previous
65 // generation, the latter one would replace the worst member
66 // of the current population.
67 //
68 // Licensing:
69 //
70 // This code is distributed under the GNU LGPL license.
71 //
72 // Local parameters:
73 //
74 // Local, double BEST, the best fitness value.
75 //
76 // Local, double WORST, the worst fitness value.
77 //
78 {
79     int i;

```

```

80  double best;
81  int best_mem;
82  double worst;
83  int worst_mem;
84
85  best = population[0].fitness;
86  worst = population[0].fitness;
87
88  for ( i = 0; i < POPSIZE - 1; ++i )
89  {
90      if ( population[i+1].fitness < population[i].fitness )
91      {
92
93          if ( best <= population[i].fitness )
94          {
95              best = population[i].fitness;
96              best_mem = i;
97          }
98
99          if ( population[i+1].fitness <= worst )
100         {
101             worst = population[i+1].fitness;
102             worst_mem = i + 1;
103         }
104
105     }
106     else
107     {
108
109         if ( population[i].fitness <= worst )
110         {
111             worst = population[i].fitness;
112             worst_mem = i;
113         }
114
115         if ( best <= population[i+1].fitness )
116         {
117             best = population[i+1].fitness;
118             best_mem = i + 1;

```

```

119     }
120
121     }
122
123 }
124 //
125 //  If the best individual from the new population is better than
126 //  the best individual from the previous population , then
127 //  copy the best from the new population; else replace the
128 //  worst individual from the current population with the
129 //  best one from the previous generation
130 //
131 if ( population[POPSIZE].fitness <= best )
132 {
133     for ( i = 0; i < NVAR; i++ )
134     {
135         population[POPSIZE].gene[i] = population[best_mem].gene[i];
136     }
137     population[POPSIZE].fitness = population[best_mem].fitness;
138 }
139 else
140 {
141     for ( i = 0; i < NVAR; i++ )
142     {
143         population[worst_mem].gene[i] = population[POPSIZE].gene[i];
144     }
145     population[worst_mem].fitness = population[POPSIZE].fitness;
146 }
147
148 return;
149 }
150 // *****
151
152 void evaluate ( )
153
154 // *****
155 //
156 //  Purpose :
157 //

```

```

158 //      EVALUATE implements the user-defined valuation function
159 //
160 //      Discussion :
161 //
162 //      Each time this is changed, the code has to be recompiled.
163 //      The current function is:  $x[1]^2 - x[1] * x[2] + x[3]$ 
164 {
165 //      return FITNESS_F == F1 ? f1() : f8();
166     switch(FITNESS_F) {
167         case F1:
168             return f1();
169             break;
170         case F8:
171             return f8();
172             break;
173         default:
174             break;
175     }
176 }
177
178 void f1()
179 {
180     int member;
181     int i;
182     double x[NVARS+1];
183
184     for ( member = 0; member < POPSIZE; member++ )
185     {
186         for ( i = 0; i < NVARS; i++ )
187         {
188             x[i+1] = population[member].gene[i];
189         }
190         population[member].fitness = ( x[1] * x[1] ) - ( x[1] * x[2] )
191         + x[3];
192     }
193     return;
194 }
195 /*

```

```

196  * Griewank Function
197  */
198  void f8 ()
199  {
200      int member;
201      int i;
202      double x[NVARS+1];
203      double part1, part2;
204
205
206      for ( member = 0; member < POPSIZE; member++ )
207      {
208          for ( i = 0; i < NVARS; i++ )
209          {
210              x[i+1] = population[member].gene[i];
211          }
212
213          part1 = 0.0;
214          for(i = 1; i < NVARS; i++)
215          {
216              part1 += pow(x[i], 2) / 4000;
217          }
218
219          part2 = 1.0;
220          for(i = 1; i < NVARS; i++)
221          {
222              part2 *= cos( x[i] / sqrt(i) );
223          }
224
225          population[member].fitness = 1.0 + part1 - part2;
226      }
227      return;
228  }
229  // *****
230
231  int i4_uniform_ab ( int a, int b, int &seed )
232
233  // *****
234  //

```

```

235 // Purpose :
236 //
237 // I4_UNIFORM_AB returns a scaled pseudorandom I4 between A and
    B.
238 //
239 // Discussion :
240 //
241 // The pseudorandom number should be uniformly distributed
242 // between A and B.
243 //
244 // Parameters :
245 //
246 // Input, int A, B, the limits of the interval.
247 //
248 // Input/output, int &SEED, the "seed" value, which should NOT
    be 0.
249 // On output, SEED has been updated.
250 //
251 // Output, int I4_UNIFORM, a number between A and B.
252 //
253 {
254     int c;
255     const int i4_huge = 2147483647;
256     int k;
257     float r;
258     int value;
259
260     if ( seed == 0 )
261     {
262         cerr << "\n";
263         cerr << "I4_UNIFORM_AB - Fatal error!\n";
264         cerr << " Input value of SEED = 0.\n";
265         exit ( 1 );
266     }
267 //
268 // Guarantee A <= B.
269 //
270     if ( b < a )
271     {

```



```

272     c = a;
273     a = b;
274     b = c;
275 }
276
277 k = seed / 127773;
278
279 seed = 16807 * ( seed - k * 127773 ) - k * 2836;
280
281 if ( seed < 0 )
282 {
283     seed = seed + i4_huge;
284 }
285
286 r = ( float ) ( seed ) * 4.656612875E-10;
287 //
288 // Scale R to lie between A-0.5 and B+0.5.
289 //
290 r = ( 1.0 - r ) * ( ( float ) a - 0.5 )
291     +          r      * ( ( float ) b + 0.5 );
292 //
293 // Use rounding to convert R to an integer between A and B.
294 //
295 value = round ( r );
296 //
297 // Guarantee A <= VALUE <= B.
298 //
299 if ( value < a )
300 {
301     value = a;
302 }
303 if ( b < value )
304 {
305     value = b;
306 }
307
308 return value;
309 }
310 // *****

```

```

311
312 void initialize ( string filename , int &seed )
313
314 // *****
315 //
316 // Purpose:
317 //
318 // INITIALIZE initializes the genes within the variables bounds
319 //
320 // Discussion:
321 //
322 // It also initializes (to zero) all fitness values for each
323 // member of the population. It reads upper and lower bounds
324 // of each variable from the input file 'gadata.txt'. It
325 // randomly generates values between these bounds for each
326 // gene of each genotype in the population. The format of
327 // the input file 'gadata.txt' is
328 //
329 //      var1_lower_bound var1_upper bound
330 //      var2_lower_bound var2_upper bound ...
331 //
332 // Parameters:
333 //
334 //      Input, string FILENAME, the name of the input file.
335 //
336 //      Input/output, int &SEED, a seed for the random number
337 //      generator.
338 {
339     int i;
340     ifstream input;
341     int j;
342     double lbound;
343     double ubound;
344
345     input.open ( filename.c_str ( ) );
346
347     if ( !input )

```

```

348 {
349     cerr << "\n";
350     cerr << "INITIALIZE - Fatal error!\n";
351     cerr << "  Cannot open the input file!\n";
352     exit ( 1 );
353 }
354 //
355 //  Initialize variables within the bounds
356 //
357 for ( i = 0; i < NVAR; i++ )
358 {
359     input >> lbound >> ubound;
360
361     for ( j = 0; j < POPSIZE; j++ )
362     {
363         population[j].fitness = 0;
364         population[j].rfitness = 0;
365         population[j].cfitness = 0;
366         population[j].lower[i] = lbound;
367         population[j].upper[i] = ubound;
368         population[j].gene[i] = r8_uniform_ab ( lbound, ubound, seed
369     );
370     }
371 }
372 input.close ( );
373
374 return;
375 }
376 // *****
377
378 void keep_the_best ( )
379
380 // *****
381 //
382 //  Purpose:
383 //
384 //  KEEP_THE_BEST keeps track of the best member of the
    population.

```

```

385 //
386 // Discussion:
387 //
388 // Note that the last entry in the array Population holds a
389 // copy of the best individual.
390 //
391 // Local parameters:
392 //
393 // Local, int CUR_BEST, the index of the best individual.
394 //
395 {
396     int cur_best;
397     int mem;
398     int i;
399
400     cur_best = 0;
401
402     for ( mem = 0; mem < POPSIZE; mem++ )
403     {
404         if ( population[POPSIZE].fitness < population[mem].fitness )
405         {
406             cur_best = mem;
407             population[POPSIZE].fitness = population[mem].fitness;
408         }
409     }
410 //
411 // Once the best member in the population is found, copy the
412 // genes.
413 //
414     for ( i = 0; i < NVAR; i++ )
415     {
416         population[POPSIZE].gene[i] = population[cur_best].gene[i];
417     }
418     return;
419 }
420 // *****
421
422 void mutate ( int &seed )

```

```

423
424 // *****
425 //
426 // Purpose:
427 //
428 //     MUTATE performs a random uniform mutation.
429 //
430 // Discussion:
431 //
432 //     A variable selected for mutation is replaced by a random
433 //     value
434 //     between the lower and upper bounds of this variable.
435 // Parameters:
436 //
437 //     Input/output, int &SEED, a seed for the random number
438 //     generator.
439 //
440 {
441     const double a = 0.0;
442     const double b = 1.0;
443     int i;
444     int j;
445     double lbound;
446     double ubound;
447     double x;
448
449     for ( i = 0; i < POPSIZE; i++ )
450     {
451         for ( j = 0; j < NVAR; j++ )
452         {
453             x = r8_uniform_ab ( a, b, seed );
454             if ( x < PMUTATION )
455             {
456                 lbound = population[i].lower[j];
457                 ubound = population[i].upper[j];
458                 population[i].gene[j] = r8_uniform_ab ( lbound, ubound,
459 seed );
460             }
461         }
462     }
463 }

```

```

459     }
460 }
461
462 return ;
463 }
464 // *****
465
466 double r8_uniform_ab ( double a, double b, int &seed )
467 // *****
468 //
469 // Purpose :
470 //
471 // R8_UNIFORM_AB returns a scaled pseudorandom R8.
472 //
473 // Discussion :
474 //
475 // The pseudorandom number should be uniformly distributed
476 // between A and B.
477 //
478 // Parameters :
479 //
480 // Input , double A, B, the limits of the interval.
481 //
482 // Input/output , int &SEED, the "seed" value , which should NOT
    be 0.
483 // On output , SEED has been updated.
484 //
485 // Output , double R8_UNIFORM_AB, a number strictly between A
    and B.
486 //
487 {
488     int i4_huge = 2147483647;
489     int k;
490     double value;
491
492     if ( seed == 0 )
493     {
494         cerr << "\n";
495         cerr << "R8_UNIFORM_AB - Fatal error!\n";

```

```

496     cerr << "   Input value of SEED = 0.\n";
497     exit ( 1 );
498 }
499
500 k = seed / 127773;
501
502 seed = 16807 * ( seed - k * 127773 ) - k * 2836;
503
504 if ( seed < 0 )
505 {
506     seed = seed + i4_huge;
507 }
508
509 value = ( double ) ( seed ) * 4.656612875E-10;
510
511 value = a + ( b - a ) * value;
512
513 return value;
514 }
515 // *****
516
517 void report ( int generation )
518
519 // *****
520 //
521 //   Purpose:
522 //
523 //       REPORT reports progress of the simulation.
524 //
525 //
526 //   Local parameters:
527 //
528 //       Local, double avg, the average population fitness.
529 //
530 //       Local, best_val, the best population fitness.
531 //
532 //       Local, double square_sum, square of sum for std calc.
533 //
534 //       Local, double stddev, standard deviation of population

```

```

        fitness.
535 //
536 //     Local, double sum, the total population fitness.
537 //
538 //     Local, double sum_square, sum of squares for std calc.
539 //
540 {
541     double avg;
542     double best_val;
543     int i;
544     double square_sum;
545     double stddev;
546     double sum;
547     double sum_square;
548
549     if ( generation == 0 )
550     {
551         cout << "\n";
552         cout << "    Generation          Best          Average
Standard \n";
553         cout << "    number          value          fitness
deviation \n";
554         cout << "\n";
555     }
556
557     sum = 0.0;
558     sum_square = 0.0;
559
560     for ( i = 0; i < POPSIZE; i++ )
561     {
562         sum = sum + population[i].fitness;
563         sum_square = sum_square + population[i].fitness * population[i]
].fitness;
564     }
565
566     avg = sum / ( double ) POPSIZE;
567     square_sum = avg * avg * POPSIZE;
568     stddev = sqrt ( ( sum_square - square_sum ) / ( POPSIZE - 1 ) );
569     best_val = population[POPSIZE].fitness;

```



```

570
571     cout << "    " << setw(8) << generation
572           << "    " << setw(14) << best_val
573           << "    " << setw(14) << avg
574           << "    " << setw(14) << stddev << "\n";
575
576     return;
577 }
578 //*****
579
580 void selector ( int &seed )
581
582 //*****
583 //
584 //  Purpose:
585 //
586 //      SELECTOR is the selection function.
587 //
588 //  Discussion:
589 //
590 //      Standard proportional selection for maximization problems
591 //      incorporating
592 //      the elitist model. This makes sure that the best member
593 //      always survives.
594 //
595 //  Parameters:
596 //
597 //      Input/output, int &SEED, a seed for the random number
598 //      generator.
599 //
600 {
601     const double a = 0.0;
602     const double b = 1.0;
603     int i;
604     int j;
605     int mem;
606     double p;
607     double sum;
608 //

```

```

606 // Find the total fitness of the population.
607 //
608 sum = 0.0;
609 for ( mem = 0; mem < POPSIZE; mem++ )
610 {
611     sum = sum + population[mem].fitness;
612 }
613 //
614 // Calculate the relative fitness of each member.
615 //
616 for ( mem = 0; mem < POPSIZE; mem++ )
617 {
618     population[mem].rfitness = population[mem].fitness / sum;
619 }
620 //
621 // Calculate the cumulative fitness.
622 //
623 population[0].cfitness = population[0].rfitness;
624 for ( mem = 1; mem < POPSIZE; mem++ )
625 {
626     population[mem].cfitness = population[mem-1].cfitness +
627         population[mem].rfitness;
628 }
629 //
630 // Select survivors using cumulative fitness.
631 //
632 for ( i = 0; i < POPSIZE; i++ )
633 {
634     p = r8_uniform_ab ( a, b, seed );
635     if ( p < population[0].cfitness )
636     {
637         newpopulation[i] = population[0];
638     }
639     else
640     {
641         for ( j = 0; j < POPSIZE; j++ )
642         {
643             if ( population[j].cfitness <= p && p < population[j+1].
cfitness )

```

```

644         {
645             newpopulation[i] = population[j+1];
646         }
647     }
648 }
649 }
650 //
651 //  Overwrite the old population with the new one.
652 //
653 for ( i = 0; i < POPSIZE; i++ )
654 {
655     population[i] = newpopulation[i];
656 }
657
658 return;
659 }
660 // *****
661
662 void timestamp ( )
663
664 // *****
665 //
666 //  Purpose:
667 //
668 //      TIMESTAMP prints the current YMDHMS date as a time stamp.
669 //
670 {
671 # define TIME_SIZE 40
672
673     static char time_buffer[TIME_SIZE];
674     const struct tm *tm;
675     size_t len;
676     time_t now;
677
678     now = time ( NULL );
679     tm = localtime ( &now );
680
681     len = strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p",
682                     tm );

```

```

682
683     cout << time_buffer << "\n";
684
685     return;
686 # undef TIME_SIZE
687 }
688 // *****
689
690 void Xover ( int one, int two, int &seed )
691
692 // *****
693 //
694 //   Purpose:
695 //
696 //       XOVER performs crossover of the two selected parents.
697 //
698 //   Local parameters:
699 //
700 //       Local, int point, the crossover point.
701 //
702 //   Parameters:
703 //
704 //       Input, int ONE, TWO, the indices of the two parents.
705 //
706 //       Input/output, int &SEED, a seed for the random number
707 //       generator.
708 //
709 {
710     int i;
711     int point;
712     double t;
713 //
714 //   Select the crossover point.
715 //
716 point = i4_uniform_ab ( 0, NVARS - 1, seed );
717 //   Swap genes in positions 0 through POINT-1.
718 //
719 for ( i = 0; i < point; i++ )

```

```

720 {
721     t                                = population[one].gene[i];
722     population[one].gene[i] = population[two].gene[i];
723     population[two].gene[i] = t;
724 }
725
726 return;
727 }

```

C.8 pga_proc.cpp

```

1 # include <mpi.h>
2 # include "params.hpp"
3 # include "sga_proc.hpp"
4 # include "pga_proc.hpp"
5
6 // *****
7 void slave_process(int &seed, MPI_Datatype& mpi_genotype,
8                     int nodes_data[][2])
9 {
10     int my_start = nodes_data[myId][0];
11     int row_count = nodes_data[myId][1];
12     for(int generation = 0; generation < MAXGENS; generation++)
13     {
14         /* Get updated subset of population from master */
15         ierr = MPI_Recv(&population[my_start], row_count,
16                        mpi_genotype, MASTER_ID,
17                        SEND_DATA_TAG, MPI_COMM_WORLD, &status);
18         /* Evaluate */
19         evaluate (my_start, row_count);
20
21         /* Send evaluated population to master */
22         MPI_Send(&population[my_start], row_count, mpi_genotype,
23                MASTER_ID,
24                RETN_DATA_TAG, MPI_COMM_WORLD);
25     }
26 }

```

```

27 void master_process(string& filename , int &seed ,
28     MPI_Datatype &mpi_genotype , int nodes_data[][2] , clock_t
    start)
29 {
30     timestamp ( );
31     cout << "\n";
32     cout << "PARALLEL_GA:\n";
33     cout << "  C++ version\n";
34     cout << "  A simple example of a parellel genetic algorithm.\n"
    ;
35
36     if ( NVARs < 2 )
37     {
38         cout << "\n";
39         cout << "  The crossover modification will not be available
    ,\n";
40         cout << "  since it requires 2 <= NVARS.\n";
41     }
42
43     initialize ( filename , seed );
44
45     evaluate (0, POPSIZE );
46
47     keep_the_best ( );
48
49     /* Run GA ops for given number of generation */
50     for(int generation = 0; generation < MAXGENS; generation++)
51     {
52         selector ( seed );
53         crossover ( seed );
54         mutate ( seed );
55         /* Master generate report */
56         report(generation);
57
58         /* Update all nodes with it's part of current population
    */
59         int start_index , item_count;
60         for(int nodeId = 1; nodeId < numNodes; nodeId++)
61         {

```

```

62         start_index = nodes_data[nodeId][0];
63         item_count = nodes_data[nodeId][1];
64         MPI_Send(&population[start_index], item_count,
mpi_genotype, nodeId, SEND_DATA_TAG,
65                 MPI_COMM_WORLD);
66     }
67
68     /* Evaluate master's parts of the population */
69     evaluate(nodes_data[0][0], nodes_data[0][1]);
70
71     /* Collect all parts of new population from each node */
72     for(int nodeId = 1; nodeId < numNodes; nodeId++)
73     {
74         start_index = nodes_data[nodeId][0];
75         item_count = nodes_data[nodeId][1];
76         MPI_Recv(&population[start_index], item_count,
mpi_genotype,
77                 nodeId, REIN_DATA_TAG, MPI_COMM_WORLD, &status
);
78     }
79
80     elitist();
81 }
82
83 cout << "\n";
84 cout << "    Best member after " << MAXGENS << " generations:\n";
85 cout << "\n";
86
87 for ( int i = 0; i < NVAR; i++ )
88 {
89     cout << "    var(" << i << ") = " << population[POPSIZE].gene
[i] << "\n";
90 }
91
92 cout << "\n";
93 cout << "    Best fitness = " << population[POPSIZE].fitness << "
\n";
94 //
95 //    Terminate.

```

```

96 //
97 cout << "\n";
98 cout << "PARALLEL_GA:\n";
99 cout << "   Normal end of execution.\n";
100 cout << "\n";
101 timestamp ( );
102
103 }
104 void create_mpi_genotype_struct(MPI_Datatype& mpi_genotype){
105     const int nitems = 6;
106     int      blocklengths[6] = {NVAR, 1, NVAR, NVAR, 1, 1};
107     MPI_Datatype types[6] = {MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE,
108     MPI_DOUBLE,
109     MPI_DOUBLE, MPI_DOUBLE};
110     MPI_Aint      offsets[6];
111     offsets[0] = offsetof(genotype, gene);
112     offsets[1] = offsetof(genotype, fitness);
113     offsets[2] = offsetof(genotype, upper);
114     offsets[3] = offsetof(genotype, lower);
115     offsets[4] = offsetof(genotype, rf fitness);
116     offsets[5] = offsetof(genotype, cf fitness);
117
118     MPI_Type_create_struct(nitems, blocklengths, offsets, types, &
119     mpi_genotype);
120     MPI_Type_commit(&mpi_genotype);
121 }
122
123 //*****
124
125 void evaluate ( int my_start, int row_count )
126 {
127     // return FITNESS_F == F1 ? f1() : f8();
128     switch(FITNESS_F) {
129         case F1:
130             return f1(my_start, row_count);
131             break;
132         case F8:
133             return f8(my_start, row_count);
134             break;

```



```

133         default :
134             break ;
135     }
136 }
137
138
139 void f1 (int my_start , int row_count)
140 {
141     int member ;
142     int i ;
143     double x[NVARS+1];
144
145     int ends = my_start + row_count ;
146
147     for ( member = my_start ; member < ends ; member++ )
148     {
149         for ( i = 0 ; i < NVARS ; i++ )
150         {
151             x[i+1] = population [member] . gene [ i ] ;
152         }
153         population [member] . fitness = ( x[1] * x[1] ) - ( x[1] * x[2] )
            + x[3] ;
154     }
155     return ;
156 }
157
158 /*
159  * Griewank Function
160  */
161 void f8 (int my_start , int row_count)
162 {
163     int member ;
164     int i ;
165     double x[NVARS+1];
166     double part1 , part2 ;
167
168
169     int ends = my_start + row_count ;
170

```

```

171  for ( member = my_start; member < ends; member++ )
172  {
173      for ( i = 0; i < NVAR; i++ )
174      {
175          x[i+1] = population[member].gene[i];
176      }
177
178      part1 = 0.0;
179      for(i = 1; i < NVAR; i++)
180      {
181          part1 += pow(x[i], 2) / 4000;
182      }
183
184      part2 = 1.0;
185      for(i = 1; i < NVAR; i++)
186      {
187          part2 *= cos( x[i] / sqrt(i) );
188      }
189
190      population[member].fitness = 1.0 + part1 - part2;
191  }
192  return;
193 }
194
195 void node_evaluate(int my_start, int row_count)
196 {
197     int member;
198     int i;
199     double x[NVAR+1];
200     int ends = my_start + row_count;
201
202     for ( member = my_start; member < ends; member++ )
203     {
204         for ( i = 0; i < NVAR; i++ )
205         {
206             x[i+1] = population[member].gene[i];
207         }
208         population[member].fitness = ( x[1] * x[1] ) - ( x[1] * x[2] )
            + x[3];

```

```

209     }
210     return ;
211 }
212
213 // *****
214
215 void node_mutate( int my_start , int row_count , int &seed )
216 {
217     const double a = 0.0;
218     const double b = 1.0;
219     int i;
220     int j;
221     double lbound;
222     double ubound;
223     double x;
224     int ends = my_start + row_count;
225
226     for ( i = my_start; i < ends; i++ )
227     {
228         for ( j = 0; j < NVARs; j++ )
229         {
230             x = r8_uniform_ab ( a , b , seed );
231             if ( x < PMUTATION )
232             {
233                 lbound = population[i].lower[j];
234                 ubound = population[i].upper[j];
235                 population[i].gene[j] = r8_uniform_ab ( lbound , ubound ,
236                 seed );
237             }
238         }
239     }
240     return ;
241 }
242
243 // *****
244
245 void node_selector ( int my_start , int row_count , int &seed )
246 {

```

```

247  const double a = 0.0;
248  const double b = 1.0;
249  int i;
250  int j;
251  int mem;
252  double p;
253  double sum;
254  int ends = my_start + row_count;
255  //
256  // Find the total fitness of the population.
257  //
258  sum = 0.0;
259  for ( mem = my_start; mem < ends; mem++ )
260  {
261      sum = sum + population[mem].fitness;
262  }
263  //
264  // Calculate the relative fitness of each member.
265  //
266  for ( mem = my_start; mem < ends; mem++ )
267  {
268      population[mem].rfitness = population[mem].fitness / sum;
269  }
270  //
271  // Calculate the cumulative fitness.
272  //
273  population[my_start].cfitness = population[my_start].rfitness;
274  for ( mem = my_start + 1; mem < ends; mem++ )
275  {
276      population[mem].cfitness = population[mem-1].cfitness +
277      population[mem].rfitness;
278  }
279  //
280  // Select survivors using cumulative fitness.
281  //
282  for ( i = my_start; i < ends; i++ )
283  {
284      p = r8_uniform_ab ( a, b, seed );
285      if ( p < population[my_start].cfitness )

```

```

286     {
287         newpopulation[i] = population[my_start];
288     }
289     else
290     {
291         for ( j = my_start; j < ends; j++ )
292         {
293             if ( population[j].cfitness <= p && p < population[j+1].
cfitness )
294             {
295                 newpopulation[i] = population[j+1];
296             }
297         }
298     }
299 }
300 //
301 // Overwrite the old population with the new one.
302 //
303 for ( i = my_start; i < ends; i++ )
304 {
305     population[i] = newpopulation[i];
306 }
307
308 return;
309 }
310 //*****
311
312 void node_Xover ( int one, int two, int &seed )
313 {
314     int i;
315     int point;
316     double t;
317 //
318 // Select the crossover point.
319 //
320 point = i4_uniform_ab ( 0, NVAR - 1, seed );
321 //
322 // Swap genes in positions 0 through POINT-1.
323 //

```

```

324     for ( i = 0; i < point; i++ )
325     {
326         t                                = population[one].gene[i];
327         population[one].gene[i] = population[two].gene[i];
328         population[two].gene[i] = t;
329     }
330
331     return;
332 }
333
334 void print_population()
335 {
336     for(int i = 0; i < POPSIZE + 1; i++)
337     {
338         cout << " | ";
339         for ( int j = 0; j < NVAR; j++ )
340         {
341             cout << "[" << population[i].gene[j] << " ] ";
342         }
343         cout << " | ";
344     }
345     cout << endl;
346 }

```