City College Plymouth

with the University of Plymouth

CITY3111

Individual Project

2019/20

BSc (Hons) Applied Computer Science

Jacob Deery

10711476

Using a Beowulf cluster to improve computational performance

Word count: 15 706

This report is submitted in partial fulfilment of the requirement for the degree of Applied Computer Science by Jacob J. Deery

For Bob Barbour & Karen Topper.
I promised I'd make it this far, did I not?

Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s).

Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged.

I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Jacob Deery

Abstract

Beowulf clusters offer great computational value for money as they can process a much greater amount of data than any single node can alone [1]. Free open-source (FOSS) implementations of various node clustering APIs now exist in abundance which make the creation of a high-performance computer (HPC) at home feasible. I intend on using one of these FOSS APIs to achieve a Beowulf cluster of my own using off-the-shelf hardware available to me. I undertake this project with a view to gain understanding of the current state in which non-commercial HPC is in, knowing that in future I might perform work using similar technology to further academic research fields such as bio-medicine, physics or engineering (amongst others).

Using my literature review, I concluded that an Open MPI cluster of five virtual nodes was able to outperform a single node by a magnitude close to the overall node increase. These results clearly show that using a larger pool of systems inter-connected by a TCP/IP network is able to offer a great speedup in computational power, even in spite of the relatively inexpensive nature of the underlying hardware, a trait that could be useful in research areas where budget is not of any particular depth.

Acknowledgements

I'd like to give thanks to the entire lecturing team at City College Plymouth for providing me with the skill-set required to have the opportunity to write this dissertation.

I'd also like to give a special thanks specifically to Tomasz Bergier for convincing me I am capable of doing a degree in the first place and giving me good guidance in the fields of statistics and high-performance computing. I'd also like to thank Andy and Anna for being superb mentors and great motivators in times of personal uncertainty.

Lastly, I'd like to give honourable mentions to Dr. Alma Rahat and David Walker at the University of Plymouth for their contributions to my work ethic and outlook on academia as a whole.

Table of Contents

Title Page

Dedication i

Declaration ii

Abstract iii

Acknowledgements iv

Table of Figures vii

Introduction 1

1 Literature Review 3

1,1 What, how and why parallel? . . . 3
1,2 A look into message-passing parallelism: MPI . . . 3
1,3 Networking with MPI: TCP, InfiniBand and Open MPI . . . 5
1,4 Understanding Parallel Programming using Open MPI: Example software . . . 5
1,5 Discussion on an alternative to Open MPI: PVM . . . 6
1,6 Conclusion on Literature Review . . . 7

2 Requirements and Analysis 8

2,1 My proposed system: Discussion on compute hardware choices . . . 8
2,2 My proposed system: Discussion on software choices . . . 9
2,3 My proposed system: In closing & additional components . . . 11

3 Design 12

3,1 Designing my cluster: Virtual Network configuration & topology . . . 12
3,2 Designing my cluster: The software design & configurations . . . 14
3,3 Building my custom software: A problem to test my multi-nodal network . . . 16

4 Implementation and Testing 20

4,1 Implementing a Beowulf cluster: Configuring and preparing . . . 20
4,2 Implementation and Testing: Creating my custom software . . . 27

5 Results and Discussion 32

5,1 Acquisition and analysis of results: Cluster performance measured by time taken . . 32
5,2 Brief discussion on project: Comments on project completion & future work 34

Conclusion 35

References 36

Appendices 39
Appendix α: Example MPI 'Hello World' program 39
Appendix β: My custom benchmarking software 40
Appendix γ: Acquired results from benchmarking software 47

Table of Figures

1,2,1 Diagram from book showing abstract message-passing architecture [2] 4
1,4,2 A compiled and running copy of Barrett et. al's 'hello world' program [3] 6

3,1,1 A topology diagram of my pre-existing network configuration including DS inner workings 12
3,1,2 A topology diagram of my revised network configuration including a second virtual switch 14
3,2,3 A general overview of my hypervisor architecture design 15
3,2,4 A visual representation of my goal message-passing architecture 16
3,3,5 A visual representation of the Ackley function [4] 17
3,3,6 A visual representation of the Eggholder function [5] 18
3,3,7 UML diagram of my intended software architecture 19

4,1,1 Screenshot of guest OS hardware configuration 21
4,1,2 Screenshot of completed 'cgdisk' process 22
4,1,3 Screenshot of base configuration menu 23
4,1,4 Screenshot of fully booted Manjaro system 23
4,1,5 Screenshot of VM cloning using Proxmox VE 26
4,2,6 Screenshot of successful Hello World execution 28
4,2,7 Screenshot of initial functioning test run 28
4,2,8 Screenshot of improved program test run 29
4,2,9 Screenshot of latest program test run 29
4,2,10 Screenshot of final program test run 30
4,2,11 Demonstration of problem area solution for effort creep 30

5,1,1 Line chart demonstrating predicted granularity occurring in my cluster 33

Key:

- First digit in figure: Chapter to locate
- Second digit in figure: Section to locate
- Final digit in figure: Figure number in chapter

Introduction

Research brief, focus & justifications

Beowulf computers were first devised as a concept by Sterling and Becker in 1995 [1] to refer to a specific cluster computer built entirely using 'beige box', off-the-shelf commodity hardware. Up until that stage in inter-nodal parallel computation, only specialist systems designed to be used as super-compute systems had existed and were certainly the preserve of high-budget projects.

Today, I believe there are sufficient tools available as free open-source software (FOSS) to facilitate Beowulf computing in a non-commercial environment. To that end, I hope to build such a cluster and either develop of acquire software which might benchmark the system with a view to prove the concept of easy-assembly high-performance computers at a minimal cost to the constructor and maintainer.

The focus of my research is aimed primarily at the construction and benchmarking of the Beowulf cluster. Specifically, I aim to demonstrate that a problem can be solved at a much greater speed when parallelism has been achieved compared to traditional single-threaded computation.

I believe that using commodity hardware to produce a cluster such as the one I aim to build has the potential to become a large part of many initiatives where the discovery of new knowledge is concerned. So-called 'cloud computing' already offers the ability to do this to a degree, however, in many cases having an in-house solution offers greater benefits and fine control to a research effort and can also be more cost-effective if the system is to be utilised long-term.

Overview of report content

For my first chapter, literature review, I hope to explore what parallelism offers and how it can be achieved, including discussions on what technologies exist (MPI, PVM, etc) to facilitate a FOSS Beowulf cluster on commodity hardware. I aim to offer examples of software that exists which demonstrates another, similar system to the one I will build that functions and conclude by exploring alternative methods to how I might achieve my goal from those discussed in my review.

Chapter two should outline the requirements of my system and what it should achieve. I hope to specify some of the tools I'll need to achieve the research goal of a functional high-performance computer and discuss any additional components I might need that do not directly fall into pre-determined categories of software or hardware. I hope to also decide at this stage if I will use a packaged benchmarking suite for my system or build my own custom benchmarking software.

Chapter three will expand on the previous chapter by forming a more concrete plan on how I will implement the Beowulf cluster. This will include discussions and visual diagrams on aspects of the system which will require meticulous construction, such as the network environment, choice of operating system and, if appropriate, custom software design. I will conclude this chapter by addressing any remaining design considerations I may need to take into account.

My fourth chapter will be centred around the implementation and testing of my system. Specifically, I will run through in chronological order the practical steps I undertake in order to complete the construction of my Beowulf cluster, discussing any hurdles I encounter as I progress. This chapter will be broken down into two sections, with the first being dedicated to the journaling of my hardware and operating system configuration and the second being dedicated to the benchmarking software creation (if applicable) and cluster testing.

The final chapter, a collection, discussion and review of my results, will hopefully collect results from a working prototype benchmarking system with a view to ascertain if my cluster is working and, if so, how well it is performing. This chapter will therefore be quite short, aiming simply to answer in a concise manner the nature of my project's success. Within, I might discuss issues with the implementation if any exist before suggesting improvements and opportunities of future continuation research.

Lastly, I will present a conclusion which re-iterates the contents of this body of work in a more concise fashion before terminating the dissertation.

Chapter 1

Literature Review

1,1 What, how and why parallel?

To begin my literature review, I shall ask the broadest of questions in this specific subject field: what is parallel computing, how is it done at a fundamental level and why should it be done? While searching for the answers to these questions, I have discovered a book titled 'Introduction to Parallel Computing' by Ananth Grama et. al. [6]

Within this book, I have found that parallel computing has been around for a while but its methods and purposes have changed over time. Grama et al. argues that parallel programming took a while to take off due to the uncertainty of hardware evolution going forward and the lack of consistent platforms to develop on. As of the book's publication this was changing however, with the standardisation of parallel platforms bringing about a much shorter development time for such projects.

The book also discusses the arguments for parallelism, as well as its real-world applications. When discussing the arguments for parallelism, the book authors suggest that pooling resources such as CPU time, memory and data links typically results in a high-performance computer, with the added benefits of high-availability, larger caching and higher aggregate bandwidth.

These benefits lend themselves well to several applications, ranging from engineering and scientific research to commercial applications such as web hosting and digital banking. One particular example that has piqued my interest is its applications in computer systems. Problems that were traditionally very computationally expensive (i.e. cryptography or distributed algorithm solving) can now be done on clusters in a much-reduced time.

Within Chapters 6 and 7 of this book, the authors expose two very different yet also very powerful methods of parallel programming. Chapter 6 covers programming using the message-passing model, a general term used to express any form of parallel program that sends its instructions to separate processes (either internally within a computer or externally to multiple nodes in a networked cluster), while Chapter 7 covers shared address space programming, which is typically achieved using POSIX threads.

Having read through both chapters, I have decided that the method best suited to my problem (i.e. building a Beowulf computer) will be message-passing. Message-passing is the older, more researched method of inter-computer parallelism with more available papers and better suited my needs as a whole for this particular project.

1,2 A look into message-passing parallelism: MPI

In order to better understand exactly why message-passing was used more often in inter-computer parallelism over shared address space parallelism, I have decided to look into methods of message-passing programming. While I have found many books on the subject matter, by far the most reputable and comprehensive title was 'Using MPI - Portable parallel programming with the Message-Passing Interface' by William Gropp et. al. [2]

Gropp et al. define message-passing as a collection of processes that only have local memory but maintain the ability to pass information between each other using some form of intercommunication protocol. Message-passing from once process to another inherently requires actions to be taken by both processes, which raises an interesting aside in my mind: does that mean each process within a message-passing cluster is a client/server in and of itself?

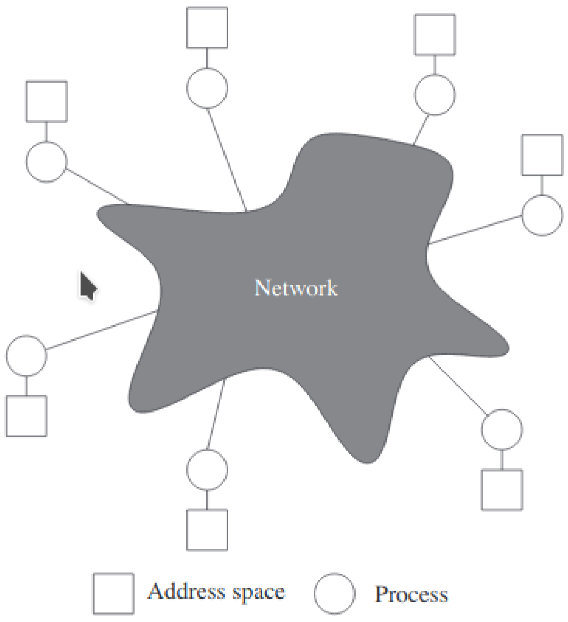


Figure 1,2,1: Diagram from book showing abstract message-passing architecture [2]

Also expressed in this chapter is the advantages found in using a message-passing model. For one, it is incredibly versatile, allowing for easy development, debugging and deployment on a wide range of hardware. Once aside that did occur to me, which I shall address later in this section, is: how does heterogeneous hardware affect the performance of a cluster?

The main point raised in this particular chapter, however, is the performance gains to be had from utilising message-passing, which is exactly what I am hunting for in my research. According to the authors, message-passing permits the host hardware full management over its caching, thus affording the CPU full and unhindered functionality which in turn drives performance gains over even the most generously designed of single-processor machines with regards to available cache memory.

Section 1,5 and Chapter 2 of this book specifically offers insight into one particular implementation of a message-passing standard: Message-Passing Interface (MPI) and its governing body, the MPI forum. The book authors tell of the development of the MPI standard as a necessity of the time in an age where message-passing softwares were plentiful, typically too bespoke for general purpose high-performance computing (HPC) and commercial in nature. The MPI Forum concluded in 1992 they would develop a standard which was portable, open and aimed to be completed in a year. The subsequent product was MPI-1 (1994) [7], released after little over a year of frantic development.

The MPI standard has since been revised numerous times, with its latest publication coming in 2015 [8]. MPI can therefore be relied on as a fresh and well-maintained standard based around a paradigm that is suitable for my research topic. Alas, no manner of searching I conduct can find an equivalent to compare MPI against that is as comprehensive, portable and recent as MPI appears to be. The next piece of secondary research that therefore comes to mind centres around discovering a suitable network implementation to run MPI atop.

Referring lastly back to a question I posed earlier in this section as an aside – "how does heterogeneous hardware affect the performance of a cluster?", Section 7,4 of this particular book expresses how MPI manages the translation of data representation between systems with incompatible data formats. While this is encouraging news for heterogeneous clusters, it does not satisfy my curiosity and additional research will be needed to cover this topic point.

1,3 Networking with MPI: TCP, InfiniBand and Open MPI

Discovering MPI will facilitate inter-nodal computation was only the first step in assessing how I can implement my project. The next step is to try and ascertain what communication standard I can use to enable optimal link speeds between all of my nodes.

I began by searching in general terms for MPI network implementations before resolving to nuance my searches down to specific network protocols I have knowledge of, the first and most popular being the Transmission Control Protocol (TCP).

The conclusion of this research turned up two papers of interest, the first of which being 'Open MPI: A Flexible High Performance MPI' by Richard L. Graham (et. al. 2005) [9]. Within the abstract of this paper, Graham et. al. observe how, while the MPI Form defines a standard, MPI implementations are not cross compatible. This historically caused a swathe of specialised MPI implementations that a researcher would need to choose based on their usecase and stick to for the duration of a project.

Open MPI aims to solve some of these issues by offering the ability to include third-party software add-ons in the MPI run-time, as well as by offering novel features not available in other MPI implementations. While my usage of MPI will likely not require any of these novel features or the inclusion of third-party software during run-time, it is nonetheless an implementation worth looking into for my project.

The main interest of this paper for me can be seen in Table 1 and Table 2, however. Here, I can see the authors have conducted a test with a view to ascertain which protocol is fastest by testing the system latency in a point-to-point zero-byte ping. Unbeknown to me, TCP actually performed poorly in comparison to both the InfiniBand implementations and the Myrinet stack. While some of this may be attributed to the hardware discrepancies seen in Table 1, these figures definitely merit my further research into InfiniBand technology with Open MPI.

I performed some additional research into Open MPI [10] and discovered the InfiniBand is also very well-suited to HPC not only because of its lower latency compared to TCP but also thanks to its impressive scalability metrics. This is, however, where I had to leave my research regarding InfiniBand as the hardware costs are astronomical and greatly exceed my budget for this project.

I therefore resolved to press on with Open MPI using TCP. My next natural step was to find some programs that had been written for an Open MPI environment.

1,4 Understanding Parallel Programming using Open MPI: Example software

To garner an understanding of how Open MPI works in a practical setting, I want to find some example source code. Open MPI has an up-to-date API reference [11] and features workshop sessions, explaining through digital lectures how to program a basic 'hello world' system. The discovered example program can be found in Appendix α.

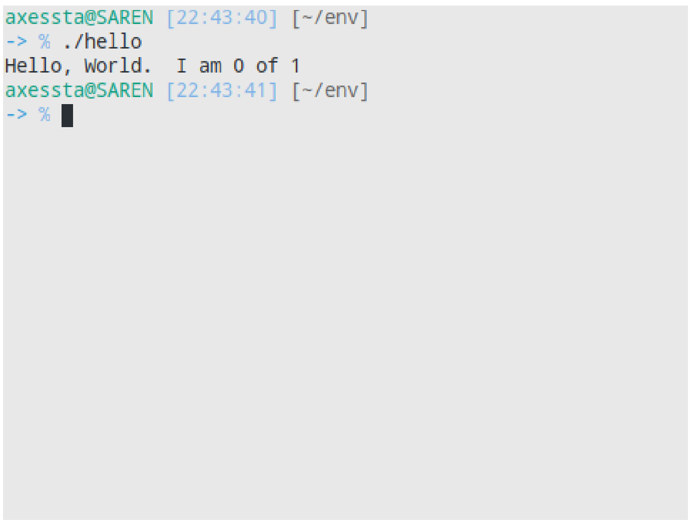
Anyone familiar with C/C++ should understand the syntax involved in the aforementioned code snippet, the only difference is the methods in use (specifically, the four MPI methods). Referencing the Open MPI documentation, each of these methods should roughly achieve the following:

- ***MPI_Init*** – This method initialises the MPI environment either using the C/C++ arcg/argv[] pointers or with NULL values.
- ***MPI_Comm_rank*** – This method will acquire the current processes rank in the MPI 'world' (i.e. the cluster). This can be useful for a number of reasons, including message-passing identification or debug printing.
- ***MPI_Comm_size*** – This method will acquire

the current MPI ‘world’ size. This is useful in ascertaining if all nodes are connecting well, as well as for dynamic task division.

- ***MPI_Finalize*** – This method will clean up the MPI subroutine when called, and is typically used just before the parent function is returned.

If my understanding the API is precise, the example code will, when compiled, display a list of all ranked processes within the MPI ‘world’ and output a ‘hello world’ from each. To ensure my understanding is correct, I’ll now attempt to display the ‘hello world’ program in action by compiling it using the custom ‘mpicc’ compiler also mentioned in the documentation (based upon the GNU C/C++ compiler).



```
axessta@SAREN [22:43:40] [~/env]
-> % ./hello
Hello, World.  I am 0 of 1
axessta@SAREN [22:43:41] [~/env]
-> %
```

Figure 1,4,2: A compiled and running copy of Barrett et. al’s ‘hello world’ program [3]

The Open MPI example code functions as intended and displays a ‘hello world’ prompt, but the world only encompasses a single process at this time. In order to introduce more processes into the world, I need to introduce a runtime component which will facilitate the spawning of additional processes as well as their communication across my cluster – even between nodes. This runtime is included in the Open MPI distribution, which I will further research in my system analysis.

1,5 Discussion on an alternative to Open MPI: PVM

In my literature review so far, I have looked at different methods available to me for creating a parallel cluster and have narrowed down on one particular framework. Before I conclude my literature review, I am going to discuss another framework that could be used and why I won’t be using it.

The first, and oldest of the two frameworks is parallel virtual machine (PVM). According to its manual [12], PVM aims to provide much of the same message-passing functionality as MPI does but instead of offering each node up as individuals to solve a whole, it attempts to simulate one large, singular virtual machine upon which many processor cores co-exist.

These design difference are born out of the differences in their development and purpose, according to Dr. Elts [13]. Take the development argument, for example. As we’ve previously discussed, MPI had a rather rapid development with a large number of stakeholders whom all had to be satiated. PVM was developed and designed earlier than MPI and by a single research group in Oak Ridge, TN with a view to create a virtual machine that could span multiple machines. MPI, in contrast, explicitly specified against parallel virtualisation within its core specification.

These differences in development and goal lead to two very different frameworks and implementations. MPI allows for flexible allocation of processes, with many topologies being feasible (likely a requirement of so many stakeholders) while PVM only allows for a master/slave architecture with poor fault tolerance if the master node falls over. Additionally, as PVM attempts to pool all memory under one virtual machine, the aforementioned cache control speedups MPI typically experiences are lost in PVM.

To better explain the previous point, imagine a suitable Republic such as the French Republic. There are two main institutions at play, somewhat analogous to PVM and MPI in function. In the French Republic, the *président* or *présidente* appoints his or her *gouvernement* who will then liaise on a regular basis, but the président retains authority in the cabinet (i.e. a master/slave architecture). This relationship can be seen as the relationship nodes are under with PVM, one of rigidity and structure.

The second pillar of any good republic's structure is the *parliament*, specifically the *Assemblée Nationale* in our example. The Assemblée Nationale is brought together collectively to perform a function in unison, with little to no guidance aside from a schedule (i.e. a mesh topology). This is more analogous to MPI, whose processes are loosely directed by a runtime but are otherwise free to communicate with any other member of the cluster as it sees fit.

For my purposes and function, having the flexibility of being able to choose my topology is critical. I don't yet know what programming challenges I may encounter, and having a more up-to-date and powerful tool-set seems like the best choice for my project going forward.

1,6 Conclusion on Literature Review

Within my literature review, I have investigated different methods one can utilise in order to produce a working parallel computer using commodity, open-source software and have investigated the principles underlying parallelism and why one might choose to use such methods.

I have taken an in-depth look at the MPI standard and expressed why this standard is suitable for my project goal, and have picked out Open MPI and a popular example implementation of the MPI standard to further express its function and usefulness to my project, offering example code for reference.

I have then taken a step back from Open MPI and looked at PVM as an alternative technology with which similar projects have been completed using before, discussing the fundamental differences between them and why PVM would not be as suitable for my task as MPI.

In conclusion to my literature review, I offer an impact case study from the Research Excellence Framework of 2014 [14]. Dr Trew et al. posit that the impact of MPI is far-reaching, with MPI being regarded the *de-facto* standard in parallel computer programming.

This has been attributed to its wide availability, with some distribution being available for almost all architectures ranging from desktop computers to the TOP500 supercomputers (2014), and the portability associated with MPI's availability.

Chapter 2

Requirements and Analysis

2,1 My proposed system: Discussion on compute hardware choices

While developing my system, the first thing I need to consider is the hardware it will be built upon. Specifically, I need to consider the computer system or systems I will use, the network hardware that will be required, any peripherals for those two interconnects and the associated costs involved with acquiring this hardware.

Before I delve into this discussion, I must first lay out some assumptions on any hardware setup I go forwards with in order to ensure there is a good understanding of what is required of my system in order for any derived results to be viewed as fair and scientific.

Firstly, any compute nodes used within my cluster must match any other compute node – including the master. For example, if a system within the cluster has an i7 CPU and another has an i3 CPU, this cannot be considered a scientific test as the cluster will likely show uneven speedups in the final results.

Lastly, any interconnect in use (i.e. 1 000 BASE-T Ethernet) must be uniform in nature. For example, if one system has a 1 000 BASE-T Ethernet adapter and another has a 100 BASE-T Ethernet adapter, this cannot be viewed as a fair test as nodes within the cluster will be communicating at different baud rates.

Raspberry Pi as cluster

Starting now on this discussion, the first consideration is to use a cluster of five Raspberry Pi 4s connected to a Gigabit switch via Cat5e Ethernet. This is inspired in large part by the research completed at the University of Southampton, where Pi Model Bs were assembled in a similar manner [15]. By utilising lots of small compute nodes, I would hope to achieve cluster built atop MPI capable of very high flop/watt performance at a reasonable price point.

The decision to build my cluster using five nodes was born out of the thought process that the more nodes I can have, the more data I can collect. Five nodes of quad-core machines would result in twenty NUMA nodes in total – sufficient processor counts to offer meaningful insight into the performance of my system.

Unfortunately, during my feasibility study, I have found that using Raspberry Pi systems is not possible in my research scope due to budgetary constraints (that is, I have neither an institutional nor a personal budget) and the required hardware is not available for lease at my institution.

A compromise: Utilising a DS

To complete my cluster, I have ended up needing to utilise hardware existing to me. To this end, I have decided to use my Dell PowerEdge R710 featuring 2 x Intel Xeon CPU X5660 @ 2,80GHz with 96 GB of DDR3 ECC memory. This system also has four 1 000 BASE-T Ethernet ports making it suitable for external management. I will run five guests within my hypervisor in order to simulate a good range of loads on my dedicated server (DS).

Using a production server comes with its own set of challenges. How do you balance the resource requirements of other virtual guests in order to maintain a fair test? How can you ensure all systems communicate fairly (i.e. one system isn't getting priority CPU or network time)? Are there any additional considerations required when designing such a system? Try try and answer those questions, I'd like to express how I intend on mitigating and adapting the load on my server machine while operating the program and cluster.

The first and obvious step I would take is to run my programs at off-peak hours (typically night time as the majority of my users are local to Europe). By doing this, I ensure that step two has minimal business impact on myself.

This leads onto the second alluded to mitigation step, which is to shut down all non-essential guests running on the server. By doing this, only CPU and network time essential to maintaining a connection to both the internet and my local area network are being consumed, leaving the vast majority of time available for my project program.

Other more adaptive tasks can be conducted on the fly as the network environment changes, which might include disabling or turning off some network-connected devices that appear to consume disproportionate network resources. Unfortunately these tactics cannot be planned for but simply reacted to in a timely manner without a more dedicated laboratory setup.

2,2 My proposed system: Discussion on software choices

As explored in my literature review, Open MPI appears to provide the best available bedrock for producing a cluster computer using commodity hardware as it offers a balance of diverse and stable APIs and good availability for a plethora of system and network architectures. Therefore, the choice with regards to software to use within my system is more aimed at the Operating System (OS) end of the software spectrum.

A preamble on the existing DS implementation

My DS runs a specialised version of Debian called Proxmox VE which is an open-source bare-metal hypervisor with a non-commercial licensing tier [16]. Proxmox VE's underlying technology is based on the existing Linux KVM with Qemu hardware virtualiser which is both computationally fast and resource efficient, two prerequisites for this project.

One of the major advantages (and considerations) I can find in using this setup is I/O. Whereas in a standard cluster computer, your bottle-neck is almost always going to come externally (i.e. from the network or associated adaptors) the DS offers a much higher theoretical peak bandwidth between guest OSes. This fact will work in my favour, however there is a chance it could also skew results away from the expected norm, as the bottleneck found in production Beowulf machines will have moved from the network to the component bus.

A choice in Guest Operating Systems

As my systems will be running atop a hypervisor, I need to make a decision on which flavour of Linux to use for my guest machines. There is a plethora of choice in the headless Linux market, but the main choices ultimately boil down to Debian/Ubuntu, CentOS, RHEL/SLES/openSUSE or Arch/Manjaro [17]. Of the four candidate OSes, three categories can be identified which might further assist in decision making given the requirements of the project.

First, there is the stable distributions which include both Debian-based OSes and CentOS. These OSes are designed for use on servers and other mission-critical hardware where stability is deemed more important than having up-to-date libraries and drivers. While there would in theory be no issue in using these OSes types for my guest machines, a more up-to-date set of libraries might help me gain additional features from any cluster I attempt to build.

The second of the three OS classifications is enterprise-grade suites which come in the form of RHEL/SLES/openSUSE in this case. Much like Microsoft Windows, enterprise-grade Linux flavours are aimed at businesses and educational institutes and are thus entirely unsuitable for the tasks I will be asking it to perform. For one part, it likely does not have the OpenMPI library in its package repositories which would make doing what I need to do with regards to building a cluster unnecessarily more challenging. For another part, it likely also has packages, libraries and drivers even more out of date than the stable classification of OSes mentioned in the previous paragraph.

Lastly, I come to the third classification which is rolling release distributions. These distributions are typically more feature-bare and unstable than the previous two categories, instead the operator is expected to build a bespoke system using a rich and up-to-date package repository. Arch Linux is the most popular rolling Linux distribution with Manjaro being a forked Arch distribution with additional tools to aid in user-friendliness.

I am choosing to use Manjaro for my project due to its merits as a bleeding-edge distro, where Manjaro Architect shall allow me to design an image with very low memory and CPU overhead and the bare-bones package subset I will need to run and test my cluster. [18].

Measuring the system: A benchmarking software

In order to provide metrics on the performance of my system as nodes are scaled and introduced, I will need to benchmark them. While there are benchmarking suites out there such as LINPACK [19] which will do a commendable job in assessing my cluster's performance, I wish to take the challenge upon myself to program a suitable binary for this job and there are two distinct reasons for this.

Firstly, LINPACK is a complex suite of benchmarking softwares which, while tried and tested, are also barely understood by myself and it would take a comprehensive reading far outside the scope of this project for me to come to understand. My logic stands that if I were to write my own benchmarking software, I could both learn how parallel programming is done and also offer deeper insight into what the results mean than if I were to use a prepackaged binary to do the same job.

Secondly, LINPACK takes an extraordinary amount of time and memory to do its job. While this wouldn't necessarily be an issue on a singular test run or on a system where the computational power is exceptionally large, my system fits neither of those descriptors. With my own benchmarking software, I can design in a mechanism to allow for well-scaled performance curves that still offer a picture of the system's overall power availability.

To fulfil this aim, I will need as a basic requirement to maximise CPU time used on all given nodes. A computationally expensive problem must be found and during testing CPU readings on each node must be taken during each run to confirm the process is fulfilling these requirements.

A final word: Other software packages required

I've discussed in detail the implementation of MPI I will be using in my literature review, as well as the existing software running on my DS that will allow guest OSes to be spawned and communicate with each other. I have also performed a condensed look on the types of OS I could and will use for my project. I will now close this portion of the chapter by discussing the other software packages that will be needed to enable my Beowulf cluster.

Assuming my guest OSes have been installed and are online, I will need a working installation of the GNU Compiler Collection (GCC) [20]. GCC is a prerequisite of OpenMPI, which uses a modified version of GCC to compile its own binaries for use with the MPI runtime element.

I will also need a working installation set to run at boot of the OpenSSH Daemon [21]. The SSH protocol appears to be the recommended method for tunnelling between nodes within an MPI cluster due to its lightweight and always-on nature, as well as for its secure cipher suite in Arch/Manjaro OSes.

These packages fundamentally meet the requirements of the project. Additional, optional components may be added on the fly but I cannot currently predict any further relevant packages being required.

2,3 My proposed system: In closing & additional components

With functional hardware and software, there is little else to write about regarding requirements. Indeed, there may be insufficient words for me to attribute to this section to justify making it a section at all, however I feel it is worth acknowledging as a separate discussion from the topics of hardware and software.

Network Hardware: The Physical Switch, Virtual Switch & NICs

Without discussing topologies (as this is a major discussion I plan on having in the next chapter), the network hardware I plan to use consists of three sub-components: the physical switch, the virtual switch and the network interface cards (NICs).

The physical switch is what connects the physical server to my home network. It is a 1 000 BASE-T Ethernet switch with 24-ports allowing for good excitability within this project. As no computational exchanges will be happening over the physical network, Gigabit speeds are sufficient in this project, especially given how my edge speeds are only 300 Megabits per second (or 35 Megabytes per second).

The virtual switch is the software switch which facilitates communications within the hypervisor, both between connected nodes and the outside world. This software switch is already configured on my DS for optimal performance and for network bonding, allowing for four times external throughput when required.

Lastly, the NICs, both physical and virtual. As mentioned, the physical NICs are four bonded 1 000 BASE-T Ethernet devices allowing for 4 Gigabits per second (or 0,5 Gigabytes per second). Additionally, the virtual switch and hypervisor virtual NICs both allow for bandwidths of 100 000 BASE-T or upwards of 100 Gigabits per second (or 12,5 Gigabytes per second) which should remove any bottleneck that could exist between the machines at a network level, instead placing the strain on the system's buses. This will change the dynamic of how the cluster will interact considerably, something I will need to heavily reflect on in my review of the project.

Chapter 3

Design

3,1 Designing my cluster: Virtual Network configuration & topology

When designing my system, many of the considerations appear to naturally come from the Requirements & Analysis stage. For example, I will need to consider how virtual components interact with one-another, as well as with the physical hardware it is housed on. I will also need to put strong consideration into how the system performs and what bottlenecks could exist which might cause the system to behave sub-optimally.

To that end, I hope to discuss my desired implementation of many of the same aforementioned components in this chapter as were discussed in the prior chapter. In this section, I will open by setting forth the architecture of my existing physical network as well as proposing a suitable virtual network to be constructed within the hypervisor.

The existing network topology: A Map

In order to design a new network topology for my Beowulf cluster to integrate with, I first need to explain to you the reader the exact nature of my current network.

In order to achieve this, I will lay out in a topographic map how all my devices currently communicate both with each other and with the internet with a brief description of exactly what devices are performing which functions. Then, to give the map some context, I will express examples of how certain transactions may occur across the network.

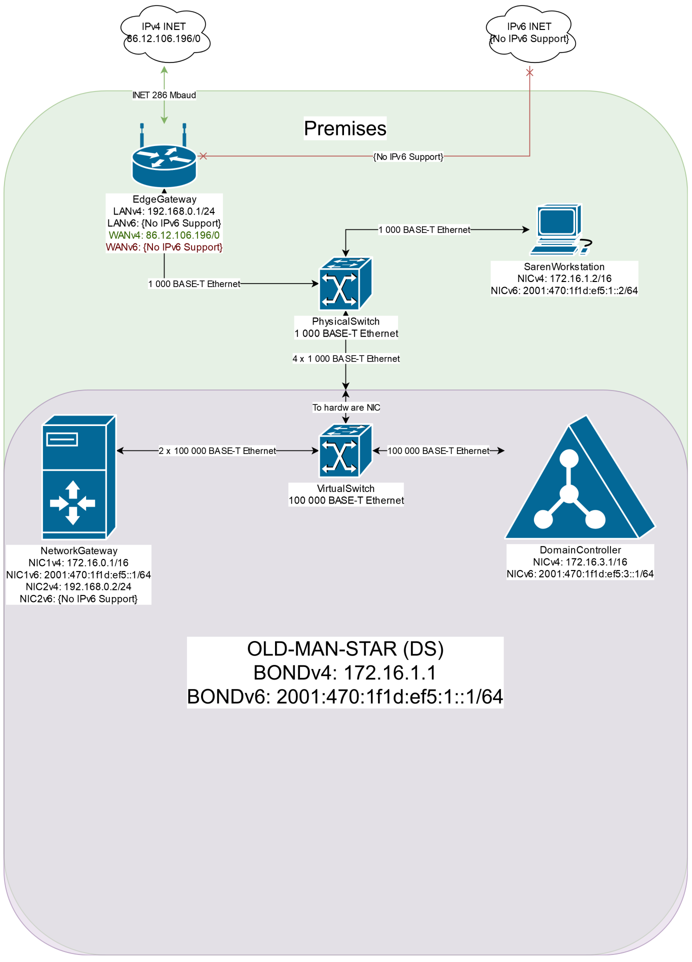


Figure 3,1,1: A topology diagram of my pre-existing network configuration including DS inner workings

Working from the top down, I first believe the Internet needs no in-depth explanation. There are currently two versions of the Internet running in parallel: the older Internet Protocol (IP) version 4 (specified in RFC 791 [22]) which we all use on a daily basis to communicate, fetch data, share the general stuff we do, whoever we are and whatever we value. The second version of the Internet is the IPv6 suite (specified in RFC 8200 [23]) which is much newer, smaller and is not cross-compatible with IPv4. This suite was published to help ease a predicted squeeze on the address space found in IPv4.

My ISP does not currently feature IPv6 support, however my network does. This will be an important consideration in the design process I am about to undertake, as I will need to take extra care in ensuring that my guest OSes are capable of downloading packages and updates via the Internet and having a pre-configured IPv6 network may cause newer OSes to think there is an IPv6 gateway available when there isn't.

Moving downwards toward the on-premise equipment, I have three separate network devices outside of the DS: the edge gateway device, the physical switch and my personal computer Saren. Firstly, the edge gateway manages communications between all internal devices and the internet with a LAN subnet of 192.168.0.0/24. The physical switch operates on layer 2 of the OSI model [24] and facilitates communications between any physical devices on my network not connecting via the edge gateway's built-in Wi-Fi access point.

Next, my PC, Saren, will be used to oversee operations and issue commands to both the physical hardware on my network and the virtualised components found on my DS. I will achieve this by using a combination of the Proxmox VE web interface to manage the virtual hardware, Virt-Viewer to directly interact with the console of each individual guest OS and SSH to access teletype sessions on each system once they are fully configured.

Finally reaching the DS itself, this is where the nuts and bolts of the operation come into life. Within the DS there is another, much faster virtual switch which is used to virtually connect all of my guest OSes together. This virtual switch also has direct access to the physical NICs on the server to facilitate external communications.

In addition to the virtual switch, there are currently a plethora of guest OSes running various components of my home network. I have decided to include the two most vital ones however, and the ones that must run in order to provision a skeleton network (i.e. one with internet).

The Network Gateway operates as a communicator between the edge gateway and the rest of the network, providing a larger subnet of 172.16.0.0/16. It also acts as the only DHCP and DNS server advertised within my home network and provides DNS forwarding and caching to facilitate fast internet access for devices trying to access the internet.

The Domain Controller manages network-wide authentication tasks as well as timekeeping and policy-setting. The Domain Controller won't necessarily be of particular use to us however it does offer DNS propagation for internal network devices so it cannot be shut down during even the most minimal network environments.

The existing network topology: Some examples

To demonstrate how devices currently connect to my network communicate, I will express some fictitious examples of communication routes. This is relevant simply because it helps build a picture in you, the reader's mind of the basic structure involved in this system.

First, lets imagine Saren wants to acquire some arbitrary data from a session-less (i.e. UDP) source on the internet. Without diving into the differences between the TCP and UDP protocols, Saren would send a request which would first travel to the network gateway via the physical and virtual switches. The network gateway would then recognise the destination of the request as external and send the packet to the edge gateway via the virtual and physical switches once more. The edge gateway will then send the request off onto the internet, and from there direct and responses via the same path the request was sent (i.e. to Saren via the network gateway).

I appreciate how complex such an example may be, so allow me to explain another, perhaps simpler to grasp example to supplement my main paragraph above. Imagine again some session-less request happening, but this time it is being done by the hypervisor itself, Old-Man-Star. As the hypervisor is technically also hosting the network router, any data exchanges all happen internally across the system busses. The resultant computation is a request packet destined for the edge router and ultimately the internet, which will go via the physical router. When viewed in this sense, one of the potential advantages of using a hypervisor to prototype OpenMPI software becomes apparent: there is a huge drop in the delay experienced when exchanging information that would otherwise need to travel between two physical nodes.

The revised network topology: The five guest OSes & other alterations

Armed with a topology map and a detailed understanding of the network structure as a whole, I can now explain to you the additions I will be making to the structure of the network and why.

As displayed in the revised topology map, I have decided to add another virtual switch into my DS's hypervisor. The logic behind this thought process is that by separating the MPI nodes onto their own switch with a dedicated IPv4 and IPv6 subnet, I can avoid regular network traffic. This solves issues that were discussed previously regarding bandwidth contention and also avoids or even negates entirely the likelihood of packet clashing.

Aside from the addition of a second virtual switch, the only other addition is the guest machines themselves. The guests are connected to both the existing virtual switch and the new, dedicated virtual switch over two 100 000 BASE-T Ethernet NICs per node. This is essentially the only major changes made to the network configuration, my philosophy being that by keeping the network fundamentally unchanged where it isn't necessary will help retain the pre-existing stability as best as possible.

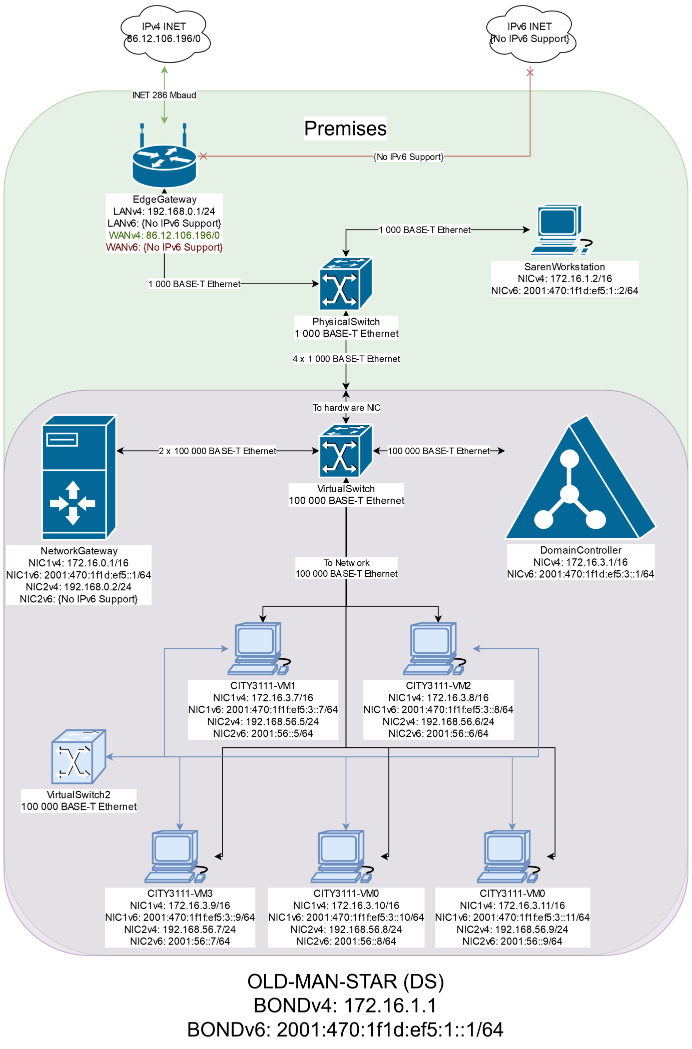


Figure 3,1,2: A topology diagram of my revised network configuration including a second virtual switch

3,2 Designing my cluster: The software design & configurations

Hardware design complete, my next stage is to design the software. Software used in my project can be divided up into two subcategories: packaged software and custom software.

In this document, the term packaged software expands to cover all software not written by myself, which includes the operating systems in use throughout the cluster in addition so any software running on them. Custom software will refer to any software I write to run atop OpenMPI.

Designing my test environment: hypervisor re-configurations & guest considerations

In order to have a place to install, compile, run or test software effectively I need the hypervisor to be configured correctly and optimally. Currently, large portions of my DS are configured for stability, which in some cases means utilising older firmware packages or opting for software solutions to handle certain virtualisation tasks.

While this is acceptable and even recommended in production environments, in development environments, the less hindrance to performance the better the end results in most cases. To best optimise my virtual Beowulf cluster, I will apply more experimental or hardware-centric virtualisation methods where possible to deliver the best possible results during testing.

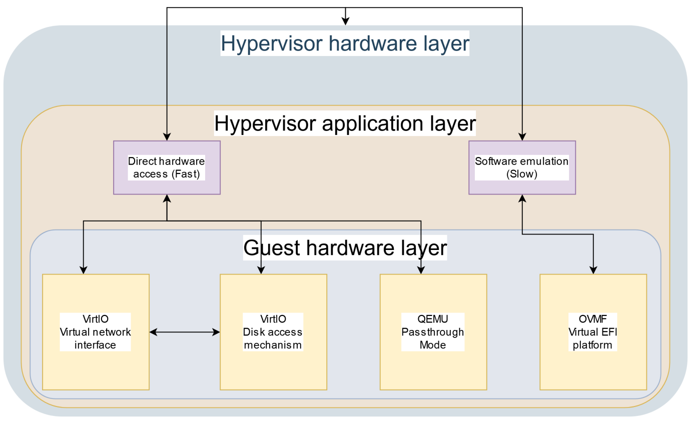

Hypervisor hardware layer
Hypervisor application layer
Direct hardware access (Fast)
Software emulation (Slow)
Guest hardware layer
VirtIO Virtual network interface
VirtIO Disk access mechanism
QEMU Passthrough Mode
OVMF Virtual EFI platform

Figure 3,2,3: A general overview of my hypervisor architecture design

My hypervisor uses KVM/Qemu for virtualisation which means it is incredibly diverse and can even emulate other CPUs and CPU architectures. While this can and, indeed, has been a feature of great usefulness to me outside of this project, it is also a potential barrier to optimal performance within this task.

The solution offered by Qemu, as seen in the figure above, is passthrough mode. By enabling this flag, Qemu will avoid emulating different CPU architectures or instruction set extensions and will instead offer the full suite of hardware CPU extensions to the virtual machine with its original model number and architecture in tact. This is the fastest available CPU model within the Qemu virtualisation software as there is virtually no software interference from Qemu/KVM during code execution. I will pass through a single core per node from each socket, offering load-balancing across both CPUs to minimise overall time requirements on the processors.

In addition to making changes to the way CPUs are presented to the guest machines, I will also be ensuring the mechanisms used for other virtual hardware components have minimal software overhead on the hypervisor. For example, Qemu/KVM offers a suite of firmware packages collectively known as the VirtIO drivers [25] which permit closer access to physical hardware from within the virtual guest.

I will apply the VirtIO device types to my network adapter and virtual disk I/O bus. This should offer me maximum bandwidth between the virtual machines as well as between the virtual machines themselves and the DS RAID array. VirtIO is the only virtual network adapter type within Qemu/KVM that allows for 100 000 BASE-T Ethernet speeds.

As the system has 96GB of RAM available, I will present the full 4GB of RAM found on the reference hardware, the Raspberry Pi. While software requirements should not exceed that figure per node, it is a trivial task to allocate more memory to the virtual guests if it is so required.

Lastly, I will also opt to use an EFI bootloader over a virtualised BIOS. While this is less relevant to performance as a whole, I still felt it was worth mentioning at the end of this subsection due to the impact it may play when hardware requests are made by a guest machine.

EFIs hands more device access over to the operating system in the early stages of booting which means that they start up faster and have the ability to call directly on hardware with greater ease. While this is unlikely to make a resounding difference it at all to my test program, it could help and enabling it will do no harm otherwise.

Designing my test environment: A breakdown of the guest software architecture

With the hypervisor configuration and architecture design complete, the next stage is to design my guests. Each guest will be as close as possible to a carbon copy of one-another, possible through last minute imaging of each machine from a preconfigured master.

The fundamental logic behind this is, if the differences between each node in the cluster are minimised, the test can be deemed as more scientific. The same background processes should run at start up, so the systems can be gauged as more alike at startup and during operation and the same firmware should be installed to the same version numbers.

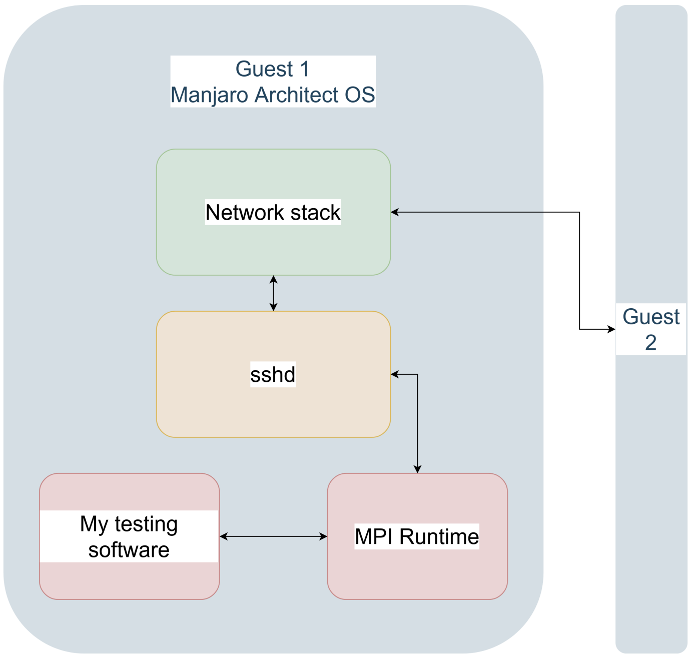


Figure 3,2,4: A visual representation of my goal message-passing architecture

The main packages I will require have already been discussed in detail, but as a recap, they are the VirtIO network device firmware, OpenSSH and the MPI runtime. Each software package will need to interact with the others, as well as with the network device firmware. Ultimately, the network device firmware will be responsible for external communications with OpenSSH being the application layer for network transactions. The MPI runtime will thus act as a translator between OpenSSH and my own custom software. With regards to my own custom software and how that communicates with the wider guest operating system, I will require the mpicc compiler (not noted in the diagram above) to compile the various hooks required by the runtime into my binary. Typically, the mpicc compiler is included in the same package as the runtime (in Arch's case, openmpi).

3,3 Building my custom software: A problem to test my multi-nodal network

In order to test my system architecture thoroughly, I am in need of a problem which can be easily divided between processes with sufficient computational expense to raise each CPU thread to its maximum throughput. The problem should also be of peer-reviewed nature in that it should use a principle already widely explored and proofed already by an external, verifiable source.

In my research for a suitable problem, I discovered a few problems which might be suitable for stress testing my Beowulf cluster, but the two I will discuss here will be hashing functions and fitness optimisation functions. I will use this section to finalise this paragraph by discussing these two methods, why I've chosen one over the other and how I plan on implementing my chosen method.

Two methods for testing my cluster: hash functions primer & discussion

Cryptographic hash functions are defined as the mathematical algorithms which can 'digitally verify' a data block. Specifically, it aims to create a string typically fixed in length which can uniquely represent a quantity of data such as a string, file, binary or file system.

The study of what we today understand as hash functions began in the 1970s with the Diffie-Hellman (DH) method [26] which underpins modern public/private cryptography methods used to exchange data over public transmission lines such as the Internet. In short years, Rabin [27], Merkle [28] and Yuval [29] (amongst others) quickly built upon the DH model, offering real-world applications to the public key cryptography concept that today drives secure communications everywhere.

Between the 1970s and 1990s cryptography of this kind did not see particularly wide mainstream adoption outside of specialist computing and it wasn't until the late 1990s and the major adoption of the Internet as a common data exchanging environment that caused an upsurge of DH-modelled algorithms to emerge and gain widespread usage for general cybersecurity purposes. [30]

Modern hash algorithms such as the SHA-2/SHA-3 suites and EdDSA still base themselves on the original DH paradigm, where the fundamental changes are less structural and more mathematical in nature. Due to the increasing mathematical complexity required to secure systems that get faster over time, calculating hash functions using a sufficiently modern algorithm against a sufficiently large block of data should result in high CPU load for a long enough time-frame to get a strong data-set with which I can scientifically analyse.

Hashing functions can therefore be seen as one potential method of benchmarking a cluster, especially when one considers how susceptible many hashing algorithms are to parallelism. However, hashing functions are notoriously difficult to write so I would more likely than not need to find and utilise source code to compile my own hashing binary which is compatible with the MPI runtime. This would in turn require me modifying existing source code to contain MPI hooks and functions.

For the context of this degree-level dissertation, I think that a hashing function would likely incur too much complexity to be completed within a reasonable time-frame. I am therefore going to review my options and look down a different avenue of research.

Two methods for testing my cluster: fitness optimisation functions primer & discussion

Fitness optimisation functions are ubiquitous in genetic programming as a way of testing how good a specific algorithm is at locating an optimal solution. Numerous fitness functions exist, however the most popular include the Rastrigin function, Ackley function, Rosenbrock function, Himmelblau's function and Eggholder function.

The main goal of fitness optimisation functions is to benchmark a genetic algorithm's performance against either other similar methods or improvements on an existing method. Every individual fitness function therefore has a different design and purpose in mind and offers different search-space constraints and granularity. To understand how fitness optimisation functions work, I will look briefly into two different methods, namely the Ackley and Eggholder functions.

The Ackley function [31] is a two-dimensional optimisation first conceived by David Ackley in 1987. It has a search space ranging between -5 and 5 and a global minima at { 0,0 }. When visibly rendered on a three-dimensional plot, it assumes a funnel-like shape with peaks and troughs lining the fabric of the search space leading towards the centre of the grid.

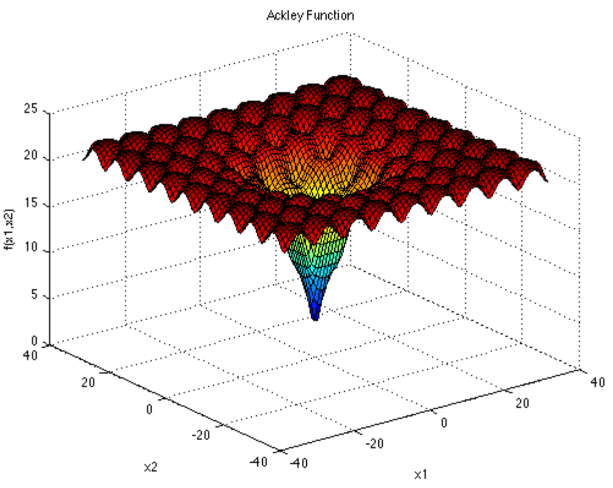


Figure 3,3,5: A visual representation of the Ackley function [4]

It is these peaks and troughs optimisation algorithms come unstuck with and it is precisely this characteristic that mathematicians hunt for in their efforts to form fitness functions. In the case of my software, in order to maximise results I hope to do the opposite of optimisation and actually add computational expense to the software rather than reduce it.

I will now take a brief look into a different function, the Eggholder.

The Eggholder function aims to fulfil fundamentally the same niche as Ackley's function did by providing a search space that contains a high volume of local minima in a comparatively small search space. In the case of the Eggholder, its search space is much larger than Ackley's with bounds between -512 and 512 with a general optima of -959,6407 at {512, 404.2319}.

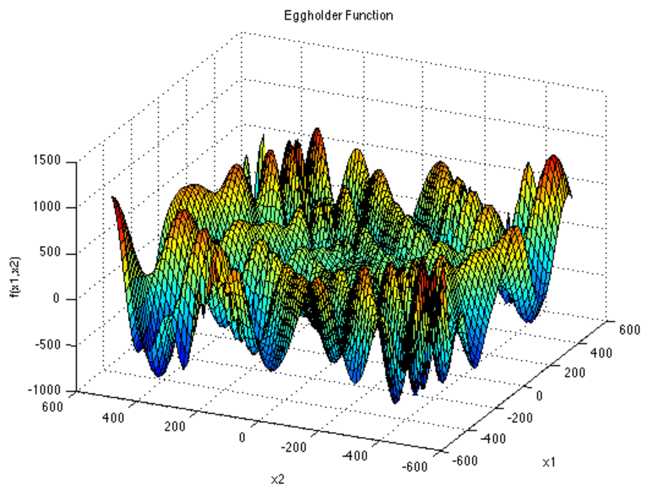


Figure 3,3,6: A visual representation of the Eggholder function [5]

Visually, Eggholder takes on a much wilder appearance than Ackley's function does. This makes it a lot tougher on optimisation techniques due to the heightened probability of getting trapped. While this doesn't necessarily convert to higher computational expense under my requirements, it does offer a larger search space which should provide deeper problem precision for me to exploit.

Unfortunately, despite heavy research, I have been unable to discover the origins of the Eggholder function. All my information, including the equation and visual models have to therefore come from Vanaret's work at the Université de Toulouse [32].

To test my cluster, I will be utilising the Eggholder function. I intend on searching the full search-space at all times, introducing nodes one by one to measure performance change as process counts increase. This will give me a suitable data-set with which I can scientifically analyse.

Implementing a fitness optimisation function: My intended plan of attack

In order to create a piece of software which satisfies all the requirements while using the Eggholder function, I need to design with brute forcing in mind. In short, brute-forcing a fitness optimisation function differs from the function's intended purpose in that with an optimisation method one would aim to find the best optima as quickly as possible. In brute-forcing, the goal is to check every point on a grid given certain boundaries and precision values.

Eggholder allows for incredibly high precision (to confidently locate the global optima one would need to run approx. 26 214 400 000 000 calculations of the function's equation). If one was to take leverage of this function's complexity in conjunction with proper job division, a true parallel system in MPI could be achieved.

My plan in designing this system is to use a hybrid-master slave system on all nodes, rather than relying on job dividing and distribution using a centralised architecture. I believe using this design principle will simplify my codebase as well as speed up my cluster. If each node can compute its own problem bunch without external influence, bandwidth and computational expense is reduced.

If possible, MPI should only be responsible for two tasks. First and foremost, MPI should indeed be responsible for runtime management which includes process spawning and communications ('world management'). Secondly, MPI should facilitate the passing of results and only results from all nodes back to the parent (i.e. node 0). It is also worth noting that the combining of results from child nodes should be the only time the parent node is seen as the master.

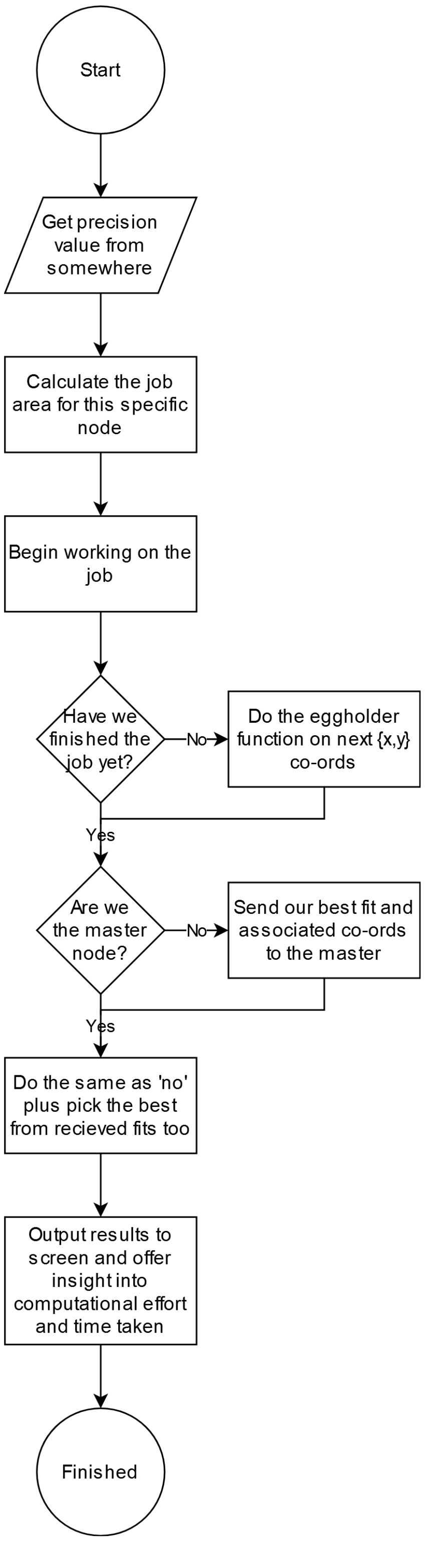

Start
Get precision value from somewhere
Calculate the job area for this specific node
Begin working on the job
Have we finished the job yet?
No
Do the eggholder function on next {x,y} co-ords
Yes
Are we the master node?
No
Send our best fit and associated co-ords to the master
Yes
Do the same as 'no' plus pick the best from recieved fits too
Output results to screen and offer insight into computational effort and time taken
Finished

Figure 3,3,7: UML diagram of my intended software architecture

As illustrated above, most of the process is handled internally by the individual processes. I hope that by utilising this design it is ensured that any reduction in speedups seen are purely based on hardware-induced granularity rather than avoidable network delays. Another feature worth noting in my design is message simplification by design. Rather than sending messages to the master every time a new best fit is found, each new best fit is stored in the node's own individual memory space. The best fit is sent back to the master at the end of the process before being compared against and stored or discarded depending on its quality comparable to prior results received.

The last two aspects of my design I intend on discussing are measurability and verbosity. Addressing the former first, I intend on measuring only one metric, that being time taken to complete the job. This can be achieved using a chrono clock in C++, most likely placed immediately before and after the loop responsible for the brute force process. By measuring the time taken to complete the job on the master, I will hopefully get a suitably accurate representation of how the process speeds change across the cluster as job sizes shrink with each new node.

Addressing verbosity, I plan on presenting several information points to help measure the inner-workings of my program. These values are as follows: computational effort per node, approximate cluster effort, delta time for master node (as discussed), the best fit, the best fit location and the node which located the best fit.

Hopefully with this design I can achieve my goal set forth in Chapter One. In the following chapter, I will explain how I go about implementing this project, including a breakdown of the hardware setup, software installation, creation and testing as well as a discussion on any problems that arise.

Chapter 4

Implementation and Testing

Design complete with research in hand, I am now ready to begin implementation. I intend on completing this process in three distinct stages: virtual hardware creation & configuration, guest software configuration and lastly custom software creation.

I expect the first two stages should complete without much upset and can probably done in a nonrepetative fashion for the most part. The last stage has additional sufficient complexity and uniqueness that leads me to conclude this part of the process will almost definitely end up being considerably more iterative in nature.

Once implementation has been achieved, I will test my system by running my program multiple times under increasing node counts. This process will be outlined in more detail in the second section of this chapter.

4,1 Implementing a Beowulf cluster: Configuring and preparing

Setting up and configuring the virtual switch & nodes

The first step I am taking in setting up and configuring my cluster is hardware preparation. Specifically, I need to make sure the network stack on the hypervisor is up to the task I will be setting for it. As set forth in my design document, the Beowulf cluster I am building will have its own dedicated switch away from the main home network to reduce packet impedance and collision.

To configure a second virtual switch on my hypervisor, I will need to modify the interfaces file found in the Debian file system running on the host. Within, I am adding a carbon copy block from the original virtual switch ('bridge').

```
[...]
auto vmbr0
iface vmbr0 inet static
  address  172.16.1.1
  netmask  16
  gateway  172.16.0.1
  bridge-ports bond0
  bridge-stp off
  bridge-fd 0
#INET

iface vmbr0 inet6 static
  address  2001:470:1f1d:ef5:1::1
  netmask  64
  gateway  2001:470:1f1d:ef5::1

auto vmbr1
iface vmbr1 inet static
  address  192.168.56.1
  netmask  24
  bridge-ports none
  bridge-stp off
  bridge-fd 0
#LOC

iface vmbr1 inet6 static
  address  2001:56::1
  netmask  64
[...]
```

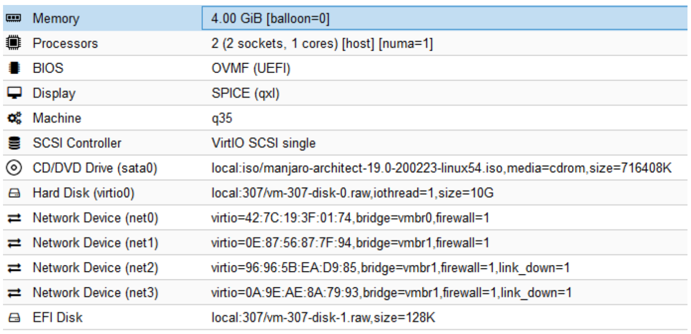
Live snip from my interfaces file

The only major changes one can see between the original virtual switch (‘vmbr0’) and the new virtual switch (‘vmbr1‘) are the changes in the address subnets used between switches and the absence of a physical network port or gateway on the new switch as these features are not required in this particular usecase.

After a reboot, the hypervisor will recognise and auto-configure the network switch and make it available for use. When it comes to configuring the virtual machines, the only step that will be required is to configure the preconfigured virtual network interface cards (NICs) to use an IP on the same subnet as the host. Routing will be handled by the RIP protocol without additional instruction.

Network configuration complete, the other step required in the virtual hardware configuration stage is to create the virtual guests. In Proxmox VE, this is completed within the GUI.

On creation of a virtual machine, both in Proxmox VE and most other hypervisors, a wizard presents itself to step the user though various configuration options. This is ideal for my usecase as I can use the options decided upon during the design stage of this project to optimise my cluster for performance.



Memory	4.00 GiB [balloon=0]
Processors	2 (2 sockets, 1 cores) [host] [numa=1]
BIOS	OVMF (UEFI)
Display	SPICE (qxl)
Machine	q35
SCSI Controller	VirtIO SCSI single
CD/DVD Drive (sata0)	local:iso/manjaro-architect-19.0-200223-linux54.iso,media=cdrom,size=716408K
Hard Disk (virtio0)	local:307/vm-307-disk-0.raw,iothread=1,size=10G
Network Device (net0)	virtio=42:7C:19:3F:01:74,bridge=vmbr0,firewall=1
Network Device (net1)	virtio=0E:87:56:87:7F:94,bridge=vmbr1,firewall=1
Network Device (net2)	virtio=96:96:5B:EA:D9:85,bridge=vmbr1,firewall=1,link_down=1
Network Device (net3)	virtio=0A:9E:AE:8A:79:93,bridge=vmbr1,firewall=1,link_down=1
EFI Disk	local:307/vm-307-disk-1.raw,size=128K

Figure 4,1,1: Screenshot of guest OS hardware configuration

As seen in the screenshot above, the end configuration is geared as best as it possibly can be for optimum performance. All network adaptors and drives are connected to the VirtIO bus, a generous memory count has been allocated and an up-to-date BIOS and chipset is being simulated.

With the primary node configured, cloning of the configuration is completed by using the hypervisor’s build-in cloning tool. Hard disk cloning will be done at a later point once the operating system has been fully configured and is ready for the custom software package.

Setting up and configuring the guest OS: Setup

With my virtual guest machine booted and confirmed functional, the next stage was to install Manjaro using the Architect installer. Manjaro’s Architect installer allows full customisation over what packages and features get installed to the system initially. To avoid bloat, I will be keeping the master image as minimal as possible.

On initial boot of the ISO, I am presented a series of questions including what timezone I wish to use, the language and keyboard settings I wish to use and the driver suite I would like to choose. Naturally, regional questions will be set to British English in this case and require no further explanation, however the driver suite I want to use is worth additional discussion.

Linux drivers typically tend to what are referred to as ‘free’ drivers. Free drivers are any drivers that are based wholly on free and open-source software (FOSS). As a FOSS OS, Linux usually has a free driver for every major brand and kind of PC hardware available and in my system this is no exception. The only hardware that might be unusual in this virtual environment is the VirtIO suite of devices.

Upon deeper inspection of the VirtIO device suite, it has become apparent to me that it is in fact part of the Linux KVM module which is in turn included in the Linux kernel. Source code is available via the Linux kernel repositories. This means that I can boot with free drivers and not lose functionality or performance in my system. Addendum to that, the Spice GPU and driver are not fundamental to the performance of a CLI-based OS and can be suitably supported by the basic Linux video driver suite.

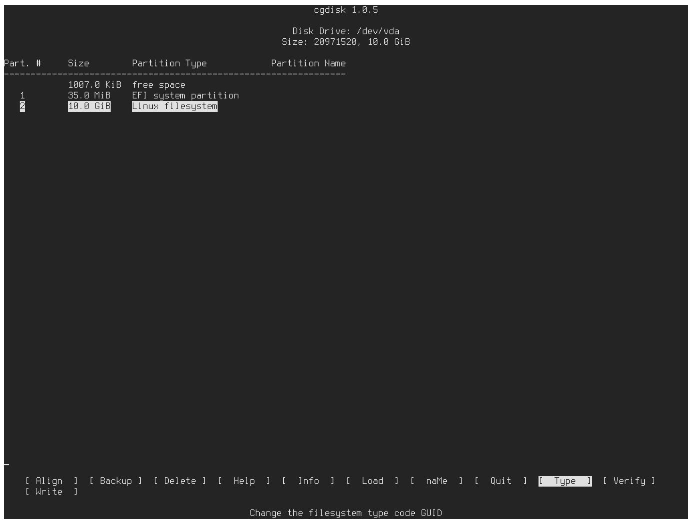
Once the system has been booted, the installer automatically starts a shell with root permissions. My first job is to populate pacman’s key-ring and launch the setup process. To do this, I ran three commands listed below.

```
# pacman-key --init

# pacman-key --populate
 [...]
gpg: next trustdb check due at 2020-03-31

# setup
updating the installer...
 [...]

[Setup has now launched]
```

Pre-setup commands

Within the setup, I am met with numerous options, many of which do not necessarily apply to me. To keep this write-up as simple as possible I will explain only the steps that apply to me, starting with disc partitioning. Architect offers automatic partitioning as a preparatory option, however this involves creating a swap drive which, once configured, is difficult to modify.As I have mentioned, I may want to modify the amount of RAM I allocate to each of my nodes. This is much easier to do by creating a swap file on the root partition instead.

My best option is therefore to create my own partition table using the 'cgdisk' utility. My partition table needs to account for an EFI boot volume, of which I will allocate 35 MB of space for. The rest of the disc becomes my root partition where the operating system will reside.



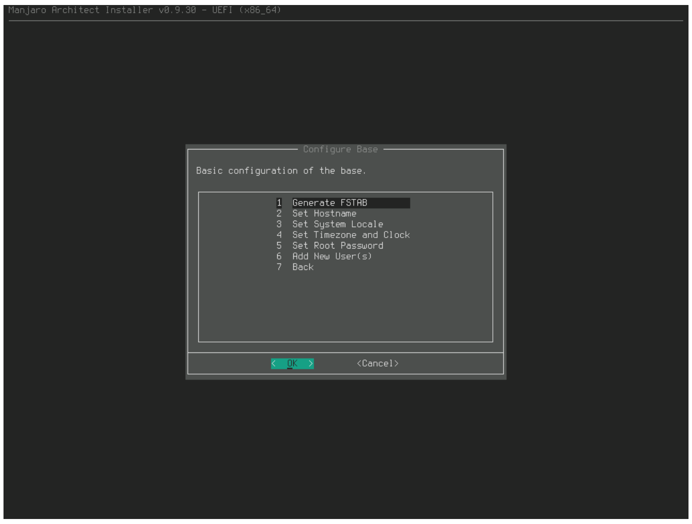
```
                              cgdisk 1.0.5

                          Disk Drive: /dev/vda
                        Size: 20971520, 10.0 GiB

Part. #     Size        Partition Type            Partition Name
----------------------------------------------------------------
            1007.0 KiB  free space
   1        35.0 MiB    EFI system partition
   2        10.0 GiB    Linux filesystem

  [ Align  ] [ Backup ] [ Delete ] [  Help  ] [  Info  ] [  Load  ] [  naMe  ] [  Quit  ] [  Type  ] [ Verify ]
  [ Write  ]

                       Change the filesystem type code GUID
```

Figure 4,1,2: Screenshot of completed 'cgdisk' process

Partition table created and file systems applied, the next step is to mount the partitions at their correct locations prior to installation. Architect has a wizard which helps with this process, all I am required to do is carefully ensure I select the correct mount points for the relevant volumes. The same wizard has also allowed me to create a swapfile on the root partition and has added it to the '/etc/fstab' file which manages partition mounting on boot.

With the hard discs ready, I am in a position to begin the installation process for Manjaro. Before I can start customising my setup however, I need to install the base packages. The wizard process for this step asks some basic questions such as which kernels to use and what development packages you would like if required. For this stage, I chose the latest stable kernel (v5,4) and the base-devel package for installation. I also chose to install kernel-headers with free drivers. This whole process took approximately two minutes.

After completing the base system installation, I'm now ready to install a bootloader. This step is mandatory if I want to be able to boot the operating system once I am finished with setup. Architect offers the 'refind' bootloader typically used to install Linux on older Mac machines as well as the 'systemd-boot' module, however most Linux installations nowadays use the 'grub2' bootloader as it is well-developed and "just works". This is the option I am choosing.



```
Manjaro Architect Installer v0.9.30 - UEFI (x86_64)

Configure Base
Basic configuration of the base.

1  Generate FSTAB
2  Set Hostname
3  Set System Locale
4  Set Timezone and Clock
5  Set Root Password
6  Add New User(s)
7  Back

<  OK  >        <Cancel>
```

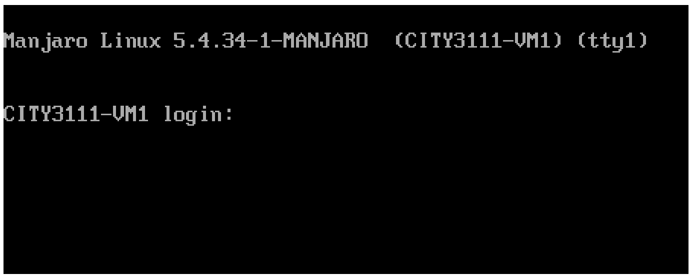
Figure 4,1,3: Screenshot of base configuration menu

Once the bootloader is installed, the final stage in Architect's setup process I will be making use of is the base configuration. These steps, supposedly meant to be done in order, will prepare the system for general usage. While most of these steps are unimportant, there are three specific steps which I need to discuss.

The first, the system host-name, will henceforth become 'CITY3111-VM*n*' where *n* represents the node number one-counting. This value will be important going forward, and will ultimately end up being unique per-system. Without this value being correctly configured or non-unique on the network, inter-nodal communications will be impossible.

The second step is setting the root password. I will need to set this to something secure as the root user ultimately has full power in Linux to do as it pleases. Once my regular user has sufficient super user access, I will disable logon to root with a password for security reasons.

The final step I am configuring is a new user account. Creating a user account on Architect is slightly more involved than just setting the password on root as it requires a decision on the user shell in use. While 'bash' is the typical Linux shell, 'zsh' is gaining popularity on other UNIX and UNIX-like OSes. I have experience using 'zsh' so this is the shell I am choosing. After entering a password, setup informs me the user is ready for use on the next boot.



```
Manjaro Linux 5.4.34-1-MANJARO  (CITY3111-VM1) (tty1)

CITY3111-VM1 login:
```

Figure 4,1,4: Screenshot of fully booted Manjaro system

Rebooting concludes the setup portion of this section, where I have installed Manjaro using minimal settings via the Architect setup and configured a user for login. The next stage is to install the packages I need and configure the system so it is suitable for use with any custom software I wish to write and compile.

Setting up and configuring the guest OS: Configuration

With my guest OS installed and booted and my root user logged in, the first step I need to undertake is a full OS upgrade check. This process is handled by Manjaro's package manager, 'pacman'. The -Syuu flag will synchronise the system's package database with the package database found at the nearest Manjaro mirror and offer to install any upgrades found.

```
[root@CITY3111-VM1 ~]# pacman -Syuu
:: Synchronising package databases
   ...
 core is up to date
 extra is up to date
 community is up to date
 multilib is up to date
:: Starting full system upgrade...
 there is nothing to do
```

Output of the 'pacman -Syuu' command

As is typical with fresh installs of many Linux distros, the system was already up-to-date, however it is still best practice to ensure the local database is in sync with the local mirror to greatly reduce the likelihood of attempting to get an out-of-date package. Going forward, I will install packages only using the '-S' flag with pacman to speed up the sync process.

Package database up-to-date, I am ready to begin installing software. The very first package I want to install is 'sudo' which allows for protected super user access from non-root users. Sudo is considered more secure than consistent use of the root user as it only affords the target process elevation rather than the entire session. To install this, I ran 'pacman -S sudo'.

With 'sudo' installed, the last two steps in allowing my user super user access are to add my user to the 'wheel' group and allow said group sudo access. To begin with, I add my user to the group by running 'usermod -a -G wheel axessta'. With the user now in the required group, I lastly modify the '/etc/sudoers' group using the 'nano' command.

```
[root@CITY3111-VM1 ~]# nano /etc/sudoers
[...]

## Uncomment to allow members of group wheel to execute any command
%wheel ALL=(ALL) ALL ## was previously commented

[...]
```

Snipped contents of the '/etc/sudoers' file

Once this step has been completed, I can test my user's permissions by logging out of the system and back in with the regular user. If the process has been completed successfully, I should be able to edit the '/etc/shadow' file using 'sudo nano /etc/shadow'. I found the 'usermod' command to be unreliable, and it took several runs for the system to register my user as a sudoer. I am unsure if this issue is directly related to 'usermod' or if 'sudo' incurs a delay either through a bug or by design, however eventually this process worked and I could modify the root user.

With functional sudo access, password-based login to the root user is no longer a desirable trait of my operating system so I wish to disable it. To do so, I need to delete the text hash for my root's user and replace it with an asterisk. In Linux's 'shadow' authentication process, an asterisk is seen as an active user without password privileges, while an exclamation point is seen as a disabled user, double-exclamation point as a blocked user and a blank field as password-free access. For my purposes, an asterisk is suitable as there may be times I still want to access the root user using 'sudo su root'.

```
[root@CITY3111-VM1 ~]# nano /etc/shadow
root:*:18326::::::
[...]
```

Snipped contents of the '/etc/shadow' file

With a super user configured, my next logical step is to configure a static network addressing structure based on the framework specified in my Design chapter. My node featured four adaptors as seen in Figure 4,1,1, labelled ens18-21. It would therefore be logical to assume that end18 mapped to my active connection through to 'vmbr0' and ens19 mapped to 'vmbr1'.

To configure the adaptors I used the 'systemd-network' configuration method standard in Arch Linux. With 'ens18', I need to ensure a suitable connection to the landline is maintained so I will include DNS and gateway configuration details. With 'ens19', only a suitable subnet and address was required to facilitate inter-nodal talks.

```
[Match]
Name=ens18

[Network]
Address=2001:470:1f1d:ef5:3::7/64
Address=172.16.3.7/16
#Gateway=2001:470:1f1d:ef5::1
Gateway=172.16.0.1
DNS=2001:470:1f1d:ef5::1
DNS=172.16.0.1
```

Full contents of the '/etc/systemd/ens18.network' file

```
[Match]
Name=ens19

[Network]
Address=2001:56::5/64
Address=192.168.56.5/24
```

Full contents of the '/etc/systemd/ens19.network' file

With a super user configured and the network functioning, I now need to install my packages. As I may require the ability to download additional files directly from the web, I will install 'wget'. As well as this, I will be cloning files from my repository so 'subversion' is another dependency for my project. Lastly, I will install the Open MPI package which, on Arch, includes the runtime and compiler.

To install these scripts, I made use of the '-S' pacman flag which synchronises the target package with the external mirror service as well as the '–noconfirm' flag which installs a package without requiring keyboard input. Additionally, I chose to enable the 'sshd' service with systemd so I could initiate SSH communications between nodes. It is worth noting here however that once I clone my other nodes the SSH package will need new certificates in order to avoid insecurities and certificate clashing.

```
pacman -Syu --noconfirm
pacman -S wget --noconfirm
pacman -S subversion --noconfirm
pacman -S openmpi --noconfirm
[...]
systemctl start sshd.service --
    force
```

Live snip of my 'pacman' commands

The last step I am able to complete uniformly on the master which can be passed evenly amongst the nodes without major reconfiguration is the hosts file. The hosts file keeps a local list of hostnames associated with IP addresses, a bit like how a DNS server would distribute domain names to clients over TCP/IP without the need for an external server. This will be useful as I won't need to manually type out IP addresses each time I wish to change the node I am connected to.

To achieve this, all I need to do is list the IPs of each node in order and the associated hostname, divided by a tabulator.

```
::1         localhost.localdomain
    localhost          CITY3111-VM1

2001:56::5   CITY3111-VM2
2001:56::6   CITY3111-VM2
2001:56::7   CITY3111-VM3
2001:56::8   CITY3111-VM4
2001:56::9   CITY3111-VM5
```

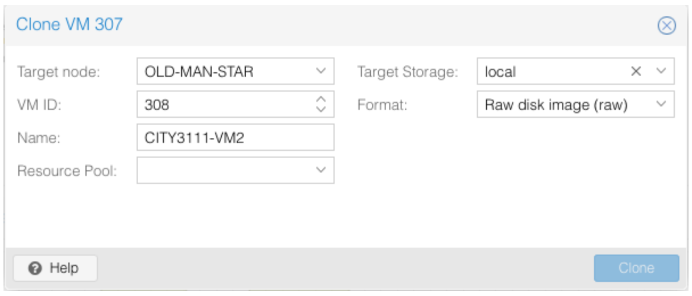
Full contents of '/etc/hosts' file

Setting up and configuring the guest OS: Cloning & Tidy-up

With my operating system now fundamentally ready for service, the last step I need to undertake is cloning and tidy-up. I expect this to be a four-pronged attack, with the the notable stages being cloning, renaming, re-addressing and re-certification.

For each of these steps, I will demonstrate how the process is completed on one node. Please bare in mind that the process is in fact going to be completed on the four additional nodes in sequence.

Firstly, I need to clone my master node out into the four other nodes. In Proxmox VE, a wizard is available which will automatically handle file replication and configuration matching. In the past I have found this option to be reliable so I will utilise it today.


Clone VM 307

Target node: OLD-MAN-STAR
VM ID: 308
Name: CITY3111-VM2
Resource Pool:
Target Storage: local
Format: Raw disk image (raw)

Figure 4,1,5: Screenshot of VM cloning using Proxmox VE

The wizard takes approximately five minutes to complete per node and, once done, boots the newly created node for you. As we currently have two nodes with conflicting preconfigured hostnames and IP addresses, I promptly halted each node in turn to avoid problems arising.

With the nodes clones, I now completed the following steps on each node in turn, always with the master node offline.

Now my nodes all exist and are accounted for, I proceeded to move onto the next stage which is renaming. To rename a system running Linux, two values must be altered. The first value is the associated hostname found on line one of '/etc/hosts' and the second is the contents of '/etc/hostname'.

Taking the '/etc/hosts' example from above first, one can see the last string on line one is 'CITY3111-VM1'. For node two, this will become 'CITY3111-VM2' and so on. Within '/etc/hostname', the same rule applies except the entire contents of the file must contain only our new system name.

Moving on to re-addressing of the system, this stage is essentially just as simple as the previous. All I am required to do is modify the two files I created in '/etc/systemd/network' and increment each node by one from its neighbour. For example, the IPv6 addresses of node one, which are '2001:470:1f1d:ef5:3::7' and '2001:56::5', become '2001:470:1f1d:ef5:3::8' and '2001:56::6' and so on.

With these two steps complete, a reboot of each node has ensured the systems are configured correctly and no IPs are overlapping. I have proven this by pinging the 1.1.1.1 and 192.16.56.1 IPs from all nodes simultaneously. If there were any hostname or IP address overlaps, one or more of the nodes would fail to connect to the network(s) and the pings would fail.

I am now ready to complete the final step, which is to regenerate the SSH keys for each node. While this step is somewhat optional and rarely causes issues, it will increase the security of my system and avoid certificate clashes which can happen with certain SSH clients or environments.

To renew the SSH certificates, I need to stop the 'sshd' service, remove and regenerate the SSH certificates and restart the service. To make this process faster, I wrote a basic shell script to do the job for me.

```
#!/bin/sh

# sudo warning
echo Please ensure this script runs
    as SUDO or root user

# halt sshd
systemctl stop sshd.service

# remove certs
rm /etc/ssh/ssh_host_*

# regen certs
ssh-keygen -t dsa -f /etc/ssh/
   ssh_host_dsa_key
ssh-keygen -t rsa -f /etc/ssh/
   ssh_host_rsa_key
ssh-keygen -t ecdsa -f /etc/ssh/
   ssh_host_ecdsa_key

# start sshd
systemctl start sshd.service

#
# done
#
```

Full contents of 'ssh-regen.sh' file

A correct start of SSH demonstrates the service is online and running. The final, short stage of the cluster's preparatory cycle is to copy my SSH keys between nodes.

Setting up and configuring the guest OS: Exchanging SSH keys between nodes

With SSH now online and running across all nodes, I finally come to the final portion of this chapter which is the exchanging of SSH keys. In order for SSH to authenticate without prompting for a password – a prerequisite of MPI – I need to create a public and private key pair and ensure each node has the public key for the master.

This means that the master will be able to initiate SSH connections on demand without prompting the user for a password on connection. To achieve this process, I will run the command below and answer the prompt with default values:

```
ssh-keygen -t ecdsa -b 256
```

Single-line command

This command will generate a key pair, however it has not yet left the local machine. In order to duplicate it across all nodes, I need to run the following:

```
ssh-copy-id -i ~/.ssh/id_ed25519.
   pub city3111-vm2
```

Single-line command

Please bear in mind that the hostname must be changed for each new target node. Once this step has been completed, my cluster can now freely communicate whenever a master node initiates a new node call.

4,2 Implementation and Testing: Creating my custom software

With my cluster environment configured, I am ready to program for the MPI runtime. To do this, I intend on compiling two programs. The first is a basic hello world program [3] which I will use to test my cluster functions as intended and all nodes respond to commands. The second program is a currently abstract custom software suite which will brute-force the eggholder function, as discussed in Chapter 3.

I hope this process to be a straight-forward one, however there is a high likelihood I may need to iterate my program several times in order to get it to work as I intend before I can conclude this chapter and move on to my review. As discussed in the second chapter, my custom software will be required to maximise CPU usage while measuring the cluster's performance so it is imperative the software is efficiently designed.

My initial cluster test: Using hello world to ensure all nodes are online

To ensure the cluster is functioning as I intend it to, I need to use the Hello World code I previously sourced and placed in Appendix α, originally completed by the Open MPI community [11].

To streamline deployment, I have set up a development suite on my Manjaro-based laptop using the same packages beneath the Apache NetBeans IDE with the C/C++ extensions. Using this system, I can program, compile and test software from the comfort of a GUI before distributing an archive file to each system for extraction and running.

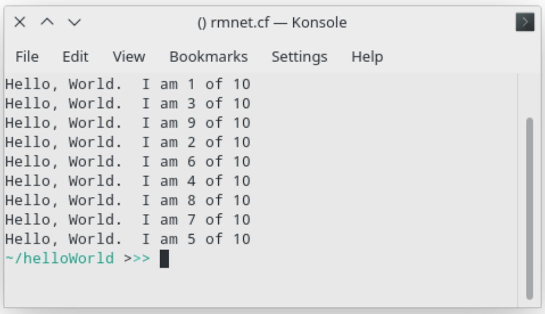
To commence my programming effort, I have compiled the source code on my laptop, as well as written a 'hostfile' and run bash script to hook the MPI runtime to the binary. The contents of the non-binary files are listed below.

```
# cat ~/helloWorld/hostfile
city3111-vm1   slots=2
city3111-vm2   slots=2
city3111-vm3   slots=2
city3111-vm4   slots=2
city3111-vm5   slots=2

# cat ~/helloWorld/run_mpi.sh
mpirun -np 10 -hostfile ./hostfile
    ./main
```

Combined 'hostfile' and 'run_mpi.sh' files

Notice how in file one, each node's hostname is suffixed with the flag 'slots=2'. This tells Open MPI there are two logical processors available on each node, in line with what we defined earlier in this chapter during configuration of the guest virtual machines.

Also, see how in the second file we've specified a flag of '-np 10'. This directly correlates to the number of logical processors in the cluster, where $2 \times 5 = 10$, while the last argument points to our binary's directory relative to the script (in this case, './main').



```
Hello, World.  I am 1 of 10
Hello, World.  I am 3 of 10
Hello, World.  I am 9 of 10
Hello, World.  I am 2 of 10
Hello, World.  I am 6 of 10
Hello, World.  I am 4 of 10
Hello, World.  I am 8 of 10
Hello, World.  I am 7 of 10
Hello, World.  I am 5 of 10
~/helloWorld >>>
```

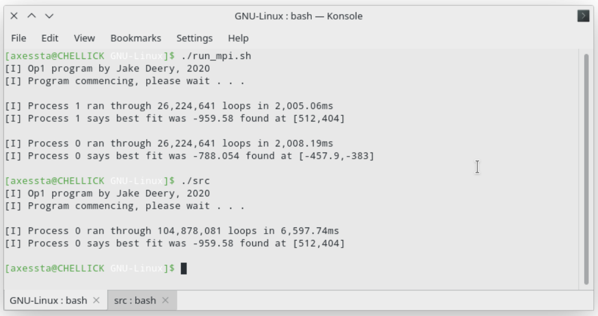
Figure 4,2,6: Screenshot of successful Hello World execution

Upon running the script, I am presented with a printout from each node specifying its identification number ('rank') and its location in the cluster as a while (where the cluster size is represented by 'size' in code). This run confirms the cluster is operational and concludes my experimentation with the 'hello world' program.

Initial functioning code: Revision 5, problems with effort shrink and other areas to improve

Before I commence a discussion on the evolution of my program over its development timeline, please note that the final code will be placed in full within Appendix β if you wish to bypass this section and read it directly.

Revision five marks the first working and compiled version of my program. To begin distributing the source code, I will checkout copies to each node. In future, to update the cluster all I will need to do is recompile from source on my laptop and update the subversion (SVN) folder on each node.



```
[axessta@CHELLICK GNU-Linux]$ ./run_mpi.sh
[I] Op1 program by Jake Deery, 2020
[I] Program commencing, please wait . . .

[I] Process 1 ran through 26,224,641 loops in 2,005.06ms
[I] Process 1 says best fit was -959.58 found at [512,404]

[I] Process 0 ran through 26,224,641 loops in 2,008.19ms
[I] Process 0 says best fit was -788.054 found at [-457.9,-383]

[axessta@CHELLICK GNU-Linux]$ ./src
[I] Op1 program by Jake Deery, 2020
[I] Program commencing, please wait . . .

[I] Process 0 ran through 104,878,081 loops in 6,597.74ms
[I] Process 0 says best fit was -959.58 found at [512,404]

[axessta@CHELLICK GNU-Linux]$
```

Figure 4,2,7: Screenshot of initial functioning test run

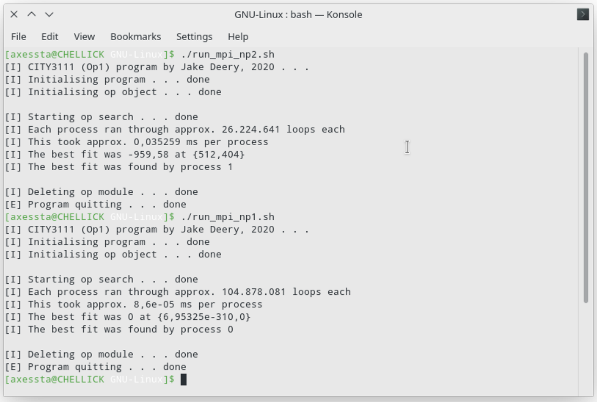
This initial revision of my software appears to work in principle, with both of my development machine's CPU cores spinning up to 100% for the 5 700 ms in which the program runs for. The inceptive promise is, however, dampened by several issues both in terms of functionality and semantics.

Addressing firstly the functionality of my program, the overall verbosity isn't yet up to the standards I wish it to be. Currently, I am only being presented with statistics on each node's effort, timings, best fit value and co-ordinates. While this data is entirely acceptable and I don't need anything else in terms of content, what will not be acceptable as I add more nodes is the lack of combined metrics.

The semantic issues that don't necessarily affect program functionality but can be defined as bugs are related to the underlying mechanics of my software. For some reason which I will need to investigate and rectify, my program is reporting conflicting total effort values. When running under a single node, my cluster appears to take 104 878 081 program loops to complete its search while two nodes combine for a computational effort of 52 449 282 program loops, somewhere around half of the total effort. This means approximately half of the search space is not being addressed in the latter case. Additional inspection shows that this effect is exponential.

Progress report: Revision 10, functional improvements and object-oriented programming

This portion of my implementation represents a step towards my target program, mainly due to the functional improvements and improved coherence and efficiency in the underlying codebase. A majority of the codebase has been rewritten to facilitate better message-passing using the MPI_Send and MPI_Recv methods.


GNU-Linux : bash — Konsole
File Edit View Bookmarks Settings Help

```
[axessta@CHELLICK GNU-Linux]$ ./run_mpi_np2.sh
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done

[I] Starting op search . . . done
[I] Each process ran through approx. 26.224.641 loops each
[I] This took approx. 0,035259 ms per process
[I] The best fit was -959,58 at {512,404}
[I] The best fit was found by process 1

[I] Deleting op module . . . done
[E] Program quitting . . . done
[axessta@CHELLICK GNU-Linux]$ ./run_mpi_np1.sh
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done

[I] Starting op search . . . done
[I] Each process ran through approx. 104.878.081 loops each
[I] This took approx. 8,6e-05 ms per process
[I] The best fit was 0 at {6,95325e-310,0}
[I] The best fit was found by process 0

[I] Deleting op module . . . done
[E] Program quitting . . . done
[axessta@CHELLICK GNU-Linux]$ 
```

Figure 4,2,8: Screenshot of improved program test run

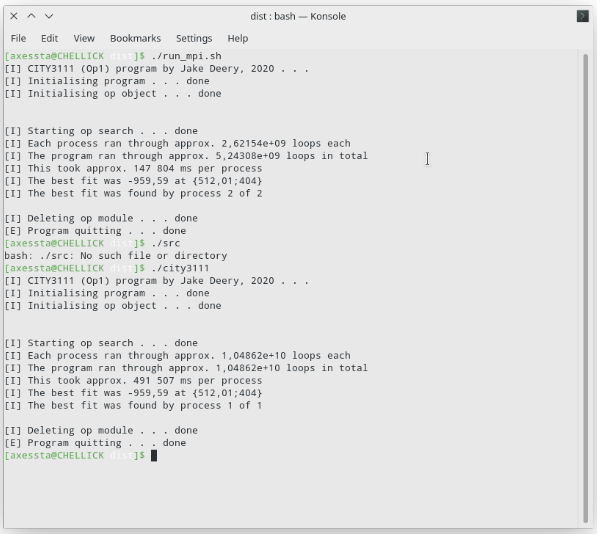
The resultant program is now much cleaner in its verbosity and has a slightly smaller memory footprint (2 612 KiB for r5 vs. 2 608 KiB for r10) thanks to its conversion from a C program (procedural) to a C++ one (objective). Despite these improvements, existing semantic bugs are unresolved and new bugs have appeared.

The most obvious issue presents itself when looking at the single-node execution, where the apparent best fit is zero at an indeterminate set of co-ordinates. This is likely due to the fact that I put a lot of the program's code inside a set of if statements that change how each node behaves (specifically, I have defined a master/slave architecture against how I initially planed to).

To resolve this, I will need to amalgamate the thought process I was using up to revision five with the improvements and concepts I have learned since. As well as this, I will need to start addressing the semantic issues that have plagued the project up until this point.

Approaching completion: Revision 30, a stable codebase and additional functional improvements

Since making my software object-oriented, I have taken advantage of new techniques at my disposal to further improve the program's efficiency, including making greater use of private variables and function arguments to make modifying variables better and to create a better, more readable flow of code.



```
[axessta@CHELLICK dist]$ ./run_mpi.sh
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 2,62154e+09 loops each
[I] The program ran through approx. 5,24308e+09 loops in total
[I] This took approx. 147 804 ms per process
[I] The best fit was -959,59 at {512,01;404}
[I] The best fit was found by process 2 of 2

[I] Deleting op module . . . done
[E] Program quitting . . . done
[axessta@CHELLICK dist]$ ./src
bash: ./src: No such file or directory
[axessta@CHELLICK dist]$ ./city3111
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,04862e+10 loops each
[I] The program ran through approx. 1,04862e+10 loops in total
[I] This took approx. 491 507 ms per process
[I] The best fit was -959,59 at {512,01;404}
[I] The best fit was found by process 1 of 1

[I] Deleting op module . . . done
[E] Program quitting . . . done
[axessta@CHELLICK dist]$ 
```

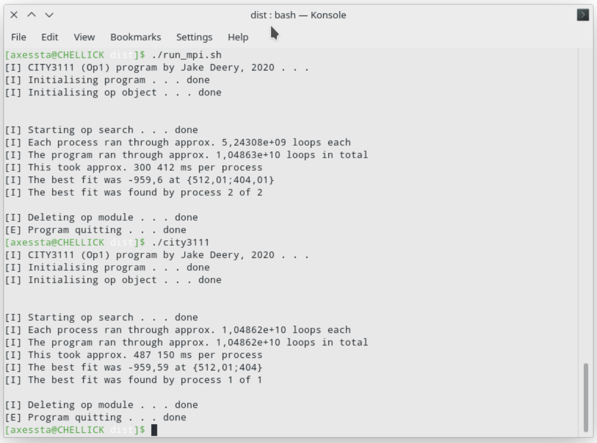
Figure 4,2,9: Screenshot of latest program test run

One example of improvements I have made in the last 20 revisions is the precision float I have added, which now makes it possible to define the precision of my search without needing to modify the object itself, as I was before. Another functional change I have made is further increases in verbosity. Before, to calculate the total effort across all nodes, one needed to multiply the printed value by the number of nodes. I have now added a line that approximates the value with a view to make it an accurate representation in the near future.

In a semantic sense, the program has only improved marginally. A singular node is now once again capable of running the process single-handedly, however I am going to need to take a major look into the problem space issues and address them head-on before the next major milestone is reached.

Final software revision: A change in search-space bounding and better total effort counting

I have finally reached a point in my implementation where I can demonstrate a codebase I am happy to say is at a stage of prototyping that is stable and bug-free enough to submit to the next stage – a review of results and discussion on improvements to be made for production.



```
[axessta@CHELLICK dist]$ ./run_mpi.sh
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 5,24308e+09 loops each
[I] The program ran through approx. 1,04863e+10 loops in total
[I] This took approx. 300 412 ms per process
[I] The best fit was -959,6 at {512,01;404,01}
[I] The best fit was found by process 2 of 2

[I] Deleting op module . . . done
[E] Program quitting . . . done
[axessta@CHELLICK dist]$ ./city3111
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,04862e+10 loops each
[I] The program ran through approx. 1,04862e+10 loops in total
[I] This took approx. 487 150 ms per process
[I] The best fit was -959,59 at {512,01;404}
[I] The best fit was found by process 1 of 1

[I] Deleting op module . . . done
[E] Program quitting . . . done
[axessta@CHELLICK dist]$
```

Figure 4,2,10: Screenshot of final program test run

At a functional level, there have been no fundamental changes whatsoever to my software. Semantically however, there are two major changes to the software and how it is addressing the problem which are of interest.

Firstly, rather than approximating the computational effort using the master node's effort value as a basis, each node now communicates its effort counters back for amalgamation. This provides a better overview of any effort shrink as it takes into account the fact that the last node in the cluster will typically have a slightly smaller problem area due to overlap avoidance techniques.

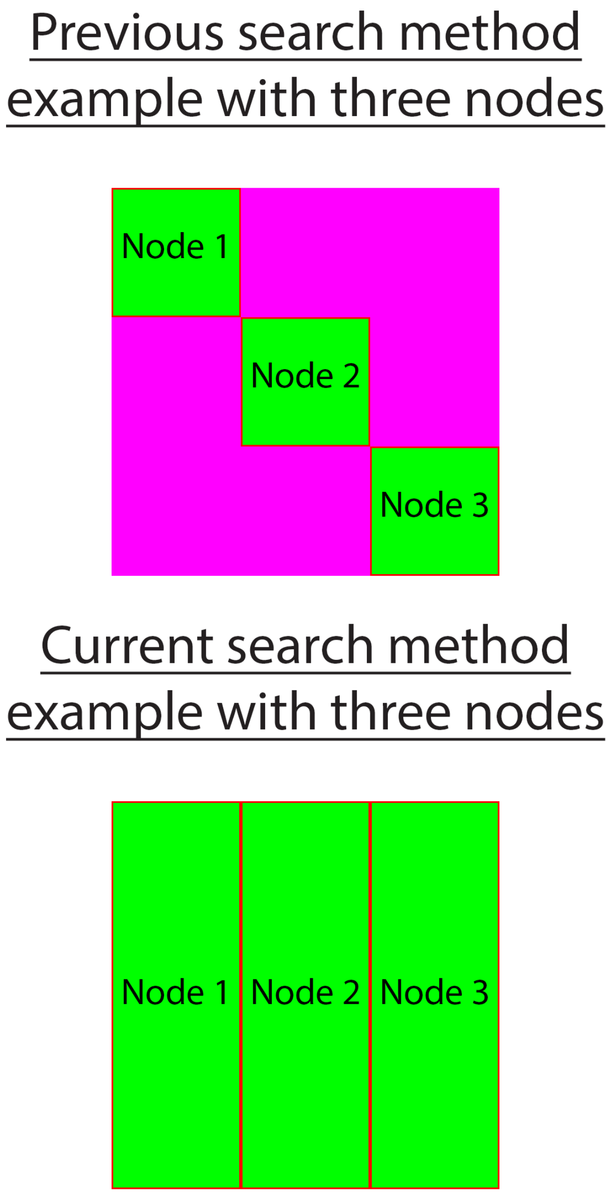

Previous search method example with three nodes
Node 1
Node 2
Node 3
Current search method example with three nodes
Node 1
Node 2
Node 3

Figure 4,2,11: Demonstration of problem area solution for effort creep

Secondly, the method I was previously using to search the problem area has been improved to cover the complete area and reduce (although not completely halt) effort creep. The problem ended up being a programming error on my behalf that was easily fixed. My initial implementation was using square search areas in which the bonding areas for the X-axis were also being used on the Y-axis.

This resulted in a set of squares that, when viewed on a 2D plot, appeared to be mirror copies moving from the top-left to bottom-right (assuming a UV plot where {-512,-512} is in the top-left). The solution I used ended up being to lock the Y-axis search to cover the entire search range from -512 to 512.

This results in a stripe-search pattern when viewed on a 2D plot which gives each node roughly the same problem areas. This also reduces problem shrink to a much lower and more manageable change which should give a more scientific result. Content on the program's ability to perform the benchmarking I need it to, I am now to proceed onto Chapter 5, where I will gather results from the cluster, analyse their meaning and discuss future work I could perform to improve my cluster or its software.

Chapter 5

Results and Discussion

My final chapter, Results and Discussion, aims to offer a comprehensive overview of my system's performance through the acquisition and analysis of results. To acquire these results, I plan on taking a sample of seven results from the system with all five available, followed by a further seven results using only four nodes and so on until there is only one node online.

The values for each cycle will then be averaged and a graph presented. A discussion on this graph will offer insight into the behaviour of my system; my hypothesis being that an increase in performance will be rapid at first but will ultimately be tempered by granularity as resources begin to become scarcer.

After I have discussed the results of my system's performance, I will finalise this chapter by touching on improvements that could be made to my system as well as aspects I believe went well in hindsight.

5,1 Acquisition and analysis of results: Cluster performance measured by time taken

The first stage in my discussion on results is acquisition. In order to collect results with a good degree of provenance, I need to mitigate or remove any environment variables that could cause a drastic change in the system's computing capacity during my code execution.

To this end, I have mitigated for instability by shutting down non-essential machines running on the hypervisor and have run all of my tests during the night-time when load should be much lower. I have also halted my web server's MySQL database and disabled PHP processing so no server pre-processing occurs there. This will, to the best of my ability, remove any Internet load that might affect the server.

Collecting and presenting results: Continued effort creep issues

The results-collecting process took approximately two hours, excluding the momentary pauses that happened while I modified the configuration files between batch runs. The entire, unedited console outputs can be found in Appendix γ, however for convenience I will now list the averages from each configuration.

- **One node** took approx. 426 417 ms to complete approx. 1 048 630 000 000 000 program loops.
- **Two nodes** took approx. 219 426 ms to complete approx. 1 048 670 000 000 000 program loops.
- **Three nodes** took approx. 145 995 ms to complete approx. 1 048 690 000 000 000 program loops.
- **Four nodes** took approx. 110 373 ms to complete approx. 1 048 730 000 000 000 program loops.
- **Five nodes** took approx. 87 939 ms to complete approx. 1 048 770 000 000 000 program loops.

Initial presentation of my results shows promise, with a clear downward trend in the amount of time the search takes place based on the number of nodes online in the system. One might also be able to identify the effects of granularity on my system simply by identifying the closing gap between the average times from each set of runs.

As I have already discussed in detail in Chapter 4, my system is still experiencing effort creep not appearing to be in any logical increments between batch runs but always the same within batches. The creep is typically somewhere in the range of 40 billion loops, give or take, between groups. To calculate how much drift each run may be experiencing, I can work out how much as a percentage the difference between each batch has compared to my single-node system.

- **One node** is our benchmark, with approx. 1 048 630 000 000 000 program loops
- **Two nodes** have crept up by 40 000 000 000 loops from my single-node batch (0,00381% increase)
- **Three nodes** have crept up by 60 000 000 000 loops from my single-node batch (0,00572% increase)
- **Four nodes** have crept up by 100 000 000 000 loops from my single-node batch (0,00954% increase)
- **Five nodes** have crept up by 140 000 000 000 loops from my single-node batch (0,01335% increase)

As one can see, the overall increase in effort, while sub-optimal, does not skew my results by results by more than 0,01% in total. I personally feel that these sorts of values do not pose a substantial threat to my research sufficiently enough to require a revisit of Chapter 4. Indeed, the worst-case scenario changes the results from the five-node cluster by an average of only 11 ms.

Collecting and presenting results: Insight using network statistics

Another metric that may be useful in measuring a Beowulf cluster's performance is its network footprint. This might be its bandwidth usage or the latency between nodes (as posited by Shipman et. al. [10]). This method of measuring a cluster's performance is typical when comparing two cluster APIs but doesn't appear to be common when ascertaining a cluster's computational power.

That in mind, using network metrics can also be informative when measuring granularity. Typically in a standard Beowulf cluster, it is the network which offers the greatest possibility of system bottle-necking. The cluster I have developed does not use standard networking largely due to its virtualised nature. This makes measuring network bottle-necking unnecessary and pointless, as any bottle-neck is going to be inside the CPU or between buses rather than inside the virtual network.

Collecting and presenting results: A visual representation

Returning at last to the discussion on my results, in order to get a better visual idea of what is happening with the timed results, I will now present my data on a line chart. I expect to see a rapid decrease in the time taken to execute my software followed by a plateauing of the speedup, in line with the first list in this section.

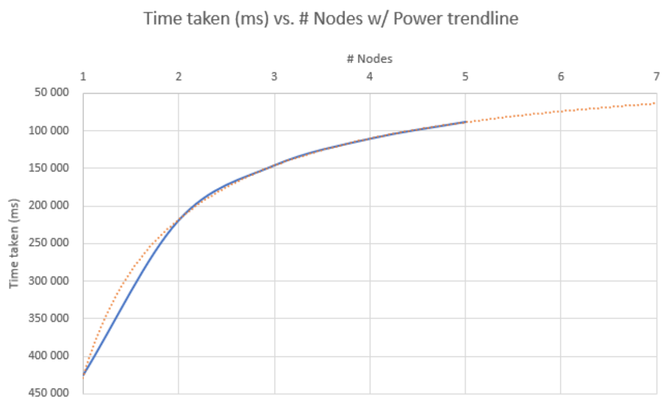


Figure 5,1,1: Line chart demonstrating predicted granularity occurring in my cluster

As suspected, my initial data-points when mapped on a smoothed line graph suggest a clear downward trend in time taken to perform full execution followed by a plateauing. Using a crude trend algorithm of $y = a \times x^b$ to predict a future trend, it can reasonably be posited that in this cluster's case with this particular problem or implementation, more than five nodes would not return a significant enough decrease in execution time to justify its resource usage or time to maintain.

The system I have created demonstrates that simply by combining the computational effort of several interconnected machines of unremarkable quality, a speedup close to the number of additional nodes is possible up to approximately five nodes. In better circumstances, I would acquire dedicated hardware for each node and improve my software further and expect a better speedup across more nodes.

5,2 Brief discussion on project: Comments on project completion & future work

With the conclusion of my project, it is now appropriate for me to retrospectively discuss the events that lead us to this point. Specifically, I intend on discussing points of merit, failings and potential future iterations on this concept.

Views on project & results

HPC is a field of computing that has existed for some time now, indeed nothing in this dissertation can be regarded as novel. That is not the purpose of my writings. What is, however, is my personal knowledge gain and to that end I feel appropriately satisfied with the progress I have made in understanding how Beowulf clusters are constructed, maintained and programmed for.

In my view, the project was largely a success, with my cluster performing well under varying workloads utilising the custom software I designed to run atop the MPI runtime. Improvements can always be made, both in hardware and software, regardless of the complexity of the system or its problem.

For example, utilising a dedicated server rather than independent hardware for each node is sub-optimal for a number of reasons that have been covered in prior chapters, and my software has some clear bugs that would need to be addressed before it could be considered a production tool. However, in the bigger picture presented, these issues are not of detriment to the project's goals.

Future work

This project has a wide scope for future research. I've explored the MPI paradigm as a method of inter-nodal computation in this project, however APIs like GASnet [33] offer a lower-level alternative to what is viewed by some a limiting tool-set.

Possibly using different APIs with more cutting-edge problems would be an area of research I would have a great degree of interest in pursuing. As computer science moves into a world of increasing problem sizes, supercomputer systems are going to be required more than ever to assist in research across an ever-broadening range of subjects. My hope is that this project can become a staging piece for my continued work within this field of academia in due course.

Conclusion

In summarised conclusion of this dissertation, I will now break down what work has been completed to achieve my research goal. I will outline the steps taken through each chapter, including the hurdles and solutions found to the problem or implementation.

In my first chapter, the Literature Review, I discussed the concepts behind parallelism and how it applies to my research goal. MPI was posited as a possible method of achieving my goal. I next discussed in detail the different networking technologies available to cluster developers, recommending TCP for a Beowulf system. I then presented an example implementation of hello world using Open MPI and argued why it would be most suitable for my system. I finally discussed a comparable library for creating a Beowulf cluster and justified my reasoning for using MPI before concluding chapter one.

My second chapter, the Requirements and Analysis discussion, I started by proposing my options with regards to compute hardware, finalising on a dedicated server with virtual nodes due to cost and time constraints. Next, I discussed software I would need, incorporating the existing hypervisor and associated software. I discussed what would be required of my operating system and determined a rolling distribution of Linux would best serve my needs. Finally, I discussed the custom software I would write and any requirements it would have to fulfil, as well as the networking tools and auxiliary components I would require.

The third chapter covered my system design, including both hardware configurations and software setups. To start, I proposed a network topology with five virtual compute nodes, based around existing systems in my network. An initial design of my virtual nodes was presented with justifications for its design and a high-level method of communication was established. I then conducted a small literature review and designed a method of benchmarking my system using a brute-force on the eggholder function with justification for its use over hashing.

The fourth chapter introduced my implementation and testing procedures. First, I configured the virtual network switches for use with my virtual compute nodes and prepared the virtual nodes for operating system use, choosing and justifying the most optimal settings for operations. I then proceeded with installation of Manjaro, using the Architect software to configure a minimal installation. I configured the operating system for use and installed the required packages, being careful not to incur unnecessary bloat while I complete this task. I then began to program my custom benchmarking software, first using the hello world example code to test the system's functionality followed by the chronicling and testing of major program milestones, correcting issues as I went.

In the final chapter, I expanded on chapter four by collecting and discussing results from my benchmarking software. Using time taken for completion as my metric of performance, I discussed issues in effort creep and concluded it did not threaten to majorly undermine my research before visually presenting my results, offering a prediction on the state of my cluster were I to add more nodes. The chapter is concluded by discussing my views on how the project went and any future work I might see in future based on this dissertation's findings.

References

[1] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer, *Beowulf: A Parallel Workstation For Scientific Computation*, vol. 24 of *Proceedings of the International Conference on Parallel Processing*. NW Boca Raton, FL: CRC Press, 1995. pp 11–14. ISBN: 978-0-262-52739-2.

[2] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI - Portable parallel programming with the Message-Passing Interface*. Cambridge, MA: MIT Press, Third ed., 2014. ISBN: 978-0-262-52739-2.

[3] B. Barrett, G. Bosilca, R. Graham, G. Shipman, T. Woodall, and J. Squyres, *Open MPI Developer's Workshop*. Online: Open MPI Project, 2006. Available: https://www-lb.open-mpi.org/papers/workshop-2006/.

[4] S. Surjanovic and D. Bingham, *Ackley Function*. Online: Simon Fraser University, 2017. Available: https://www.sfu.ca/ ssurjano/ackley.html.

[5] S. Surjanovic and D. Bingham, *Eggholder Function*. Online: Simon Fraser University, 2017. Available: https://www.sfu.ca/ ssurjano/egg.html.

[6] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*. Harlow, Essex: Addison-Wesley Pearson, Second ed., 2003. ISBN: 978-0-201-64865-2.

[7] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, vol. 1,0. Online: Message Passing Interface Forum, 1994. Available: https://www.mpi-forum.org/docs/mpi-1.0/mpi-10.ps.

[8] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, vol. 3,1. Online: Message Passing Interface Forum, 2015. Available: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[9] R. L. Graham, T. S. Woodall, and J. M. Squyres, *Open MPI: A Flexible High Performance MPI in Parallel Processing and Applied Mathematics: 6th International Conference*, vol. 3 911 of *Theoretical Computer Science and General Issues*. Poznan, Poland: Springer, 2006. pp 228–239. ISBN: 978-3-540-34141-3.

[10] G. M. Shipman, T. S. Woodall, R. L. Graham, A. B. Maccabe, and P. G. Bridges, *Infiniband Scalability in OpenMPI in Proceedings of the 20th International Conference on Parallel and Distributed Processing*. Rhodes Island, Greece: IEEE, 2006. pp 1–10. ISBN: 978-1-4244-0054-6.

[11] Open MPI Project, *Open MPI v4.0.3 documentation*, vol. 4,0. Online: Open MPI Project, 2020. Available: https://www.open-mpi.org/doc/v4.0/.

[12] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM 3 User's Guide and Reference Manual*. Oak Ridge, TN: Oak Ridge National Laboratory, 1995. Available: https://www.researchgate.net/publication/2650048_Pvm_3_User's_Guide_And_Reference_Manual.

[13] E. Elts, *Comparative analysis of PVM and MPI for the development of physical applications on parallel clusters*. St. Petersburg, Russia: Faculty of Physics, Saint-Petersburg State University, 2004.

[14] A. Trew and L. Clarke, *The Message Passing Interface (MPI): An International Standard for Programming Parallel Computers*. Edinburgh, Scotland: University of St Andrews & University of Edinburgh, 2014. Available: https://ref2014impact.azurewebsites.net/casestudies2/refservice.svc/GetCaseStudyPDF/35271.

[15] S. Cox, J. T. Cox, R. P. Boardman, S. J. Johnston, M. Scott, and N. S. O'Brien, *Iridis-pi: A low-cost, compact demonstration cluster in Cluster Computing*, vol. 17. Berlin, Germany: Springer, 2013. pp 349-–358. ISSN: 1386-7857.

[16] Proxmox Server Solutions GmbH, *Open-source virtualisation management platform Proxmox VE*. Online: Proxmox Server Solutions GmbH, 2020. Available: https://www.proxmox.com/en/proxmox-ve.

[17] Atea Ataroa Limited, *DistroWatch.com: Put the fun back into computing. Use Linux, BSD.* Online: Atea Ataroa Limited, 2020. Available: https://distrowatch.com/.

[18] Manjaro Linux, *Manjaro - enjoy the simplicity*. Online: Manjaro Linux, 2020. Available: https://manjaro.org/.

[19] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK Users' Guide*, vol. 8 of *Other Titles in Applied Mathematics*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1979. ISBN: 978-0-89871-172-1.

[20] GCC Team, *GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF)*. Online: The GNU Project, 2020. Available: https://gcc.gnu.org/.

[21] OpenBSD Foundation, *OpenSSH*. Online: OpenBSD Foundation, 2020. Available: https://www.openssh.com/.

[22] Information Sciences Institute, *Internet Protocol*. Marina del Rey, CA: University of Southern California, 1981. Available: https://tools.ietf.org/pdf/rfc791.pdf.

[23] Internet Engineering Task Force, *Internet Protocol, Version 6 (IPv6) Specification*. Online: Internet Engineering Task Force, 2017. ISSN: 2070-1721.

[24] J. D. Day and H. Zimmermann, *The OSI reference model*, vol. 71 of *Proceedings of the IEEE*. New York, NY: Internet Engineering Task Force, 1983. pp 1334–1340. ISSN: 1558-2256.

[25] Open Virtualisation Alliance, *Virtio - KVM*. Online: Open Virtualisation Alliance, 2020. Available: https://www.linux-kvm.org/page/Virtio.

[26] W. Diffie and M. E. Hellman, *New Directions in Cryptography in IEEE Transactions on Information Theory*, vol. 22. Ronneby, Sweden: IEEE, 1976. pp 644–654. ISSN: 0018-9448.

[27] M. O. Rabin, *Digitalized Signatures and Public-Key Functions as Intractable Factorization in Foundations of Secure Computation*. Orlando, FL: Academic Press, 1978. pp 155–166. ISBN: 978-0-12-210350-6.

[28] R. C. Merkle, *Secrecy, Authentication, and Public Key Systems*. Ann Arbor, MI: UMI Research Press, 1979. ISBN: 978-0-8357-1384-9. Notes: Re-published in 1982.

[29] G. Yuval, *Secrecy, Authentication, and Public Key Systems*. Ann Arbor, MI: UMI Research Press, 1979. ISBN: 978-0-8357-1384-9. Notes: Re-published in 1982.

[30] B. Preneel, *The First 30 Years of Cryptographic Hash Functions and the NIST SHA-3 Competition in Topics in Cryptology – CT-RSA 2010*, vol. 5 985 of *Lecture Notes in Computer Science*. San Francisco, CA: Springer, 2010. pp 1–14. ISBN: 978-3-642-11925-5.

[31] D. Ackley, *A Connectionist Machine for Genetic Hillclimbing*. Norwell, MA: Kluwer Academic Publishers, 1987. ISBN: 978-0-89838-236-5.

[32] C. Vanaret, *Thesis: Hybridization of interval methods and evolutionaryalgorithms for solving difficult optimization problems*. Toulouse, France: Université de Toulouse, 2015. Available: https://arxiv.org/abs/2001.11465.

[33] GASnet Team, *GASnet Communication System*. Online: GASnet, 2020. Available: https://gasnet.lbl.gov/.

Appendices

Appendix α: Example MPI ‘Hello World’ program

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("Hello, World.  I am %d of %d\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

Example Open MPI ‘Hello World’ program written in C [3]

Appendix β: My custom benchmarking software

```
// city3111, main.h, jake deery, 2020
#pragma once
#ifndef MAIN_H
#define MAIN_H

// include – objects
#include ”opClass.h”

#endif





// city3111, main.cpp, jake deery, 2020
#include ”main.h”

int main(int argc, char** argv) {
  // init – setup
  locale systemLocale(”fr_FR.UTF–8”);
  cout.imbue(systemLocale);

  // create opClass object
  opClass * opClassObj = new opClass();

  // call the method
  opClassObj–>doSearch(0.01);

  // delete object
  delete opClassObj;

  return 0;
}
```

```
// opClass.h, jake deery, 2020
#pragma once
#ifndef OPCLASS_H
#define OPCLASS_H

// includes - etc
#include <mpi.h>
#include <cmath>
#include <chrono>

// typedef - float
typedef float float32_t;
typedef double float64_t;
typedef long double float96_t;

// namespaces - std
using std::cout;
using std::flush;
using std::locale;

// namespaces - etc
using uTimeGet = std::chrono::steady_clock;
using uTimeOut = std::chrono::duration<float64_t, std::milli>;

class opClass {
public:
  // public methods
  opClass();
  ~opClass();
  float64_t doSearch(float64_t precision);

  // publics vars

private:
  // private methods
  float64_t getFit(float64_t x, float64_t y);

  // private vars
  int32_t worldSize = 0;
  int32_t worldRank = 0;
};

#endif /* OPCLASS_H */





// opClass.cpp, jake deery, 2020
#include ”opClass.h”

opClass::opClass() {
  // init - mpi
  MPI_Init(NULL, NULL);

  // Get the number of processes
  MPI_Comm_size(MPI_COMM_WORLD, &worldSize);

  // Get the rank of this process
  MPI_Comm_rank(MPI_COMM_WORLD, &worldRank);

  if(worldRank == 0) {
    // init - cout
    cout << ”[I]␣CITY3111␣(Op1)␣program␣by␣Jake␣Deery,␣2020␣.␣.␣.␣.␣” << ”\n”;
    cout << ”[I]␣Initialising␣program␣.␣.␣.␣.␣done” << ”\n”;

    // init - cout
    cout << "[I]␣Initialising␣op␣object␣.␣.␣.␣.␣done" << "\n";
    cout << "\n";
  }
}

opClass::~opClass() {
  // end mpi
  MPI_Finalize();

  if(worldRank == 0) {
    // deleting - cout
    cout << "[I]␣Deleting␣op␣module␣.␣.␣.␣.␣done" << "\n";

    // cout done
    cout << "[E]␣Program␣quitting␣.␣.␣.␣.␣done" << "\n";
  }
}

float64_t opClass::getFit(float64_t x, float64_t y) {
  // vars
  float64_t fit;

  // calculate the fit
  fit = -1.0 * (y + 47.0) * sin(sqrt(abs(y + x / 2.0 + 47.0))) + -1.0 * x * sin(sqrt(abs(x - (y +
     47.0))));

  // end process
  return fit;
}

float64_t opClass::doSearch(float64_t precision) {
  // consts
  const float64_t defaultStartBound = -512.0;
  const float64_t defaultEndBound = 512.0;
  const float64_t defaultBoundRange = 1024.0; // abs(512.0 - (-512.0))
  const float64_t cWorldSize = worldSize - 1.0;
  // vars
  float64_t startBound = 0;
  float64_t endBound = 0;
  float64_t bestFit[3];
  float64_t bestRank = 0.0;
  float64_t effort = 0.0;
  float64_t bunchSize = 0.0;
  uTimeOut deltaTime;

  // defines
  bunchSize = defaultBoundRange / worldSize;

  // begin calculating current bunches -- basic
  startBound = defaultStartBound + (bunchSize * worldRank);
  endBound = startBound + bunchSize;

  // calculating current bunches -- fix for precision to avoid overlap
  if(endBound < 0) endBound += precision;
  else if(endBound == 0 || endBound > 0) endBound -= precision;

  // calculating current bunches -- re-adjust last process to cover entire range
  if(worldRank == cWorldSize) endBound += precision;

  // declaare start of search
  if(worldRank == 0) cout << "\n[I]␣Starting␣op␣search␣.␣.␣.␣.␣" << flush;

  //
  // main for loop

  // getting time
  auto startTime = uTimeGet::now();

  for(float64_t x = startBound; x < endBound + precision; x += precision) {
    for(float64_t y = startBound; y < endBound + precision; y += precision) {
      float64_t fitness = getFit(x,y);

      if(fitness < bestFit[2]) {
        bestFit[0] = x;
        bestFit[1] = y;
        bestFit[2] = fitness;
      };

      // effort counter
      effort++;
    }
  }

  // end and calculate time
  auto endTime = uTimeGet::now();
  deltaTime = endTime - startTime;

  // alert done
  if(worldRank == 0) cout << "done" << "\n";

  //
  // main system echo
  if(worldRank == 0) {


    // for loop for slave messages
    for(uint64_t i = 0; i < (worldSize - 1); i++) { // -1 because we have already stored data from proc#0
      // vars
      float64_t currentFit[3];
      uint64_t currentRank = 0;

      // wait for news
      MPI_Recv(&currentFit[0], 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Recv(&currentFit[1], 1, MPI_DOUBLE, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Recv(&currentFit[2], 1, MPI_DOUBLE, MPI_ANY_SOURCE, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Recv(&currentRank, 1, MPI_INT, MPI_ANY_SOURCE, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```cpp

      // if the received values are the best, keep them
      if(currentFit[2] < bestFit[2]) {
        bestFit[0] = currentFit[0];
        bestFit[1] = currentFit[1];
        bestFit[2] = currentFit[2];
        bestRank = currentRank;
      }
    }

    cout << "[I] Each process ran through approx. " << effort << " loops each" << "\n";
    cout << "[I] The program ran through approx. " << effort * worldSize << " loops in total" << "\n";
    cout << "[I] This took approx. " << deltaTime.count() << " ms per process" << "\n";
    cout << "[I] The best fit was " << bestFit[2] << " at {" << bestFit[0] << ";" << bestFit[1] << "}" << "\n";
    cout << "[I] The best fit was found by process " << bestRank + 1 << " of " << worldSize << "\n";
    cout << "\n";
  } else {
    // send the results
    MPI_Send(&bestFit[0], 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    MPI_Send(&bestFit[1], 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
    MPI_Send(&bestFit[2], 1, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
    MPI_Send(&worldRank, 1, MPI_INT, 0, 3, MPI_COMM_WORLD);
  }

  // end process
  return 0;
}
```

Custom MPI benchmarking software based on the eggholder optimiser function written in C++

Appendix γ: Acquired results from benchmarking software

```
================================================
= 1 OF 5 NODES, 02 PROCESSES, PRECISION = 0,01 =
================================================

= 1 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 5,24308e+09 loops each
[I] The program ran through approx. 1,04863e+10 loops in total
[I] This took approx. 426 463 ms per process
[I] The best fit was -959,6 at {512,01;404,01}
[I] The best fit was found by process 2 of 2

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 2 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 5,24308e+09 loops each
[I] The program ran through approx. 1,04863e+10 loops in total
[I] This took approx. 426 406 ms per process
[I] The best fit was -959,6 at {512,01;404,01}
[I] The best fit was found by process 2 of 2

```

```
[I] Deleting op module . . . done
[E] Program quitting . . . done

= 3 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 5,24308e+09 loops each
[I] The program ran through approx. 1,04863e+10 loops in total
[I] This took approx. 426 399 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 2 of 2

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 4 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 5,24308e+09 loops each
[I] The program ran through approx. 1,04863e+10 loops in total
[I] This took approx. 426 383 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 2 of 2

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 5 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 5,24308e+09 loops each
[I] The program ran through approx. 1,04863e+10 loops in total
[I] This took approx. 426 409 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 2 of 2

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 6 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 5,24308e+09 loops each
[I] The program ran through approx. 1,04863e+10 loops in total
[I] This took approx. 426 412 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 2 of 2

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 7 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done

```

```

[I] Starting op search . . . done
[I] Each process ran through approx. 5,24308e+09 loops each
[I] The program ran through approx. 1,04863e+10 loops in total
[I] This took approx. 426 445 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 2 of 2

[I] Deleting op module . . . done
[E] Program quitting . . . done



================================================
= 2 OF 5 NODES, 04 PROCESSES, PRECISION = 0,01 =
================================================

= 1 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 2,6218e+09 loops each
[I] The program ran through approx. 1,04867e+10 loops in total
[I] This took approx. 219 375 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 4 of 4

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 2 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
```

```
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 2,6218e+09 loops each
[I] The program ran through approx. 1,04867e+10 loops in total
[I] This took approx. 219 402 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 4 of 4

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 3 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 2,6218e+09 loops each
[I] The program ran through approx. 1,04867e+10 loops in total
[I] This took approx. 219 516 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 4 of 4

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 4 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
```

```
[I] Each process ran through approx. 2,6218e+09 loops each
[I] The program ran through approx. 1,04867e+10 loops in total
[I] This took approx. 219 438 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 4 of 4

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 5 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 2,6218e+09 loops each
[I] The program ran through approx. 1,04867e+10 loops in total
[I] This took approx. 219 375 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 4 of 4

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 6 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 2,6218e+09 loops each
[I] The program ran through approx. 1,04867e+10 loops in total
[I] This took approx. 219 418 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
```

```
[I] The best fit was found by process 4 of 4

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 7 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 2,6218e+09 loops each
[I] The program ran through approx. 1,04867e+10 loops in total
[I] This took approx. 219 461 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 4 of 4

[I] Deleting op module . . . done
[E] Program quitting . . . done



=================================================
= 3 OF 5 NODES, 06 PROCESSES, PRECISION = 0,01 =
=================================================

= 1 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,7479e+09 loops each
[I] The program ran through approx. 1,04869e+10 loops in total
```

```
[I] This took approx. 146 156 ms per process
[I] The best fit was −959,583 at {512,003;404}
[I] The best fit was found by process 6 of 6

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 2 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,7479e+09 loops each
[I] The program ran through approx. 1,04869e+10 loops in total
[I] This took approx. 145 812 ms per process
[I] The best fit was −959,583 at {512,003;404}
[I] The best fit was found by process 6 of 6

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 3 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,7479e+09 loops each
[I] The program ran through approx. 1,04869e+10 loops in total
[I] This took approx. 145 958 ms per process
[I] The best fit was −959,583 at {512,003;404}
[I] The best fit was found by process 6 of 6
```

```
[I] Deleting op module . . . done
[E] Program quitting . . . done

= 4 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,7479e+09 loops each
[I] The program ran through approx. 1,04869e+10 loops in total
[I] This took approx. 146 044 ms per process
[I] The best fit was −959,583 at {512,003;404}
[I] The best fit was found by process 6 of 6

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 5 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,7479e+09 loops each
[I] The program ran through approx. 1,04869e+10 loops in total
[I] This took approx. 145 976 ms per process
[I] The best fit was −959,583 at {512,003;404}
[I] The best fit was found by process 6 of 6

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 6 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,7479e+09 loops each
[I] The program ran through approx. 1,04869e+10 loops in total
[I] This took approx. 146 182 ms per process
[I] The best fit was −959,583 at {512,003;404}
[I] The best fit was found by process 6 of 6

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 7 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,7479e+09 loops each
[I] The program ran through approx. 1,04869e+10 loops in total
[I] This took approx. 145 836 ms per process
[I] The best fit was −959,583 at {512,003;404}
[I] The best fit was found by process 6 of 6

[I] Deleting op module . . . done
[E] Program quitting . . . done



==================================================
= 4 OF 5 NODES, 08 PROCESSES, PRECISION = 0,01 =
==================================================

= 1 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,31105e+09 loops each
[I] The program ran through approx. 1,04873e+10 loops in total
[I] This took approx. 109 863 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 8 of 8

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 2 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,31105e+09 loops each
[I] The program ran through approx. 1,04873e+10 loops in total
[I] This took approx. 110 628 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 8 of 8

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 3 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
```

```
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,31105e+09 loops each
[I] The program ran through approx. 1,04873e+10 loops in total
[I] This took approx. 110 264 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 8 of 8

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 4 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,31105e+09 loops each
[I] The program ran through approx. 1,04873e+10 loops in total
[I] This took approx. 109 804 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 8 of 8

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 5 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
```

```
[I] Each process ran through approx. 1,31105e+09 loops each
[I] The program ran through approx. 1,04873e+10 loops in total
[I] This took approx. 110 756 ms per process
[I] The best fit was -959,6 at {512,01;404,01}
[I] The best fit was found by process 8 of 8

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 6 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,31105e+09 loops each
[I] The program ran through approx. 1,04873e+10 loops in total
[I] This took approx. 110 681 ms per process
[I] The best fit was -959,6 at {512,01;404,01}
[I] The best fit was found by process 8 of 8

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 7 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,31105e+09 loops each
[I] The program ran through approx. 1,04873e+10 loops in total
[I] This took approx. 110 617 ms per process
[I] The best fit was -959,6 at {512,01;404,01}
```

```
[I] The best fit was found by process 8 of 8

[I] Deleting op module . . . done
[E] Program quitting . . . done



==================================================
= 5 OF 5 NODES, 10 PROCESSES, PRECISION = 0,01 =
==================================================

= 1 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,0489e+09 loops each
[I] The program ran through approx. 1,04877e+10 loops in total
[I] This took approx. 87 874,8 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 10 of 10

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 2 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,0489e+09 loops each
[I] The program ran through approx. 1,04877e+10 loops in total
```

```
[I] This took approx. 87 892,6 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 10 of 10

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 3 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,0489e+09 loops each
[I] The program ran through approx. 1,04877e+10 loops in total
[I] This took approx. 87 905,5 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 10 of 10

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 4 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,0489e+09 loops each
[I] The program ran through approx. 1,04877e+10 loops in total
[I] This took approx. 88 008,3 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 10 of 10

```

```
[I] Deleting op module . . . done
[E] Program quitting . . . done

= 5 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,0489e+09 loops each
[I] The program ran through approx. 1,04877e+10 loops in total
[I] This took approx. 88 038,5 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 10 of 10

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 6 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,0489e+09 loops each
[I] The program ran through approx. 1,04877e+10 loops in total
[I] This took approx. 87 895,7 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 10 of 10

[I] Deleting op module . . . done
[E] Program quitting . . . done

= 7 =
[I] CITY3111 (Op1) program by Jake Deery, 2020 . . .
[I] Initialising program . . . done
[I] Initialising op object . . . done


[I] Starting op search . . . done
[I] Each process ran through approx. 1,0489e+09 loops each
[I] The program ran through approx. 1,04877e+10 loops in total
[I] This took approx. 87 961,4 ms per process
[I] The best fit was −959,6 at {512,01;404,01}
[I] The best fit was found by process 10 of 10

[I] Deleting op module . . . done
[E] Program quitting . . . done
```

Unedited results generated from my final prototype custom MPI benchmarking software