Acknowledgments

First and foremost, I would like to express my gratitude to the promotor, Prof. dr. Wim Lamotte, who suggested this thesis topic based on my interest in single-board computers. Without his recommendation, I would not have been able to delve into the world of parallelism, clustering and single-board computers. My special thanks also goes to Prof. dr. Peter Quax, the co-promotor, who designed and constructed the physical setup of the cluster. I have to again thank both for their time spent by supporting and correcting me during the whole duration of the thesis. Also, a well-meant thank you for the investment in the hardware, making this thesis possible.

I would also like to thank Prof. dr. Niel Hens and dr. Steven Abrams for their interest, time and willingness to provide their code and assistance for the statistics use case. The same also goes to dr. Jori Liesenborgs, who provided and installed his programs several times for the physics use case.

Finally, I would like to express how grateful I am to my fellow students, my parents and my girlfriend, Laura, for their support over the past few years. Thank you!

Abstract

In the field of high performance computing (HPC), systems such as supercomputers and computer clusters are used to solve computationally large problems. However, barriers have been informally identified regarding the usage of HPC systems. Not every potential user can gain access to a supercomputer nor can it be quickly bought since such a machine comes at a large total cost of ownership. Existing systems are shared among multiple users, leading to problems. Additionally, they are usually remotely accessed, raising privacy and security concerns. Finally, the usage of the system itself can be a threshold since its usage is more complicated than a machine typically owned by a user.

In order to solve some of the problems in HPC, a computer cluster consisting of low-cost single-board computers (SBCs) was developed. 64 Odroid-C1+ SBCs are provided as compute nodes. The cluster should be a private, single user system that is able to bridge the gap between user and HPC systems. Therefore, this thesis is a feasibility study, exploring if it still scales well and delivers adequate performance despite its cost compared to other systems. Furthermore, attention should be paid to general usability and ease-of-use. This thesis also takes the form of a case study, as the cluster is evaluated and compared to other systems through real-world problem cases of statistics and physics.

Through both use cases, the performance of the cluster was assessed. The cluster was found to scale equivalently to a full-fledged cluster for programs that have a large parallel fraction. The performance of the parallel regions is very reasonable considering the cost of the cluster, although sequential code severely diminishes the advantages gained through parallelism. Several methods for user friendliness were deployed and were informally confirmed to reduce the usage threshold. Therefore, we believe that the cluster is a successful attempt at bringing the power of parallelism to the general public. It does not replace existing HPC machines, rather, it bridges the gap between user systems and full-fledged clusters. It is complementary to both, since for large sequential parts, the user's own machines are the better choice. For raw performance, an existing HPC machine should still be used.

Dutch Summary

Introductie en Onderzoek

Binnen high performance computing (HPC) worden er systemen zoals supercomputers en computerclusters gebruikt om computationeel intensieve problemen op te lossen. Zulke systemen zijn uiteraard vele malen performanter en complexer dan de gemiddelde machines van een eindgebruiker. Informeel werden er enkele problemen blootgelegd met betrekking tot het gebruik van HPC systemen. Eerst en vooral heeft niet iedereen zomaar toegang tot HPC systemen, wat kan leiden tot het gebruik van notebooks en desktop PC's, wat verre van optimaal is. Bestaande systemen worden bovendien vaak gedeeld met verscheidene gebruikers of groepen, wat kan zorgen voor problemen bij het eerlijk verdelen van het gebruik ervan. Verder zijn HPC systemen veelal enkel bereikbaar op afstand, aangezien zelf een supercomputer bezitten erg kostelijk kan zijn. Privacy en beveiligingsproblemen kunnen hierbij een rol spelen, aangezien gebruikers weinig harde garanties kunnen hebben op het beschermen van hun data. Ten slotte kan het gebruik van zo'n machine zelf een belemmerende factor zijn, aangezien de systemen complexer zijn dan de machines van de gebruiker.

Deze thesis poogt enkele van de huidige problemen bij HPC aan te pakken door een systeem voor HPC te ontwikkelen dat privé is en voor een enkele gebruiker bestemd is, als een alternatief voor bestaande HPC systemen. Er wordt een computercluster ontwikkeld, waarvan de prijs laag wordt gehouden door gebruik te maken van goedkope componenten: single-board computers (SBC's). Naarmate de prijs van de cluster laag is, verwachten we niet dat de cluster bestaande HPC systemen zal vervangen. Het is eerder de bedoeling dat de cluster een plaats inneemt tussen de systemen van de eindgebruiker en HPC systemen. De thesis neemt bijgevolg de vorm aan van een haalbaarheidsstudie, waarin wordt gekeken of een goedkope SBC cluster effectief gemaakt kan worden met een lage kostprijs en die flexibele code toelaat. Verder zou de cluster nog steeds goed moeten schalen, aanvaardbare performance moeten leveren en concrete problemen moeten aankunnen. De cluster moet tevens eenvoudig zijn in gebruik, zowel voor het interpreteren van resultaten als voor het aansturen van specifieke problemen en voor algemeen gebruik. De thesis neemt bovendien de vorm aan van een gevallenstudie. De opgesomde vereisten worden getoetst aan de hand van twee concrete problemen uit de statistiek en fysica.

Cluster Architectuur

De belangrijkste componenten van de cluster zijn uiteraard de *compute nodes*, de computers die effectief de berekeningen van gebruikers uitvoeren. Hiervoor werden er 64 SBC's voorzien. De SBC die gekozen werd, is niet de populaire Raspberry Pi 2B, omdat het onder andere te kort schoot op vlak van netwerksnelheid (10/100 Mbps, gigabit is gewenst). Uit de andere bordjes die beschikbaar zijn op de markt (augustus 2015) werd de Odroid-C1+ gekozen, een goedkope SBC die voldoet aan de door ons opgestelde eisen voor dezelfde prijs als de Pi 2B.

Op de bordjes werd er MPI voorzien, wat toelaat programma's uit te voeren in een gedistribueerde omgeving. Om gebruik te maken van de compute nodes, werd er een *login node* voorzien in de vorm van een eenvoudige laptop, waarop gebruikers kunnen inloggen. Op deze laptop bevindt zich onder meer een *resource manager* en een *scheduler*, zodat gebruikers hun jobs kunnen laten uitvoeren op de cluster. Bovendien kan de laptop gebruikt worden voor monitoring zodat gebruikers een beeld kunnen krijgen hoe de cluster daadwerkelijk wordt gebruikt voor hun programma's. De cluster werd ook zodanig opgesteld dat deze qua gebruik aansluit bij de cluster van het Vlaamse Supercomputer Centrum (VSC), aangezien gebruikers aan de UHasselt hier reeds ervaring mee kunnen hebben of om de overgang te vergemakkelijken.

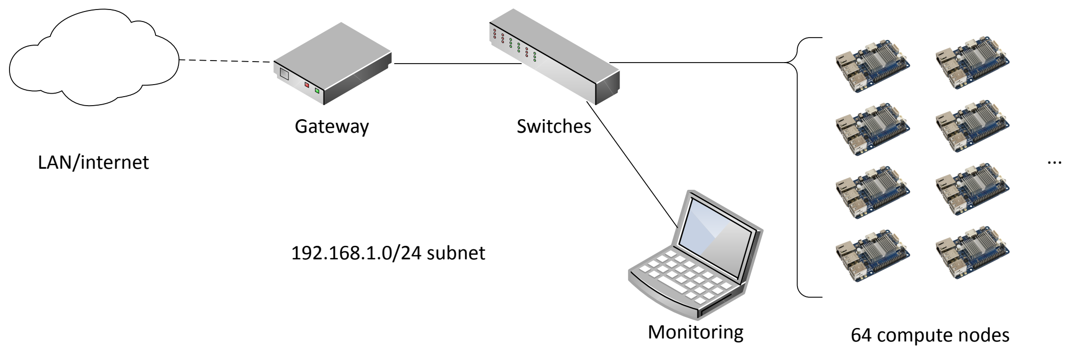


FIGURE 1: Schematische opstelling van de componenten

Evaluatie

Om de performantie van de cluster te evalueren, wordt er gebruik gemaakt van twee use cases. Een eerste case uit de statistiek schatte de kans in op een uitbraak van mazelen binnen België. De code werd door de onderzoekers uitgevoerd op hun eigen machines, met lange uitvoeringstijden tot gevolg. De aangeleverde code geschreven in R was initeel nog sequentieel, waardoor de code voor uitvoering op een cluster eerst nog geparallelliseerd diende te worden. Uit dit proces blijkt dat voor parallelle berekeningen in R, de meest aangewezen methode niet altijd de meest geschikte methode voor performantie is. Om een beeld te krijgen van de performantie, wordt de SBC cluster vergeleken met een systeem dat een systeem van de eindgebruikers benadert, namelijk een PC met een CPU van 4 cores.

De case leerde ons dat de code beperkt schaalt, tot 28 van de 64 bordjes. Het gebruik van meer dan 28 bordjes zorgt voor een langere uitvoeringstijd, wat zuiver te wijten valt aan de manier waarop R is geïmplementeerd. De performantie op zich is een enorme vooruitgang ten opzichte van een desktop PC. Enkel het preprocessing gedeelte, bestaande uit een sequentieel stuk en multithreaded gedeelte, is relatief minder interessant om uit te voeren op de cluster. In feite wordt er dan de vergelijking gemaakt tussen een enkel bordje en de PC, wat niet heel eerlijk is. Het nadeel is dat lange sequentiële stukken de parallelle fractie kunnen onderdrukken, wat best actief vermeden wordt.

Buiten de performantie op zich, werd er ook gekeken naar de gebruiksvriendelijkheid van de cluster. Voor dit specifieke probleem werden er voor zowel in- als uitvoer interfaces voorzien met het verlagen van de gebruiksdrempel als doel. Op deze manier kunnen gebruikers anders dan de ontwikkelaars zelf voor dit probleem de cluster gebruiken. Ze hebben hierbij geen technische kennis van de cluster nodig, wat niet meteen of geen optie is bij grotere, niet-privé clusters. Uiteraard zijn zulke interfaces enkel nuttig bij relatief statische code die herhaaldelijk met andere parameters moet uitgevoerd worden.

Bij een tweede use case uit de fysica werd er het gravitationele lenseffect gesimuleerd door middel van een ray tracing algoritme. Dit maakt dat het probleem embarassingly parallel is, aangezien elk werkerproces een verzameling pixels kan verwerken onafhankelijk van de anderen. Het programma heeft bovendien geen preprocessing stap zoals bij de vorige case. Er wordt een vergelijking gemaakt van de cluster met een reeds bestaande cluster van het VSC, ThinKing. Uit de resultaten blijkt dat de SBC cluster ongeveer een factor 4.6 trager is, wat erg meevalt gezien de kostprijs. Ook schaalt de cluster analoog voor oplopende processoraantallen voor dit probleem.

Conclusie

In deze thesis werd er aangetoond dat een cluster van SBC's gemaakt kan worden voor HPC doeleinden, gelijkaardig aan een 'echte' cluster zoals die van het VSC. Bovendien is de cluster met zijn kostprijs erg goedkoop vergeleken met de voorgenoemde cluster. De flexibiliteit van de code werd aanngetoond door de use cases. De code diende niet aangepast te worden specifiek voor de SBC cluster. Voor problemen met een grote parallelle fractie, biedt de cluster zeker een meerwaarde ten opzichte van standaard laptops of desktop systemen. Sequentiële delen zorgen er echter voor dat deze meerwaarde kan vervallen. Verder werd er door beide cases aangetoond dat de cluster effectief met realistische problemen overweg kan. Voor de gebruiksvriendelijkheid werden er tevens verschillende opties aangereikt die informeel werden bevestigd.

Waar kunnen we de SBC cluster uiteindelijk situeren? De cluster is een succesvolle poging om parallellisme op een goedkope manier naar een breder publiek te brengen.

De cluster slaagt er in systemen van de eindgebruiker en volledige clusters zoals die van het VSC te overbruggen. We kunnen de cluster dus tussen beiden situeren. Het systeem is complementair aan beiden omdat het niet de high-end HPC systemen gaat kunnen vervangen, waar absolute performantie gehaald wordt tegen een uiteraard veel hogere kostprijs. Standaard desktop machines kunnen dan weer gebruikt worden om lange sequentiële delen die teleurstellend zijn qua uitvoeringstijd op te vangen, waarna het parallelle gedeelte op de cluster kan worden uitgevoerd. Verder toonde het probleem uit de statistiek aan dat de cluster ook een ideale hulp kan zijn bij het parallellisatieproces, wat minder snel mogelijk is bij een gedeelde cluster.

Contents

Acknowledgments **I**

Abstract **II**

Dutch Summary **III**

1 Introduction **1**

1.1 Problem Statement . . . 1
1.2 Research Questions . . . 2
1.3 Thesis Outline . . . 3

2 High Performance Computing **5**

2.1 High Performance- and Throughput Computing . . . 5
2.2 Problem Types and Applications . . . 5
2.2.1 Example: N-Body Problems . . . 6
2.2.2 Example: Molecular Dynamics . . . 8
2.2.3 Example: Problems with a Probabilistic Interpretation . . . 9
2.2.4 Example: Sequence Alignment . . . 11
2.3 HPC Systems . . . 12
2.3.1 Supercomputer . . . 13
2.3.2 Cluster . . . 14
2.3.3 Grid . . . 15
2.3.4 Access Methods . . . 15
2.4 Conclusion . . . 16

3 Parallelism Within a Cluster **17**

3.1 A Single Multi-Core Node: Shared Memory . . . 17
3.2 Multiple Nodes: Distributed Memory . . . 19
3.3 Hybrid Approaches . . . 20
3.4 Other Strategies . . . 21
3.4.1 GPU Programming . . . 21
3.4.2 Coprocessor . . . 22
3.5 Utilizing the Cluster Resources . . . 22
3.6 Conclusion . . . 23

4 SBC Cluster Architecture **24**

4.1 Processor Architecture . . . 24

4.2 Single-Board Computer Clusters in Literature . . . 26
4.3 Choosing the Hardware . . . 27
4.4 Compute Node Setup . . . 30
4.4.1 Operating System . . . 30
4.4.2 RAM Allocation . . . 30
4.4.3 Specific Hardware Configuration . . . 31
4.5 Network Setup . . . 31
4.6 Software Setup . . . 33
4.6.1 Resource Management and Scheduling . . . 33
4.6.2 Message Passing . . . 35
4.6.3 Cluster Synchronization . . . 35
4.6.4 Network File System . . . 36
4.7 Monitoring . . . 36
4.7.1 Monitoring Host . . . 36
4.7.2 Monitoring Tools . . . 37
4.8 The Cluster Front-End . . . 38
4.8.1 Basic Job Submission and Compilation . . . 38
4.8.2 Web-Based Job Submission . . . 39
4.8.3 Monitoring . . . 40
4.8.4 A Multi-User System . . . 41
4.9 Conclusion . . . 42

5 UC 1: Measles Resurgence in a Highly Vaccinated Population 43
5.1 Introduction . . . 43
5.2 Original Code . . . 44
5.3 Parallel Computations in R . . . 46
5.3.1 Packages and Options . . . 46
5.3.2 Basic snow Functionality . . . 46
5.4 Parallelization of the Code . . . 47
5.4.1 Method . . . 47
5.4.2 Parallel Bootstrap Implementation . . . 48
5.4.3 GAM Fitting Parallelization . . . 49
5.4.4 Proposed Scheme . . . 49
5.5 Parallel Implementation Remarks . . . 50
5.5.1 Running the Program . . . 50
5.5.2 The clusterExport Function . . . 51
5.5.3 Load Balancing and Static Scheduling . . . 52
5.5.4 File I/O . . . 52
5.6 Performance . . . 53
5.7 User Friendliness . . . 58
5.8 Conclusion and Lessons Learned . . . 60

6 UC 2: Gravitational Lenses 62
6.1 Introduction . . . 62
6.2 Cases of Interest . . . 62
6.2.1 Gravitational Lens Inversion . . . 63

6.2.2 Gravitational Lens Simulation 64
6.3 Performance 64
6.4 Conclusion 68

7 Conclusions and Future Work **69**

Bibliography **73**

Chapter 1

Introduction

1.1 Problem Statement

From time to time, it is possible that a sequential computer's computing speed is insufficient to solve computationally heavy problems within reasonable time. At this point, programs designed to compute them will require more than just a sequential approach. Additionally, such problems have over the years grown in number and size.

According to Pfister [1, 2], computing performance can be improved in three ways:

1. Work harder;
2. Work smarter;
3. Get help.

Here working harder implies the usage of better hardware and working smarter stands for general efficiency and better algorithms. Yet in the end, help is needed through the use of multiple systems or processing units instead of just a single, sequential system. These multiple systems lead us to parallelism, or the ability to do computations in parallel. Parallelism on a large scale brings us to the field of high performance computing (HPC).

High performance computing systems such as supercomputers, computer clusters and grids provide the necessary resources to solve computationally large problems. In order to keep up with the ever-increasing problem size, HPC systems too, have grown in number and size. Their speed has increased dramatically: from gigaFLOPS in the early 1990s to several petaFLOPS today. This improvement was driven mainly by the increasing demands from users within a scientific, engineering or industrial context [3].

Together with the growth of HPC systems, the amount of HPC users has increased. However, their number is limited to just a tiny fraction of all computer users [3]. Moreover, barriers still exist for current or potential users regarding the usage of HPC systems. Several key problems were identified.

Firstly, some users simply do not have access to a supercomputer. Projects usually originate from a scientific or industrial context. This however does not mean that every university or company has the means or permission to access a supercomputer. Alternatives such as common desktop machines or even notebooks are used, which is far from ideal. Research is slowed down due to large waiting times when executing intensive programs on these machines.

Secondly, existing supercomputers are usually not intended for a sole user. Instead, thus in a traditional multi-user environment, processes or jobs of a single user can be affected by other users [4]. In case of a distributed environment such as a computer cluster, this would mean that not every compute node is available for a single user at a certain point in time. The interconnect network can suffer as well when under heavy load. Other problems, such as the misuse of shared resources on a single node, e.g. the “login node” where users typically log in to start their jobs, can lead to frustration. In addition to these problems, systems with a large amount of users can suffer from long queue times as user jobs are placed in a queue, scheduled for execution.

Another problem worth mentioning is that supercomputers are usually placed at a remote location from the user perspective. Users access the remote HPC machine through a connection with their own machines. At that location, there is no real guarantee that adequate security measures are taken in order to protect confidential user data. Transferring large amounts of data from and to the remote system may also be impractical. In order to solve these problems, one could opt to place an HPC machine in-house. Such a task is no easy feat as such systems are expensive, take up a lot of space and require maintenance resulting in a large total cost of ownership.

A final barrier can be the usage of the system itself. Using an HPC system efficiently and effectively can be hard and even more so for people with a background different from IT. They can be discouraged from using HPC systems as certain aspects, e.g. starting or managing jobs and gathering the results, can be rather complex compared to the same on their own systems. Furthermore, optimizing or testing code can be hard, as it cannot be quickly run on the system due to the aforementioned queue times.

1.2 Research Questions

Considering the current problems in HPC, this thesis attempts to overcome some of them. To do so, a system for HPC purposes will be developed that can provide a similar but private, in-house alternative for remote HPC systems that is intended for a single user. It should take the form of a computer cluster composed of low-cost commodity hardware while still providing a cost-effective solution. As such, it is not our intention to replace existing HPC machines. Rather, the cluster should be situated in-between existing HPC systems and the user's own notebook or desktop machine in order to bridge the gap between both worlds.

With the low-cost aspect of the cluster in mind, it is of interest to use inexpensive single-board computers (SBCs) as components for the cluster. A well-know SBC is the Raspberry Pi, originally released in February 2012, which revolutionized the interest in SBCs for the general public. Over the following years, many other commercial boards were developed. Currently, in the year 2015, is it a possibility to effectively use more mature SBCs for HPC purposes? The following research questions (RQ) can be distilled:

RQ 1. Is it possible to develop a computer cluster that

- (a) consists of low-end single-board computers;
- (b) remains inexpensive;
- (c) maintains code portability, that is, code does not need to be drastically changed in order to run on the cluster.

RQ 2. If so, when compared to existing systems, does the SBC cluster

- (a) scale well;
- (b) deliver *reasonable* performance;
- (c) have the ability to tackle diverse real-world problems.

RQ 3. Can the cluster guarantee user friendliness for users even with a background different from IT, regarding

- (a) viewing the cluster status and interpreting results;
- (b) general usability and ease-of-use;
- (c) the ability to support usability for domain-specific problems.

This thesis is both a **feasibility study** and a **case study**. The feasibility of the cluster is evaluated through real-world problem cases from different domains, i.e. statistics and physics. This way, it is possible to gain an understanding of the performance of the cluster and its feasibility in general when compared to other (HPC) machines. Moreover, to improve user friendliness of the cluster, contact with HPC users is actively pursued to allow adequate measures to be taken.

1.3 Thesis Outline

First, a broad understanding of the HPC field must be acquired. What kind of real-life problems are situated in the HPC field? What machines are deployed to solve them? These questions among others will be answered in Chapter 2. Since we are interested in computer clustering, now properly situated in the HPC field, Chapter 3 will narrow down to clusters. Parallelism concepts are introduced and it is discussed how parallelism is gained through different strategies within clusters.

The gained knowledge of these chapters is then applied in the following chapters. Chapter 4 discusses the system architecture. Of course, a suitable SBC candidate has to be selected for the cluster first. The setup of both hardware and software components are discussed and evaluated. Furthermore, general user friendliness measures for the cluster are described.

Chapters 5 and 6 encompass two use cases of real-world problems. After gaining an understanding of the problems, the performance and feasibility of the SBC cluster is assessed. The results are compared with the results of systems that are currently used for each respective case. Moreover, for use case 1, the existing code itself was not yet suited for execution on a cluster. All parallelization efforts and considerations are described as well. Finally, through the use cases and the rest of this thesis, conclusions are drawn and ideas for future work are given.

Chapter 2

High Performance Computing

As mentioned in the Introduction, many computational heavy problems exist today and are solved by several types of HPC machines. In this chapter we will take a closer look at the high performance computing (HPC) paradigm. What kind of problems exist and how are they computed, by what kind of machines?

2.1 High Performance- and Throughput Computing

For high performance computing, pure computing performance is of importance. Floating point operations per second (FLOPS) is the primary metric here to measure the performance [5, 6]. Other metrics include, for example, tasks per second, IO rates, and so on. For a scientist, these metrics are a strong indicator of computational throughput, benefiting his computational-heavy research.

High throughput computing on the other hand is a computing paradigm quite similar to high performance computing. The two paradigms should not be confused, however. Instead of using metrics such as FLOPS, other metrics are used: the number of operations or jobs per minute, day, month or even year. High efficiency is not a direct priority, as HTC is more interested in how many jobs can be completed over a long period of time. In order to do this, the maximum amount of computing resources has to be made available to the users. This being HTC's main challenge, it differs from HPC, where the speed of every single job completion counts [5].

2.2 Problem Types and Applications

The HPC problem set is rich and spans over many industries and areas of research. Over the past two decades, the commercial use of HPC has been growing rapidly [7].

This growth can be attributed to somewhat more traditional sectors, such as the automotive and aerospace sectors. Other sectors have a role as well, e.g. the finances and biotechnology sectors.

Looking at the types of problems that are present in these and other domains, then modeling and simulations form a first and important category in the HPC problem space. Simulations and models are used when it is not feasible or possible to do physical experiments, as certain problems can simply be too large or small in size, or can take too long or short in time. For example, HPC enables the simulation of car crashes in order to reduce expensive physical testing. Another example is the improvement in noise coming from an aircraft. Through extensive acoustic simulations, the noise can be reduced. Also, in a medical context, it is possible to study the effects of a new drug through simulations before moving to patient trials. Other examples include the calculation of atmosphere-ocean climate models [8] or weather forecasting in general.

Besides simulations, more 'direct' calculations, not part of a simulation, are a possibility as well. Here, we can situate the rendering of modern CGI films. Since each frame takes many hours to render, utilizing HPC resources is a must. In the financial world, HPC can aid when performing highly complex calculations to estimate the value of assets and to assess risks.

HPC also acts as an enabling technology in areas that require the processing of large-scale data. An example here can be the processing of data that is generated by satellites and telescopes. The amount of observations can be enormous, resulting in a lot of data which must be analyzed. This also enables the calculation of orbits of objects in space, allowing the prediction of events such as collisions in the near future.

Finally, we will take a closer look at a few HPC problems that were not yet mentioned, that is, N-body problems, molecular dynamics, probabilistic problems and sequence alignment [9]. The following sections will discuss them and show why they are indeed intensive problems and where and how parallelization can be used in order to solve them. In addition to these examples, two problems, both being simulations albeit in different research domains, will later be examined in-depth as use cases in chapters 5 and 6.

2.2.1 Example: N-Body Problems

N-body problems are problems that deal with e.g. the gravitational or electrical forces exercised by a finite amount of particles on each other. These N particles can vary in size, from very small particles such as atoms to planets, stars and so on. This problem class is typically situated among others in the domain of astrophysics. For example, our solar system can be modeled as an N-body problem, with each particle representing a planet or other object. Another example is the dynamics of globular cluster star systems, a problem which emerged in the 20^{th} century [10].

The algorithm to solve this class of problems is rather simple:

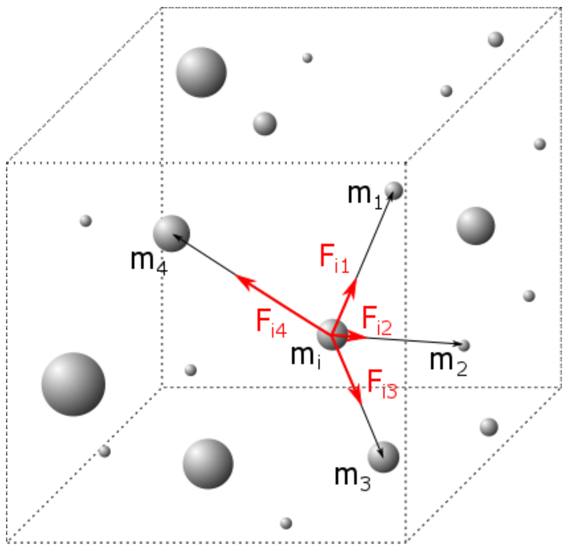


FIGURE 2.1: The N-body problem: calculating the resulting force on a particle m_i given the positions of the other particles in a closed system

1. Given the positions of all particles, calculate the force on every particle;
2. Throughout a defined time frame, move the particles according to the constant force calculated in step 1.
3. Repeat these steps if necessary as defined by the simulation time.

Picking smaller time frames will result in better approximations of the real-world behavior. Note that this algorithm only applies to a closed system, as no external forces are incorporated and thus ignored. Collisions are ignored as well in order to avoid nearest-neighbor interactions.

Simplicity is the main benefit of the described algorithm. It can even be easily implemented in parallel. Unfortunately, its computational complexity makes this algorithm not favorable, having $O(N^2)$ operations for N particles. A better approach is making use of the Barnes-Hut algorithm, a popular method for solving N-body problems. Again, this algorithm is meant for collisionless particles and used when close encounters among pairs of particles are not of real importance. The algorithm consists of the following steps:

1. Divide the 3D closed system into cells, in a recursive fashion where every cell is to contain a single particle at most. Organize them in an octree.
2. Compute the total mass and center of the mass for every cell on every level of the tree.
3. Use the levels of cells to calculate the interactions between particles: for a particle p, for every top level cell c, use the mass and center of c if c is far enough from p. Otherwise, use the children of c.

Thus the number of computations is reduced by merging particles into cells, resulting in a complexity of $O(NLog(N))$ if sequentially executed. Implementation-wise, octrees are not trivial for high performance systems. The problem is irregular as the particles can be seemingly randomly distributed in space and it dynamically changes its irregularity since every particle can move in any direction. This results in unequal sizes of subtrees and unequal computation times.

On a single machine, that is in shared memory as further discussed in Chapter 3, every particle can be seen as a task and is easy to implement for multiple processing units. However, the time to complete a task can vary because the processing time differs from subtree to subtree. The number of particles or tasks, N, is probably much higher than the available processing units, which makes it even out itself: a finished processing unit can simply start processing the next subtree instead of remaining idle.

Tackling this problem with multiple machines at once is harder. Every particle or task could potentially need access to any part of the entire octree. The Barnes-Hut algorithm can be adjusted slightly in order to solve this, which is of less importance here. What is important is that load balancing is absolutely necessary, as opposed to the shared memory version. Processing units of the distributed machines that would process an arbitrary task would be left idle otherwise; explicit communication between them is required.

2.2.2 Example: Molecular Dynamics

A molecular dynamics simulation is a technique for gaining information on the behavior of molecules. In the simulation, the molecular system's behavior is calculated atom by atom for a given period of simulation time. The motions resulting from the calculated forces between the molecules provide information and properties of the dynamic system in general.

In the fields of biophysics and structural biology, molecular dynamics simulations are an important asset and are frequently used to simulate biological molecules such as nucleic acid or proteins. In addition to biological molecules, they can also be used for synthetic molecules or nanotechnological devices that possibly have not or cannot be created yet. In this case, the main applicable fields are nanotechnology and materials science.

Molecular dynamics is essentially a variation of the N-body problem class. Already mentioned were the particles present in N-body problems (molecules and atoms) and the time frame for every computation step which is the total simulation length split into tiny parts. A set of molecules acts with respect to Newton's second law of motion $F = ma$, with F the force exerted, m and a the mass and acceleration of a particle respectively. Molecular dynamics can be calculated analogous to the simple algorithm for N-body problems, described in Section 2.2.1.

However, computing the forces on every particle is expensive. The period of interest, or the simulation length, consists of an immense amount of small time steps. The simulation has to take the available computational power into account. In order to let the computation finish within reasonable time, trade-offs between the number of particles, time step size and simulation time have to be made. A million molecules can be simulated for a long period of time. On the contrary, the simulation time should probably be small for billions of molecules. And yet, the simulation has to be at least long enough to provide adequate information about the system that is being studied. Picking a simulation time that is too short is described with an analogy [11] as

> *"...making conclusions about how a human walks when only looking at less than one footstep."*

Describing how molecular dynamics can be parallelized falls out of this thesis's scope as it is more complex than e.g. just the octree method described in Section 2.2.1. What is important is that every particle in theory influences all other particles. If each particle were mapped on a single processing unit, communication between processing units would become a serious bottleneck. Combined with the huge number of simulation steps, the computation would take a long time. More advanced algorithms have been developed in order to reduce the communication overhead, but molecular dynamics remains a challenging problem even for today's high-end machines.

2.2.3 Example: Problems with a Probabilistic Interpretation

Monte Carlo methods are a broad class of algorithms that use repeated random sampling to solve problems, instead of computing the exact solutions. Since the result is approximated, some uncertainty will be present. On average however, we will get the right solution thanks to the 'law of large numbers' in statistics. Uncertainty of the result decreases when the number of samples increases. In theory, Monte Carlo methods can be used to solve any problem with a probabilistic interpretation. Some problem examples are statistical in nature e.g. Brownian motion which deals with particle movement in a fluid or radioactive decay. However, Monte Carlo methods can cover more than just statistical problems. In computer science, ray tracing is an example in the computer graphics area but it can be used for AI tactics in games as well. In physics, it can play a role in e.g. weather forecasting and galaxy evolution modeling.

Generally, Monte Carlo methods may vary, but their structure tends to be more or less the same [12]:

1. Define a domain of inputs;
2. Generate random inputs using a probability distribution function for the domain;
3. Do a computation on the inputs;
4. Aggregate the results.

The following example may shed more light on the idea of Monte Carlo methods [13]. To solve integral I,

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

function f is evaluated at random points in the domain $[a, b]$, with the probability distribution function $p(x_i)$,

$$I \approx I_{estimator} = \frac{f(x_i)}{p(x_i)},$$

where the expected value of $I_{estimator}$ equals,

$$E[I_{estimator}] = I = \int_a^b \frac{f(x_i)}{p(x_i)} \cdot p(x_i)\ dx_i.$$

This process can be repeated by taking the average value of N estimators,

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

Let us illustrate the above concepts with a concrete example. Suppose we want to calculate the sum of all numbers $S = s_1 + s_2 + \ldots + s_n$ between 1 and 500 using Monte Carlo integration. When picking random numbers, we know that each number has the same probability of being picked. Step by step, according to the general Monte Carlo structure:

1. The domain of inputs is $[1, 500]$.
2. The probability distribution function is $p(i) = \frac{1}{500}$, as every number has the same odds of being picked. Five randomly generated samples are 82, 452, 124, 283 and 174.
3. Compute and aggregate:

$$\begin{aligned} I_{estimator} &= \frac{1}{5} \cdot \frac{82 + 452 + 124 + 283 + 174}{1/500} \\ &= 111500 \\ &\approx 125250 \end{aligned}$$

This example is of course too easy for Monte Carlo methods to be used. In this case, the sum from 1 to n can simply be calculated by the formula $\frac{n \cdot (n+1)}{2}$. Furthermore, the approximated solution is strictly speaking incorrect, but can be improved by increasing the number of samples.

Monte Carlo methods can be easily parallelized because every function evaluation can be done independently in parallel. Although this may seem easy, in practice a problem rises. Generating truly random numbers in parallel is not trivial. Pseudo random numbers are generated using system environment variables and noise. On Unix-like operating systems, `/dev/random` can provide such numbers. In parallel however, multiple processes on the same machine using the same generator can result in too similar or identical 'random' values. The user should be aware of this problem and his code must actively avoid this situation. A parallel random number generator has to be deployed, which can be done in various ways.

2.2.4 Example: Sequence Alignment

In bioinformatics, sequence alignment is used to arrange the sequences of DNA, RNA or proteins to identify regions of similarity between sequences. This yields information about conserved regions within different species. Finding relations between species or a common ancestor in the evolution of these species is then possible.

Sequences can be limited in length or be similar to each other, making it possible to align them by hand. However, this is usually not done in practice as sequences are long and complex. Many algorithms have been developed over the years; dynamic programming being a straightforward, formally correct method. However, this approach is expensive, requiring a lot of memory and computing time. Other algorithms using heuristics or probabilistic methods have been designed, both not providing the most optimal match but performing more efficiently.

One of the primary research areas in bioinformatics is Multiple Sequence Alignment (MSA) [14], the alignment of three or more sequences at a time. Pairwise alignment is the base case for just two sequences. For n individual sequences, aligning them naively pairwise causes the search space to increase exponentially. Finding the perfect match for n sequences is extremely difficult and has been shown to be NP-complete [15]. Figure 2.2 is an example of interspecies MSA. Here, a protein has been aligned for several species: a rainbow trout, rat, sheep, pig, common squirrel monkey and human. The stars at the bottom of the figure indicate a match between the listed proteins.

```
sp|P49843|GCR_ONCMY  YGVLTCGSCKVFFKRAVEGWRARQNTDGQHNYLCAGRNDCIIDKIRRKNCPACRFRKCLQ
sp|P06536|GCR_RAT    YGVLTCGSCKVFFKRAVE---------GQHNYLCAGRNDCIIDKIRRKNCPACRYRKCLQ
sp|P35547|GCR_SHEEP  YGVLTCGSCKVFFKRAVE---------GQH------------------------------
sp|Q9N1U3|GCR_PIG    YGVLTCGSCKVFFKRAVE---------GQHNYLCAGRNDCIIDKIRRKNCPACRYRKCLQ
sp|O46567|GCR_SAISC  YGVLTCGSCKVFFKRAVE---------GQHNYLCAGRNDCIIDKIRRKNCPACRYRKCLQ
sp|P04150|GCR_HUMAN  YGVLTCGSCKVFFKRAVE---------GQHNYLCAGRNDCIIDKIRRKNCPACRYRKCLQ
                     ******************         ***
```

Figure 2.2: Illustration of interspecies MSA of a protein for respectively rainbow trout, rat, sheep, pig, common squirrel monkey and human. Generated by Clustal Omega [16]

Due to its NP-completeness, using a heuristic algorithm to tackle this problem is essential. An example is progressive alignment, which builds up a final MSA by combining pairwise alignments, starting with the most similar pair and progressing to the most distantly related. While more efficient than dynamic programming, this approach is still computationally expensive. Progressive alignment together with parallelization is the key for acceptable computation times, since the many pairwise alignments can be done independently in parallel [17]. ClustalW [18] especially uses this approach. Clustal in general is a series of software specially targeted towards MSA. Different versions have been developed for several parallel approaches, targeting shared memory multiprocessors and distributed memory clusters; both types are further discussed in Chapter 3. Clustal Omega [16] is another version of this software and was used to create Figure 2.2.

2.3 HPC Systems

In order to tackle HPC problems, powerful computers are needed. Increasing the speed of the computer's processing units would be the most straightforward solution for the machine to become faster. Indeed, according to Moore's law, the number of transistors on a chip doubles every two years, leading to an exponential growth (Figure 2.3) and doubling the speed of the chip. This is made possible by shrinking the transistors again and again, which was anno 2015 still possible as proven by Intel's Broadwell and Skylake [19] generation of processors. Both generations of chips feature transistors measuring only 14 nanometers in size.

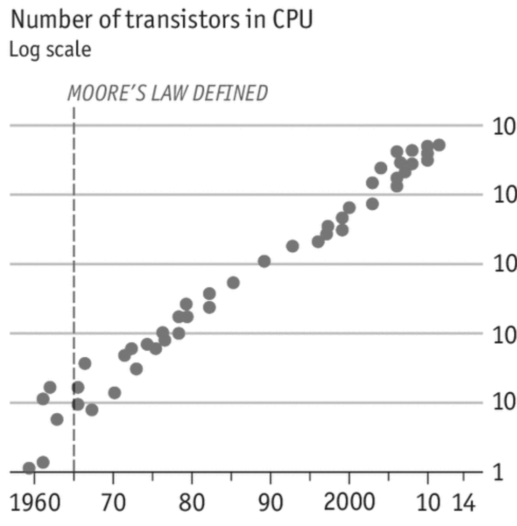


FIGURE 2.3: Moore's law: exponential growth (logarithmic scale) of the number of transistors in a CPU throughout the years [20]

However, due to the laws of physics, shrinking the transistors even further is becoming more and more difficult. Current predictions by Intel state that Moore's law can be upheld at least until 2025. The transistors would then measure just 5 nanometers [20], which is a physical boundary causing Moore's law to fall off. Hereby price is another important factor as chips become relatively more expensive. The price per transistor will then gradually increase.

A solution to this problem is not to rely on single, faster processing units but to use more than just one processing unit. They can then be coordinated in order to combine their efforts, which is viable and cost-effective [2]. The system containing multiple processing units can thus compute in parallel and can be called a “parallel computer”.

Today, three paradigms have been identified for HPC: traditional HPC, grid computing and cloud computing [21]. Within traditional HPC, supercomputers and computer clusters can be located. In this thesis however, we will focus on computer clusters. They are the most widely used system [6], having been installed for research and industrial applications [22–24]. However, to understand some of the available systems, the following sections will provide a brief overview of them – supercomputer, cluster and grid – and will discuss the differences and similarities between them.

2.3.1 Supercomputer

A supercomputer is a broad term. One could use this term for a large computer cluster; it is a supercomputer after all. However, this section interprets ‘supercomputer’ as a single, tightly-coupled stand-alone machine. In 1964 the first ‘real’ supercomputer, the CDC 6600, was released at a price point of $8 million. Since this model completely outperformed any other computer of that time, calling it a supercomputer was justified [25, 26]. Later, in 1976, Seymour Cray delivered the Cray 1 performing at 80 MFLOPS priced at $5 to $7 million. Through vector processing and other innovations, the Cray-1 became the most successful supercomputer in history [27]. Its successors, the Cray X-MP (800 MFLOPS, 1982) and Cray-2 (1.9 GFLOPS, 1985), gained performance but were commercially not as successful as the original model.

Clearly, a dedicated supercomputer was in the past the only way to compute large problems. They used specialized integrated circuits (ICs) and as such consisted of non-commodity hardware. Such machines are still produced today, although on a much lesser scale. Should we compare the performance to an average 2013 smartphone, then we can let the numbers speak for themselves: 1 GFLOPS for a smartphone versus Cray-1’s 80 MFLOPS [28]. We’ve come a long way.

An example of a more modern supercomputer is Cerebro, a partition of the cluster of the Flemish Supercomputer Centre (VSC) [29]. It is a symmetric multiprocessor system (SMP), which involves a large shared memory that is accessible many (multi-core) processors. Cerebro features 64 sockets, each with 10-core Intel Xeon E5-4650 CPUs. 14TB of RAM together with the CPUs are spread out over 32 blades. The blades are interconnected through a SGI-proprietary NUMAlink6 interconnect. Although this system may seem similar to a computer cluster (“nodes connected to a network, working together”), they are different. Since all blades are tightly coupled with each other and controlled by a single operating system, they cannot form a ‘true cluster’ as each node cannot exist independently with respect to the definition of a cluster in Section 2.3.2.

Note that supercomputers should not be confused with mainframes, although both systems may seem similar at first. Their key difference lies in their purpose. Supercomputers are designed for raw performance to solve a single computationally difficult problem at a time. On the contrary, mainframes are designed for usage by many users, concurrently. They function somewhat as 'large servers' and can run complex business applications, like transaction processing or performing tasks on large amounts of data.

2.3.2 Cluster

A computer cluster is a centralized approach for parallelism. The term "cluster" is quite broad however, making a definition necessary. Dongarra et al. [30] narrowed "cluster" down to the definition:

> *"A parallel computer system comprising an integrated collection of independent "nodes" each of which is a system in its own right capable of independent operation and derived from products developed and marketed for other standalone purposes."*

Furthermore, it is possible to make a distinction between a cluster and a commodity cluster:

> *"Moreover, a commodity cluster is a cluster in which the network(s) as well as the compute nodes are commercial products available in the market for procurement and independent application by organizations (either end users or separate vendors) other than the original equipment manufacturer."*

According to the above definitions, compute nodes form the resources in a cluster. These nodes range from especially made computers to general-purpose machines in case of a commodity cluster. The nodes themselves are interconnected to each other by a low-latency, high-speed network, e.g. gigabit Ethernet, SCI, Myrinet or Infiniband [2, 31, 32]. They can communicate with each other by passing messages or by direct internode memory access [23]. Unfortunately, internode communication is slow when compared to intranode communication: a factor of 10 to 1000 times slower, depending on the interconnect network technology used. The number of compute nodes is relatively static, since they have to be bought and installed alongside the networking peripherals.

An advantage of using e.g. message passing as communication between nodes, is that it allows for heterogenity in the cluster. This is the case for both hardware and software. New nodes (new hardware) can simply be added to the cluster, along with the existing older nodes. The same applies for the software or the operating system. Heterogenity grants expandability and scalability [32], but introduces an additional layer of complexity with regards to management.

Regarding access to a computer cluster, it is usually private and locally owned. Within the organization however, resources can be shared among different users. Diverse groups of users representing e.g. projects can simultaneously use a cluster [21]. This implies that the cluster is not necessarily locally owned. It can be located at a remote location for some groups in case of a large organization.

The commodity cluster is not surprisingly the most popular way to do HPC [30, 33, 34], Figure 2.4. The delivered performance is high and costs are relatively low compared to non-commodity clusters. The rise of the commodity cluster has also further allowed code portability to increase [21].

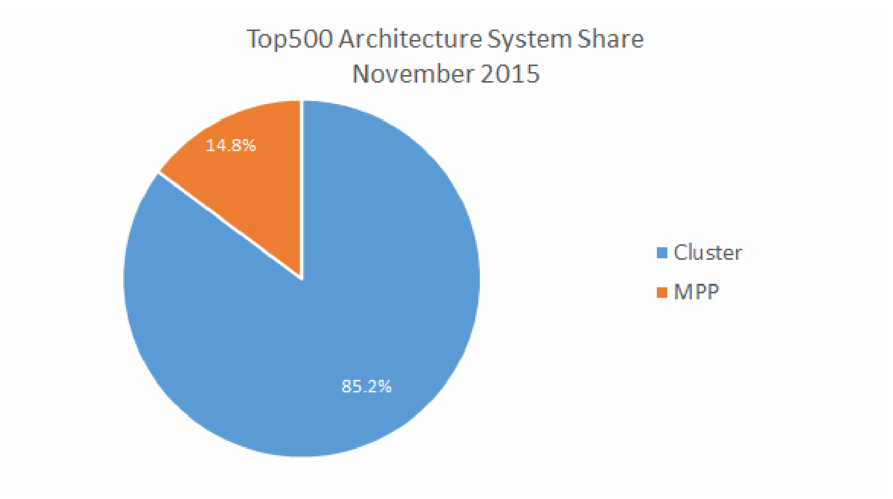


Figure 2.4: Top500's architecture system share, November 2015 [35] (adapted)

2.3.3 Grid

Besides computer clusters, a grid is another system for achieving parallelism. It is a distributed system but unlike clusters, it follows a decentralized approach. While clusters are physically located at one geographical location, grids can be dispersed between multiple locations [31]. A grid can be seen as a a network that ties multiple clusters together. They then are a collection of unified resources to be used by many users. It is essentially a cluster of clusters, allowing even larger problems to be tackled.

Due to its geographically distributed nature, challenges arise. The nodes or clusters are distributed between different groups of owners in physical locations, which makes administration and maintenance a challenge. The same applies to security and authorization since these locations have to be connected. The grid can also be burdened when the user groups simultaneously utilize the grid.

2.3.4 Access Methods

Given that HPC machines are large, expensive and require maintenance, it is only natural that not everyone has access to the HPC infrastructure. First and foremost, the access to HPC infrastructure can be private. It can be used exclusively by a single organization, which in turn can be composed of multiple users or groups, e.g. business units or research teams. It is owned, managed and operated by the organization or a third party. The cluster that will be developed during this thesis falls into this category.

A second method is community-based access. Compared to the private access method, multiple organizations within a certain community now have exclusive access to the infrastructure instead of only a single organization. They usually have some sort of shared concerns, for example, in term of research, mission or policy. One or more of the organizations have the responsibilities of ownership and management. An example of this type of access is the cluster of the VSC, which is accessible to the Flemish universities of Belgium. More specifically, two clusters, ThinKing and Cerebro, can be accessed by the universities of Leuven and Hasselt [29].

A last option is public access. In this case, the HPC infrastructure is made accessible for the general public. Typically, it is owned and managed by a business, academic, or government organization. An example here is the cloud, which is essentially a cluster or a grid. Although the cloud is very similar to grid computing, it has evolved beyond grid computing. Instead of a focus on infrastructure that delivers storage and compute resources as is the case in grids, the focus lies on economy based delivery of abstract resources and services [36]. In particular, these services and resources can take the forms of Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) [31].

IaaS grants the users most of the layers of the cloud. It is almost fully customizable, allowing even the middleware and operating system to be changed and thus making HPC a possibility. A well-known provider of this type of service is Amazon's Elastic Compute 2 (EC2). Evangelinos et al. [8] have shown that it is possible to do HPC on Amazon's EC2 by installing a parallel middleware, MPI. They described the final performance for their problem, atmosphere-ocean climate models, as adequate and comparable to low-cost computer clusters. Zaspel et al. [37] also used Amazon's EC2, in their case for fluid simulations. They observed that the cloud indeed can be used for mid-sized HPC problems.

2.4 Conclusion

This chapter has provided a brief overview of the HPC problem space and the systems that are deployed to solve them. Several example problems were illustrated and give an understanding of how complex or large such problems can be. In chapters 5 and 6, similar problems will be discussed as use cases in order to evaluate the cluster's ability to tackle them. Both of these problems fall in the category of simulations, since they cannot be simply tested in the real world.

First however, with the HPC background in mind, we must make a narrow-down to computer clusters. So far, it was not yet mentioned how clusters allow parallelism. The following chapter will therefore introduce us to parallelism concepts and diverse strategies to implement parallelism within a cluster.

Chapter 3

Parallelism Within a Cluster

Up until now, HPC machines were discussed broadly. In the remainder of this thesis however, we will focus on computer clusters. In the previous chapter, a cluster or a cluster node was described as a 'black box'. Now, this chapter aims to dive deeper in the ways to achieve parallelism in a cluster. Here, the main way to achieve parallelism depends on the way memory is made accessible. Besides the way memory accessibility is of influence, other technologies and other hardware besides the CPU for calculations are available as well.

3.1 A Single Multi-Core Node: Shared Memory

A single compute node in a cluster is a computer, containing a CPU with typically more than just one core. These nodes, or multi-core systems, can then execute multiple processes at a time. Within a single process, multiple threads of execution can exist. Since threads share data, they are one possible mechanism for parallel programming. Of course, for this to be faster than an equivalent sequential single-threaded execution, it is assumed that the CPU indeed consists of multiple cores. For a four-core system, that means that the four threads of a sole process each get mapped on a CPU core.

Currently, the de facto standard for multi-core processing in shared memory is OpenMP, the Open standard for shared memory MultiProcessing. OpenMP is not a stand-alone programming language, but it is an extension to the programming languages C/C++ and Fortran. It is based on special compiler directives and a set of supporting library calls. OpenMP's main approach to parallelism is the parallel execution of loops.

OpenMP operates with a fork-join programming model in which a program with a single thread of execution (the master thread) spawns a team of threads to carry out work concurrently. Parallelism is gained by declaring e.g. a loop to be 'parallel', by using the `parallel` directive among others: this is the 'fork', the master thread creates a number of worker threads. After processing the parallel part of the code, the spawned threads are terminated, with only the master thread remaining: the threads are joined. Moreover, since OpenMP is fundamentally based on multithreading, dynamic parallelism is a possibility since the number of execution streams operating in parallel, or threads, can vary in time. Figure 3.1 summarizes and illustrates the fork-join model.

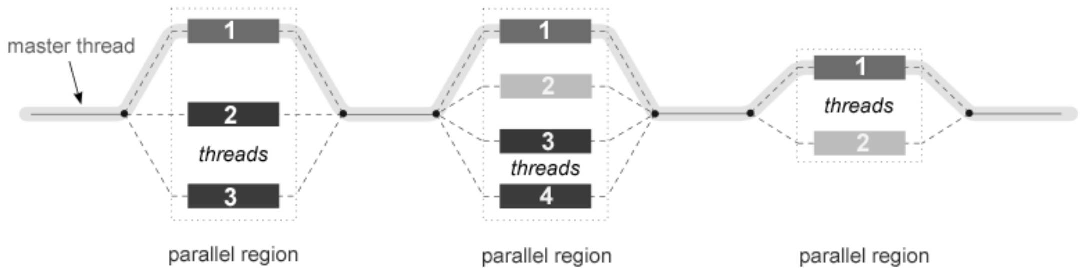


FIGURE 3.1: Fork-join model of OpenMP [38]

In Listing 3.1, the very basic "Hello world" program is shown as its OpenMP variant. The parallel region is clearly visible and in this case only executes a print statement. It is surrounded by the OpenMP `parallel` directive, to indicate that the code lines in the following block should be run in parallel. The lines of code before and after the parallel region are run by the master thread only. It spawns (forks) and joins threads respectively. The output of the program would be "Forking", followed by n lines of "Hello world X" messages. Here, X represents the thread number, made available by the `omp_get_thread_num()` call and n has the value of $(number\ of\ cores) \times (number\ of\ threads/core)$. Finally, only the master thread prints "Joining".

```
int main (int argc, char *argv[])
{
    printf("Forking\n"));
    #pragma omp parallel
    {
        printf("Hello world %d\n", omp_get_thread_num());
    }
    printf("Joining\n"));
    return 0;
}
```

LISTING 3.1: The hello world code example for OpenMP

OpenMP is generally easy to use, that is, it requires a relatively low amount of additional programming work. Since it is supported by almost all main compiler vendors, code portability is gained and the programmer's life is eased even further. However, the programmer is still responsible for any synchronization that his code may require, as race conditions, data conflicts or deadlocks are possible by default and remain a threat. If we look back at the output of the code in Listing 3.1, the n "Hello world X" lines are not necessarily printed 'in order', that is, X can be printed as $2, 3, 0, 1$ for 4 threads. There is no guarantee that the threads would print $0, 1, 2, 3$ as perhaps expected.

Another limitation is that OpenMP is designed for shared-memory systems, or in a cluster context, just for a single node. Deploying it cluster-wide is not an option, although attempts have been made. Intel's Cluster OpenMP [39] is an extension to OpenMP, allowing it to operate even with distributed-memory systems. Although this proved to be successful for smaller applications, more complex applications required additional optimization [40]. However, the project was discontinued. Another strategy is needed, tuned for distributed-memory systems, which is discussed in the following section.

3.2 Multiple Nodes: Distributed Memory

Naturally, a computer cluster consists of more than just one compute node. Each node has its own memory, which is by default not shared with the other nodes. Parallelism within this distributed memory thus requires a different approach than the shared memory case with OpenMP.

The answer we are looking for is to make use of a Message Passing Interface, or MPI in short, a widely accepted standard for distributed-memory programming. It is an API that allows communication by passing messages between processes through library calls. With MPI, a program consists of multiple processes that by default share no data at all. All data that needs to be shared, needs to be done so explicitly through MPI. Data sharing or communication in general can mainly be done in two ways, that is, through peer-to-peer calls or through collective calls.

The peer to peer communication category involves, as its name implies, calls between two peers, represented as two processes. The processes can send and receive data to and from each other respectively. In contrast, collective calls involve all participating processes in the cluster. An example of a collective call is the broadcast call, where one process can share its data with every other participating process. Note that the term 'participating process' was carefully chosen. Both types of calls use a communicator to pass messages. The default communicator is `MPI_COMM_WORLD`, which is always initialized with all available processes. Other communicators can be defined as well. Collective calls can then be used to e.g. broadcast data to all processes of a communicator that is a subset of `MPI_COMM_WORLD`.

In Listing 3.2, the basic "Hello world"-program is revisited, this time as an MPI implementation analogous to the OpenMP version. `MPI_Init()` spawns a number of processes, usually externally specified by the programmer. After this call, the parallel section starts: every process requests the communicator, `MPI_COMM_WORLD` for its rank. A rank can be seen as an address within the communicator, or in high-level terms, the ID of the process, similar to the thread ID in the OpenMP example. The "Hello world"-message is then printed together with the requested rank, after which the the processes are ended by `MPI_Finalize()`. Note that in this very basic example, the `rank` variable is local to each process. No peer-to-peer or collective calls for data sharing were needed.

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

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Hello world %d\n", rank);

    MPI_Finalize();
    return 0;
}
```

Listing 3.2: The hello world code example for MPI

The main advantage of MPI is naturally the ability to allow code execution over distributed memory systems. Although it is intended for distributed memory, it can be used in a shared memory context as well. Another advantage of the MPI programming model is that data distribution and synchronization are completely in the programmer's control. This however, is a disadvantage as well. Because the programmer has to do these things himself, strategies for data partitioning and managing the communication have to be carefully developed. This results in code that can be hard to write, especially when compared to e.g. the OpenMP approach. In an OpenMP program, all the programmer may have to do is adding a compiler directive; while MPI may require a lot of code rewriting and explicit data sharing.

Another disadvantage is the duplication of data. Sometimes it is necessary for a lot of processes to have access to a piece of data, which then has to be shared. Since the MPI model is distributed, data is duplicated and results in an increased total amount of memory usage when compared to the shared memory approach.

3.3 Hybrid Approaches

Looking back at both the advantages and disadvantages of OpenMP and MPI, one can wonder if they can be combined to reduce some of their disadvantages. In other words, is it possible to use the best of both worlds? The answer is yes: both programming models can indeed be combined. The hybrid OpenMP-MPI model even matches the cluster architecture: MPI can be used for the communication between compute nodes with distributed memory, while OpenMP can then in turn allow parallelization within a shared memory node.

With the hybrid model, some of MPI's disadvantages can thus be eliminated. Since MPI launches separate processes, data is duplicated. On every node, OpenMP can overcome this problem by needing only a single process but with multiple threads. Data now resides in the shared memory of each node and needs not to be duplicated on a single machine, freeing otherwise occupied memory. Furthermore, since only a single process is needed per node, the MPI communication over the network in the cluster is reduced, which is the main improvement for performance and scalability [41].

Although advantages are gained by combining OpenMP and MPI, the hybrid scheme needs to be nuanced. Advantageous as this approach may seem, translating the hybrid model to code can be rather difficult. Since additional complexity is introduced, coding will require greater time and effort. A trade-off has to be made: does the problem *require* a hybrid approach, that is, is it worth investing time and effort? In some cases, yes, due to e.g. memory or networking constraints. Although this question has to be asked on a per-problem basis, the potential increase in scalability and performance may be worth the effort.

Besides programming complexity, other performance problems can arise. Chorley and Walker [41] found that for their application, only MPI instead of the hybrid solution is better for clusters with few cores per node as the shared memory overhead is relatively large. Another contributing factor here is the communication, which has not significantly been reduced. Generally though, more cores per node leads to the hybrid model outperforming the MPI-only equivalent.

Lastly, since the hybrid approach depends on MPI, the interconnect network used in the cluster also effects it. In the same study [41], it was found that a purely MPI-based solution can outperform the hybrid version when using a fast interconnect such as Infiniband. This is the case for a larger number of cores than when using a slower interconnect, e.g. 10 GigE. In other words, environments with a relatively slow interconnect can benefit relatively more from the hybrid approach.

3.4 Other Strategies

Up until now, this chapter assumed that the general-purpose CPU is used for all computations and this for every compute node in the cluster. However, it is possible to use different kinds of hardware besides the CPU. Often, accelerators are deployed in clusters. Several nodes can then be designated as "accelerator nodes", equipped with a device such as a graphics processing unit (GPU) or a coprocessor next to the host, the CPU. This section will briefly cover when and how they can be used.

3.4.1 GPU Programming

A graphics processing unit or GPU is a processor designed for graphics processing. The GPU was designed around the 'graphics pipeline', where identical operations are performed on many data elements in order to render graphics. Although the GPU was originally intended for graphics only, it has evolved over the years and can now also handle non-graphics computations.

CPUs serve for general purpose computations with a moderate efficiency. By restricting functionality of a processor, it may be possible to raise its efficiency, or lower its power consumption at similar efficiency. A GPU is not designed for the flexibility of a CPU. Rather, the GPU can be seen as a collection of 'single instruction, multiple data' (SIMD) processors. In SIMD, each processor executes the same instruction in parallel, but this can be done on different input data. Since the the GPU can handle parallel computing and has a much higher performance when compared to a CPU, it is suitable for HPC applications.

Enabling technologies here are CUDA, OpenCL and DirectCompute. Although the GPU was described as SIMD, CUDA allows more than traditional SIMD architectures, increasing flexibility. Typically, the CUDA architecture is referenced to as single instruction, multiple threads (SIMT). It has proven to be quite successful at programming multithreaded many-core GPUs and scales transparently to hundreds of cores. Currently, scientists in both academia and industry use CUDA to achieve dramatic speedups for their code [42]. However, it is only available for NVIDIA devices. The other technologies have a broader group of supported devices and function largely the same.

In a cluster, some nodes can thus be equipped with a GPU. A user can choose to solely use the single node with its GPU. For larger problems, MPI can again be deployed in order to use multiple GPUs concurrently. Lastly, even OpenMP can be added for some scenarios as demonstrated by Yang et al. [42].

3.4.2 Coprocessor

A coprocessor is an accelerator and has the same purpose as that of the GPU: accelerating computations. A recent example of a coprocessor is the Intel Xeon Phi, based on Intel's Many Integrated Core Architecture (MIC) [43]. The Xeon Phi consists of many 64 bit x86 cores combined onto a single chip. Although this may seem similar to the GPU albeit with fewer cores, some key differences can be found.

Compared to the GPU, not only a SIMD/SIMT model is supported. The Xeon Phi has general purpose cores, which means it can run 'whole' programs. GPUs on the contrary need the SIMD requirement to be present in the problem case they are solving to achieve performance. Furthermore, the cores accept ordinary code that can also run on ordinary CPUs. As a result, existing code parallelized with e.g. OpenMP or MPI is supported.

3.5 Utilizing the Cluster Resources

Now that we know how parallelism can be achieved in a cluster, it is important to know how the cluster's parallelism can be effectively utilized by users. After all, every node in a cluster can in theory act as an individual computer. There is no single operating system covering the entire system, which may be the case in e.g. SMP systems. Through

MPI, code can be run if the participating hosts in the cluster are specified. This requires knowledge of the physical cluster setup, which is preferably not exposed and is not user-friendly. Also, is is possible that the hosts may already be busy at that point, resulting in an execution failure or bottlenecking as the resources are already in use. A better approach is needed.

In a cluster, a resource manager should be deployed which controls the distributed resources. The manager can mark a node as 'occupied' when e.g. all cores of its multi-core processor are in use. Users also no longer need to start their program with plain MPI commands. They can simply submit a 'job' (i.e. a script containing or launching a portion of work) to the manager which will then in turn try to find enough resources as specified by the user for the job to complete.

Jobs that are submitted by users will end up in a queue if not all requested resources are immediately available. Here, the resource manager is usually assisted by a job scheduler. Sometimes, a FIFO approach for the job queue is sufficient, which is especially the case in an environment with just a single queue. However, especially in multi-user environments with multiple queues (based e.g. on expected job runtime: 2hours, 4hours, 12hours etc.), the scheduler has to decide which job gets to run first. The same FIFO scheduling method can be taken, but more advanced techniques are preferable to respect resource usage fairness for all users.

3.6 Conclusion

This chapter covered ways to allow parallelism in a cluster. Both shared memory and distributed memory strategies were discussed. Other strategies were mentioned as well, that is, making use of a GPU or coprocessor. Our cluster will not make use of such peripherals, however. We will focus solely on the CPU, since there is no GPU available on inexpensive SBCs that can be used for HPC purposes. Furthermore, we have gained knowledge of what software components are of necessity, that is, a resource manager and a scheduler.

With this chapter in mind, it is now a possibility to describe the architecture of our own cluster. The following chapter will thus describe how the cluster is set up. It will explain what choices were taken to allow parallelism and the usage of it.

Chapter 4

SBC Cluster Architecture

In the previous chapters, a narrow-down to parallelism within clusters was made. With this in mind, it is now possible to build and describe our own cluster architecture and components. However, the single-board computers that will be used in the cluster will not have the typically used x86 architecture for processors. Instead, they will have ARM processors, making it interesting and necessary to gain an understanding of the importance of the ARM architecture first.

4.1 Processor Architecture

Advanced RISC Machine or ARM is an instruction set, developed by ARM Holdings. It is a reduced instruction set computing (RISC) architecture designed for processors. As of 2015, ARM processors have a market share of more than 95% for smartphones, more than 85% for tablets and over 90% for wearables [44]. Logically, as ARM processors were designed to meet the needs of the handheld sector. Handhelds and wearables have a limited battery capacity and therefore require low-power and energy-efficient components. A small form factor is also required as space on the devices is limited. These embedded processors do naturally not have the same performance as their counterparts in desktop and server environments, where ARM has a lower penetration grade. However, having a market share of less then 1% for servers in 2015, that same share is expected to raise to 20 to 25% for the year 2020 [45, 46].

What exactly makes ARM processors so attractive? Besides being energy efficient and low-cost, other advantages are present as well. One of ARM's most interesting aspects is its way of licensing, illustrated in Figure 4.1. The ARM instruction set architecture (ISA) is defined by ARM Holdings. With regards to the ISA specification, an implementation is made in the form of a CPU core which it is licensable to many licensees, e.g. Broadcom, Texas Instruments and Samsung. The licensees in turn integrate the ARM core together with peripherals into a system-on-chip (SoC). Finally, the SoC is sold to board makers who place the SoC and other components on a board.

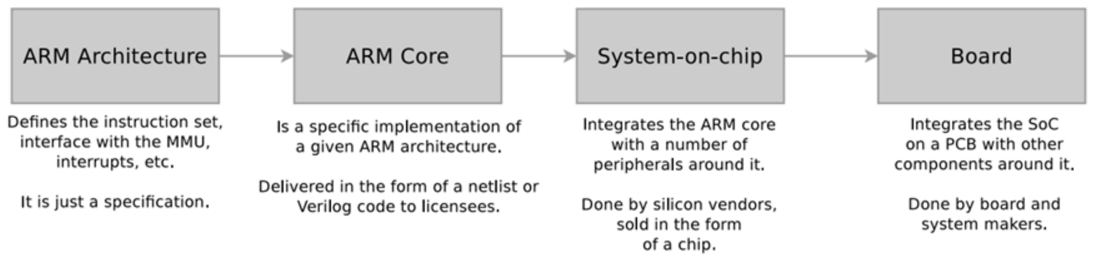

ARM Architecture
Defines the instruction set, interface with the MMU, interrupts, etc.
It is just a specification.
ARM Core
Is a specific implementation of a given ARM architecture.
Delivered in the form of a netlist or Verilog code to licensees.
System-on-chip
Integrates the ARM core with a number of peripherals around it.
Done by silicon vendors, sold in the form of a chip.
Board
Integrates the SoC on a PCB with other components around it.
Done by board and system makers.

FIGURE 4.1: The ARM implementation pipeline [47]

If we apply the ARM implementation pipeline on the Raspberry Pi B+, we can see that originally the CPU core was based on the ARMv6 architecture. The architecture was implemented in an ARM1176JZF-S core, licensed to Broadcom who created the BCM2835 SoC. Lastly, the Raspberry Pi Foundation used Broadcom's SoC on its board: the Raspberry Pi B+. The whole process is schematically summarized in Figure 4.2.

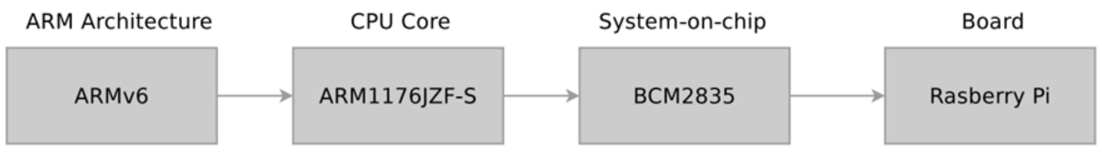


FIGURE 4.2: A case of the ARM implementation pipeline: the Raspberry Pi B+ [47]

In stark contrast with ARM's licensing system, there is Intel's x86 architecture. Intel has, with exceptions, only licensed x86 to AMD. As ARM has many licensees, CPU costs are further reduced as there are more vendors to compete with each other. Intel's only competitor in the x86 segment is AMD, leading to a de facto monopoly and keeping prices relatively high.

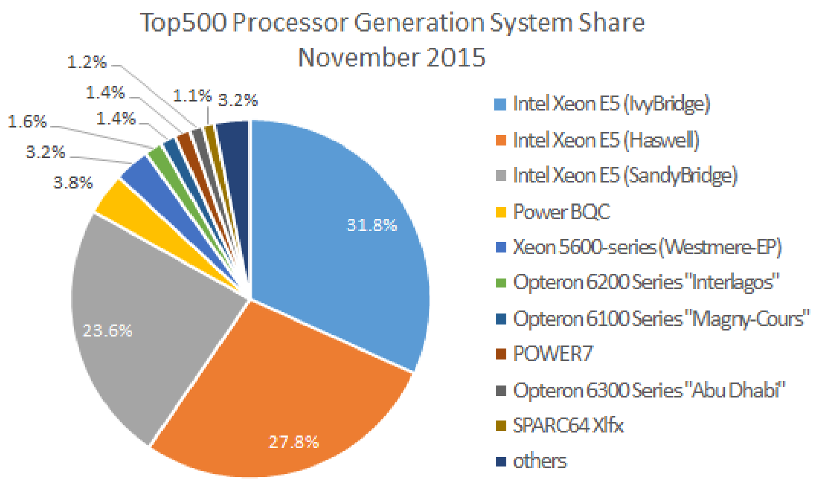


FIGURE 4.3: Top500's processor generation system share, November 2015 [35] (adapted)

Today in HPC, CPUs are supplied mainly by Intel but also by AMD and IBM. Intel dominates the top 500 HPC systems clearly (Figure 4.3). As previously mentioned, ARM currently has only a small percentage of the server share. Here too, Intel dominates as their processors provide excellent performance but come at high power usage and cost. Therefore, it is interesting to investigate whether ARM processors can compete with the conventional processors used in this sector.

4.2 Single-Board Computer Clusters in Literature

Several attempts to create a cluster consisting of low-cost single-board computers have already been undertaken. Looking at literature, three categories of clusters can be distinguished. This section will briefly discuss their pros and cons on a general level.

The first category consists of hobby projects. Many of these projects are described in blogs, tutorials, how-to guides and so forth [48–51]. Unsurprisingly, the Raspberry Pi and its successors are the most used boards in this category as the Pi is the original, most popular single-board computer on the market. However, this kind of clusters is not discussed in depth in their respective articles since they only discuss the creation and setup of the system.

A second category can still be found in hobby projects. These projects are conducted mainly in the same way as in the first category but have a critical point of difference: they go beyond the description of the setup and the final "It works!" conclusion [52, 53]. In these projects, the choice of hardware is more reasoned and the setup is mentioned in greater detail. But most importantly, the performance is assessed through benchmarks. Furthermore, performance is often compared with similar systems or even server-grade processors. A different kind of project is the Hypriot Cluster Lab [54], where again a cluster has been created of Raspberry Pis. This time however, it is a Docker cluster, allowing virtualization and containers within the cluster.

Lastly, academic research has been performed as well. Projects in this category range from educational projects by professors for their courses to published papers [55–58]. These papers discuss the cluster setup in detail and study the performance (FLOPS, I/O rates, etc.) through benchmark suites, such as HPL [59]. They conclude that ARM clusters are interesting and promising towards the future, but are currently (that is, in 2013) most useful in educational settings.

From hobby projects to academic research, many successful attempts have already been undertaken. However, what they all have in common is that none of the projects runs code from actual, real-world use cases. All benchmarks performed are 'artificial' in a way, a stress test. Yes, performance can be somewhat objectively assessed depending on the testing suite used. What is missing is an actual case from a user to test the system; something this thesis will contribute to.

4.3 Choosing the Hardware

Starting this thesis, a decision had to be made about which single-board computer is suited for utilization in a computer cluster. This is no easy task as currently, in August 2015, many varieties of single-board computers are available. We are, however, inspired by the Raspberry Pi and its successors. In Table 4.1, several interesting system specifications of two Raspberry Pi models are listed. Note that not every specification is listed. Components such as the GPU, GPIO (general purpose input/output) capabilities, USB ports etc. are important when making a decision. In a cluster setup however, they are of lesser necessity.

Name	**Raspberry Pi B+**	**Raspberry Pi 2 B**
Date	February 2012	February 2015
ARM Arch.	ARMv6	ARMv7
CPU Arch.	ARM1176JZF-S	ARM Cortex-A7
SoC	Broadcom BCM2835	Broadcom BCM2836
Cores	1	4
Clock Speed	700MHz	900MHz
RAM	512MB SDRAM	1GB LPDDR2
Storage	MicroSD	MicroSD
Networking	10/100 Mbps	10/100 Mbps
Price	35$	35$

TABLE 4.1: Single-board computer reference models: successors of the original Raspberry Pi

The progression of the Raspberry Pi over the years is interesting to note. The Raspberry Pi B+ is currently somewhat outdated when compared to the newer 2 B model. The newer model has four cores clocked at higher speeds – an enormous improvement in computational power. We will consider the Raspberry Pi 2 B model as our base model, since it is a well-know board that provides a balance between price and features. Moreover, at only 35$, it is not very expensive. An increase in price comes with more or better features, which can be continued until we reach the capabilities of a full-fledged computer, which is not our target. Rather, the price per board has to remain low. Thus, we aim at a comparable, if not better, single-board computer compared to the Raspberry Pi 2 B. The following criteria aid to find a board with our target specifications:

- **ARM architecture**: ARMv7. We want the latest, most performing architecture.
- **Cores and speed**: Four or more cores clocked at least 1GHz. Having more cores per board is cost-effective, as it is e.g. cheaper to buy a single board with four cores instead of two dual-core boards of the same architecture.
- **RAM**: At least 1GB. This is the absolute minimum, since it will still partly be occupied by the operating system and other services. This way, we can guarantee an availability of around 200MB of RAM per core (in case of 4 cores).

- **Networking**: 1000 Mbps is preferred. It is vital in a cluster to have a high-performance network as it is possible that a large amount of data has to be transferred at once. Of course it cannot be directly compared to Infiniband but it should nonetheless be an improvement over the Pi's 100 Mbps, which could quickly bottleneck the cluster.

- **Price**: As low as possible, while keeping performance intact. The price should be comparable to the Pi's 35$. We define an upper limit to be ± 45$. More advanced boards quickly make a 'jump' in price point to 80$ and higher, which exceeds our upper price limit by a large portion. Although these boards still remain relatively cheap, as a component for a cluster, this would result in a large increase in total cost.

With these specifications in mind, a subset of the available single-board computers can be made. Table 4.2 lists several interesting boards with their specifications. This selection was made mainly by filtering out the more expensive boards – that is, boards with a price tag higher than 50$. This price point was chosen just above our upper price limit, to perhaps still include some potentially interesting boards.

Name	**A20-OLinuXIno-LIME2**	**ODROID-C1+**	**BeagleBone Black rev C**	**Orange Pi Plus**	**Banana Pi M2**
Date	September 2014	August 2015	2014	February 2015	May 2015
ARM Arch.	ARMv7	ARMv7	ARMv7	ARMv7	ARMv7
CPU Arch.	ARM Cortex-A7	ARM Cortex-A5	ARM Cortex-A8	ARM Cortex-A7	ARM Cortex-A7
SoC	Allwinner A20	Amlogic S805	Sitara AM335x	Allwinner H3	Allwinner A31
Cores	2	4	1	4	4
Clock Speed	1GHz	1.5GHz	1GHz	1.6GHz	1GHz
RAM	1GB DDR3	1GB DDR3	512MB DDR3L	1GB DDR3	1GB DDR3
Storage	MicroSD, SATA	MicroSD, eMMC	4GB eMMC	MicroSD, 8GB eMMC, SATA	MicroSD
Networking	10/100/1000 Mbps	10/100/1000 Mbps	10/100 Mbps	10/100/1000 Mbps	10/100/1000 Mbps
Price	45$	35$	45$	50$	45$

TABLE 4.2: Hardware comparison of several single-board computers

The first board, the A20-OLinuXIno-LIME2 [60], has most of the specifications we desire. Memory, networking and price point are according to our wishes. Its open-source hardware and software is a plus. Unfortunately, its processor has only two cores, where we expect it to have at least four. The Odroid-C1+ [61], our second selected board, is very similar (with regards to our selected specifications, at least) to the LIME2. Although similar, it has an additional two cores, is higher clocked and costs less. The BeagleBone Black (rev C) [62] is a great board for prototyping as it has many GPIO options available. Unfortunately, where it shines in connectivity possibilities it lacks in other hardware. Having just a single core processor, half as much RAM and 100 Mbps Ethernet it is not an ideal board for a cluster, especially when compared to the previously discussed boards. The Orange Pi Plus [63] and the Banana Pi M2 [64] are both very similar boards, the only really notable differences being the SoC and the CPU clock speed.

Purely based on hardware, we can summarize that the OlinuXIno-LIME2 and BeagleBone Black can be dropped from our selection. The Banana Pi's manufacturer claims to be open-source, but is neither cooperative nor provides details about the hardware for the board. The Banana Pi models have also been manufactured by different companies – each in order to hitchhike on the Rapsberry Pi's success. As a result, there is no real cohesion between the models. Therefore, we decide not the further consider the Banana Pi. Lastly, both the Odroid and Orange Pi board have active communities. The hardware and software are (mostly) open-source. Despite the Orange Pi being slightly better in the hardware aspect, the price point of the Odroid board remains more attractive. Therefore, the Odroid-C1+ (Figure 4.4) is our final decision.

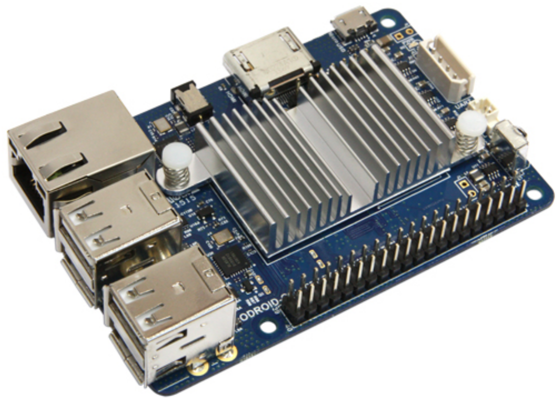


FIGURE 4.4: The Odroid-C1+ board [61]

At the end of February 2016, a newer model of the Raspberry Pi was released. With a higher clocked 64-bit processor and new features such as Wi-Fi and it being an upgrade to the Raspberry Pi 2 B, it still lacks gigabit Ethernet (Table 4.3). In March, a newer version of the selected Odroid-C1+ became available: the Odroid-C2. This board has similar upgrades (but no wireless connection) as the Pi, but again beats the Pi on the CPU end, among other specifications, while being slightly more expensive. If this thesis was started around March 2016, the Odroid-C2 would have been the board to choose.

Name	**Raspberry Pi 3 B**	**ODROID-C2**
Date	February 2016	March 2016
ARM Arch.	ARMv8	ARMv8
CPU Arch.	ARM Cortex-A53	ARM Cortex-A53
SoC	Broadcom BCM2837	Amlogic S905
Cores	4	4
Clock Speed	1.2GHz	2GHz
RAM	1GB DDR2	2GB DDR3
Storage	MicroSD	MicroSD
Networking	10/100 Mbps	10/100/1000 Mbps
Price	35$	40$

TABLE 4.3: Single-board computer reference models: March 2016

4.4 Compute Node Setup

As the Odroid boards are intended as computation nodes in the cluster, they require a minimal setup. Considerations regarding this setup are discussed.

4.4.1 Operating System

Two official system images are provided: Ubuntu 14.04 and Android 4.4.2. Naturally, the Android image is not an option since we want to build an HPC cluster. The Ubuntu image is interesting, but is unfortunately a desktop version, containing many superfluous packages.

Besides these official images, many other community-driven operating system images are also made available on the forums. Here we opted for a minimal Debian Wheezy image [65] instead of the official Ubuntu image. The need to have an actual minimal install is the primary motivator, as many unnecessary packages would need to be checked and removed first otherwise. This way, the system can be built bottom-up instead of having to break it down first.

4.4.2 RAM Allocation

By default, the 1GB of RAM on the board is shared between the CPU and GPU. Since the Odroid board are used as computation nodes, their GPUs for display through their HDMI ports are not necessary. Several tests were conducted in order to change the CPU/GPU-memory split in order to reduce the amount of memory allocated to the CPU. As an outcome, three configurations are possible, shown in Table 4.4.

RAM (CPU)	**GPU limitations**
846MB	None, full GPU functionality
983MB	GPU disabled
958MB	Console only

Table 4.4: RAM allocated to the CPU accompanied with limitations to the GPU on the Odroid-C1+

A maximum of 137MB of RAM can be gained for the CPU by completely disabling the GPU. As interesting as this may be, ultimately the third option was chosen. Full GPU functionality and only console functionality differs 25MB. Despite this being a small amount of memory 'lost' for the CPU, it opens up options for possible debugging scenarios. A monitor for investigation can still be connected this way, with a minimal footprint on the memory.

4.4.3 Specific Hardware Configuration

The Odroid-C1+ features a connector for a real time clock battery. The battery is not included by default. When the board is powered off and disconnected from a power supply, the system time settings of the boards are lost. The date is reverted to 1970, leading to undesirable behavior of several processes starting on boot. RTC batteries were not purchased in order to reduce cost and maintenance overhead, as the batteries would have to be replaced from time to time. Instead, every node in the cluster synchronizes its clock with a local NTP server very early during the boot process.

The Odroid-C1+'s SoC, the Amlogic S805, provides a hardware accelerated random number generator (HRNG). When enabled, the CPU is offloaded by the HRNG. Besides sparing the CPU, it also increases the amount of randomness to applications that require it, e.g. cryptography or statistics. This results in an increase in security and performance for those applications. Although many applications just request one 'truly' random number from the operating system (from e.g. `/dev/random`) to seed their internal pseudo random number generator, it is an increase in performance nonetheless.

4.5 Network Setup

Since the cluster can be considered as an entity on its own consisting of many components, it needs its own /24 subnet. Specific IP addresses within the subnet are summarized in Table 4.5. All hardware components of the cluster are illustrated schematically in Figure 4.5.

IP range	**Purpose**
192.168.1.1	Gateway
192.168.1.96 – 192.168.1.99	Switches
192.168.1.100 – 192.168.1.200	Compute nodes reserved
192.168.1.201	Monitoring host
192.168.1.202	Base node

TABLE 4.5: IP addresses for devices in the network

At the edge of the subnet, a simple pfSense box has the role of a gateway. It performs network address translation so access from the subnet to LANs or the internet is possible and vice versa. This can pose potential security risks; remote access can be restricted with its built-in firewalling capabilities. Besides filtering traffic, other services are running as well. A DHCP server provides IP addresses for the network. Together with a DNS server, a dynamic DNS (DDNS) service is enabled. This is a necessity for a better working of the cluster, as it allows dynamic host name resolution for the compute nodes. Their names can then be resolved, which is a lot more flexible than hardcoding their IP address/hostname pairs. Finally, an NTP server is running in order to support functionality in offline environments.

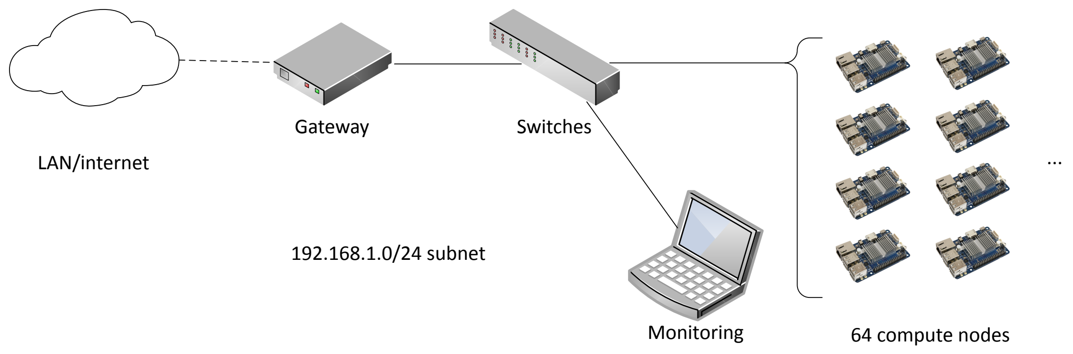


FIGURE 4.5: Schematic setup of the hardware

Four 24-port D-Link gigabit switches are installed to provide the necessary connections to all compute nodes and other devices. Between switch pairs are four-way connections to increase bandwidth and avoid bottlenecks. The 64 compute nodes are spread out over two cases, each with two of the switches. Figure 4.6 shows the final physical setup.

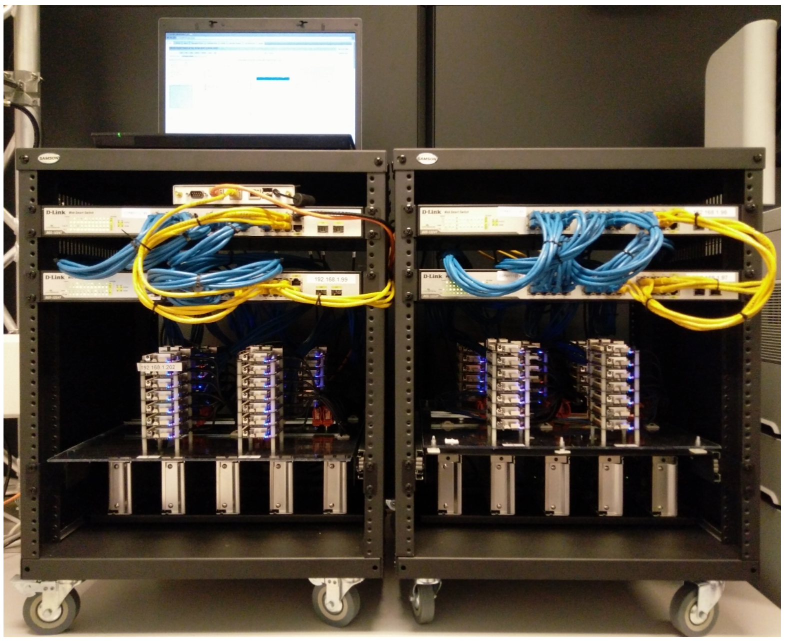


FIGURE 4.6: Physical setup of the hardware

Note that in Figure 4.6, no fans are visible. The Odroid boards in principle passively cool themselves. However, after fully utilizing the cluster, temperatures rose due to a lack of airflow and the boards being closely stacked next to each other. Temperatures of 79 degrees Celsius were measured on the boards, with thermal throttling starting at 80 degrees Celsius. Consequently, fans were added in front of the cluster to avoid any damage and thermal throttling.

4.6 Software Setup

This section briefly covers the most important (software) aspects present in the cluster. It is discussed how they operate and where the components reside in the cluster.

4.6.1 Resource Management and Scheduling

As previously mentioned in Section 3.5, a resource manager is required. For clusters, two well-known resource managers are Torque [66] and SLURM [67]. Besides some differences between them, they both provide the same high-level features. Torque is based on the Portable Batch System (PBS) and is also used at the cluster of the Flemish Supercomputer Centre (VSC). Since researchers at the EDM and Hasselt University already have experience with the VSC cluster, opting for the Torque resource manager is the preferred choice. This way, PBS scripts for job submission can largely be reused. Furthermore, SLURM uses a different syntax for its job files and commands, which would require additional user training.

A Torque cluster consists of one head node and one or more compute nodes. The monitoring host is the head node in our context. It runs the `pbs_server` daemon to which PBS jobs and PBS commands can be submitted. The server process also contains the job queue(s), where in our case a single queue is sufficient for a single-user scenario. Compute nodes run the `pbs_mom` daemon in order to make communication with the head node possible. Figure 4.7 illustrates the Torque setup schematically.

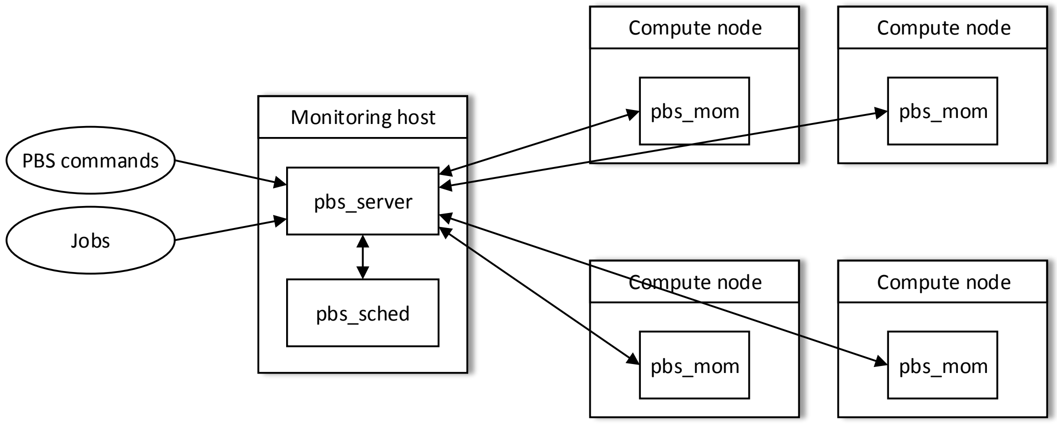


FIGURE 4.7: Torque: schematic overview for four compute nodes

The jobs in the queue of the `pbs_server` daemon will not automatically start; a scheduler which manages the queue is required on the head node. The scheduler interacts with the server to make local policy decisions for resource usage and to allocate nodes for jobs. The schedulers Moab [68] or Maui [69] come to mind, but are somewhat too advanced in features for our relatively simple cluster. Since our cluster is limited in size and is mainly a single-user environment, a more simple scheduler is needed. Torque provides a simple FIFO scheduler, which is sufficient for our case.

Users can submit jobs using the `qsub` command to `pbs_server`. When it receives a new job, the scheduler is informed and tries to find the nodes required for the job. When found, the list of nodes is reported to the server. Finally, the server sends the new job to the first node in the node list and instructs it to launch the job.

Of course, the PBS server first needs to know which hosts in the network are part of the cluster. This can be done by specifying a node file, filled with entries in the format `<hostname> [np=] [properties]`. Here `<hostname>` is as it implies, the hostname of the compute node. Entries between square brackets are optional; `np` being the amount of available cores and `properties` being one or more user-defined properties. Torque can be instructed to auto-detect the `np` value. In Listing 4.1, a few sample entries are shown. As properties, the physical locations were supplied to ease any troubleshooting that may occur.

```
node-001e06201146 np=4 cluster01 rack01 01
node-001e062011b2 np=4 cluster01 rack01 02
node-001e06200fa1 np=4 cluster01 rack01 03
node-001e06201099 np=4 cluster01 rack01 04
```

Listing 4.1: Sample node file used by Torque

Lastly, after the user's job has run, he is provided with two output files for the standard output and error streams. For his convenience, information about the used resources is appended to the standard output file. This information is generated through Torque's prologue and epilogue scripts, called at the start and end of the execution of the job, respectively. These scripts are not provided by default and have to be created by the cluster administrator. Thus, prologue and epilogue scripts were written. A sample standard output file is shown below in Listing 4.2.

```
<user standard output>
==============================================================
Epilogue Args:
Date: Tue Apr 26 15:54:36 CEST 2016
Allocated nodes:
node-001e06201196
node-001e06201196
node-001e06201196
node-001e06201196
node-001e06201146
node-001e06201146
node-001e06201146
node-001e06201146
node-001e062011b2
node-001e062011b2
```

```
node-001e062011b2
node-001e062011b2
node-001e06200fa1
node-001e06200fa1
node-001e06200fa1
node-001e06200fa1
Job ID: 83.cluster-monitor
User ID: cluster
Group ID: cluster
Job Name: job.pbs
Session ID: 3413
Resource List: neednodes=4:ppn=4,nodes=4:ppn=4,walltime=00:10:00
Resources Used: cput=00:22:20,energy_used=0,mem=2368828kb,vmem
   ↪ =22264712kb,walltime=00:00:18
Queue Name: batch
Account String:
```

Listing 4.2: Sample Torque standard output file

4.6.2 Message Passing

As Section 3.2 discussed, an interface for message passing is needed to allow parallelization in distributed memory environments. Two well-known open-source MPI libraries exist: OpenMPI [70] and MPICH [71]. Both libraries are very similar to each other but have some differences, e.g. MPICH does not support Infiniband while MPICH does. However, for our cluster setup, both libraries serve the same purpose and as such there is no real preference to pick one above the other. Ultimately though, OpenMPI was chosen as it is also used on the VSC cluster.

OpenMPI has to be installed on all the compute nodes and the master node, the monitoring host. In principle, it can simply be installed with several `apt-get` commands. Once installed, jobs can be run in parallel on multiple nodes with the `mpirun` command:

```
mpirun -np 4 --hostfile hosts ./exec
```

Where `np` specifies the number of cores per node, `hostfile` is a file containing a list of participating node IP addresses, followed by an executable.

Although this method works, it is far from ideal as discussed in Section 3.5. Since a resource manager, Torque, was installed, OpenMPI needs to be prepared for such a system. Instead of installing it with a package manager, it was compiled from source (version 1.10.1) with options to gain compatibility with Torque.

4.6.3 Cluster Synchronization

Every Odroid-C1+ board needs a MicroSD card as its own, tiny ‘hard drive’ on which the operating system and various software packages are installed. The image of the

MicroSD card can be copied to the MicroSD cards of other nodes. When booting for the first time, the hostname of the node will be changed to `node-<MAC address>`, where `<MAC address>` is the unique MAC address of a node minus any colons; see Listing 4.1 for examples. Thus when building the cluster or when adding a new node, the node's MicroSD card has to be flashed with the latest image. For this purpose, the mentioned process is suitable. For keeping the cluster updated however, it is not. Installing new software would require that all MicroSD cards are rewritten, which is not feasible.

In order to keep the software of all compute nodes up to date, a 'base node' is elected on boot, based on assigned IP address (see Section 4.5). The other nodes then perform an incremental synchronization with it, using the tool `rsync`. By using this system, all software is mirrored to the other nodes. After the synchronization process, a quick reboot is necessary on the other nodes in order to start or end any services or processes that were added or removed.

Specifically, this means that all software *should* be installed on the base node. All software, whether installed by a package manager such as Apitude or not, will be synchronized though the cluster after a quick reboot. This also implies that any files or software packages that are present on another node but not on the base node, will be removed.

4.6.4 Network File System

A network file system (NFS) is a necessity because every compute node needs to have access to the executable and other files for jobs. To achieve this, the `cluster` user's home directory `/home/cluster` on the monitoring host is exported to the 192.168.1.0/24 subnet. Every compute node can then mount the directory in the same but local location.

4.7 Monitoring

Suppose a user or admin wants to consult the state of the cluster. He could use the `pbsnodes` command, which queries the PBS server and outputs some information regarding the state of the compute nodes. It is possible to get some indication about the health of a single node or the cluster itself, but this would be far too cumbersome. Instead, a dedicated monitoring software package is needed. Furthermore, a graphical representation of the data through graphs and plots would ease the interpretation of the data.

4.7.1 Monitoring Host

The monitoring host is a simple laptop running a Linux distribution; in this case, Linux Mint Rosa 17.3. This machine has several functionalities. It coordinates and monitors the Odroid cluster and makes job submission possible. This way, the core functionality of the Odroid boards remains the same for all of them, that is, computations. Most importantly however, the monitoring host can be used, as its name implies, to monitor the cluster.

4.7.2 Monitoring Tools

Several monitoring packages are available on the web. Ganglia [72] was the first one encountered, which is a system monitoring tool that is scalable and distributed and intended for high-performance computing systems like clusters and grids. Despite this package doing everything we need, other tools can perhaps add to this functionality or improve it. Collectd [73] is a tool that can monitor on par with Ganglia. However, collectd only outputs RRD files; creating graphs is not included. Cacti [74] is another package similar to Ganglia, but seems less mature. Ganglia has a wide use base and is used all over the world; Cacti, not so much. Nagios [75] is yet another package that provides monitoring. Not only cluster are supported, but also e.g. generic servers, services, switches etc. Ganglia again seems to be more suited for its task in our context: cluster monitoring, which is why Ganglia was ultimately chosen.

Ganglia consists of two main daemons: `gmond` and `gmetad`. The Ganglia monitoring daemon, `gmond`, is a lightweight process that needs to be installed on every machine that needs monitoring. It monitors the system resources and sends statistics by default to multicast address 239.2.11.71. It also receives messages from other nodes, making them accessible for polling by `gmetad`, the Ganglia meta daemon. `gmetad` is another service that collects and aggregates from `gmond` or other `gmetad` sources. It stores their data in indexed round-robin databases by using RRDtool [76].

In our cluster, the `gmond` daemon is installed on every compute node and the monitoring host. Instead of multicast, unicast is used. The `gmond` instances on the compute nodes are configured to only send data by UDP to the cluster head node, the monitoring host. On the monitoring host, `gmond` was configured not to send data, but to receive data from compute nodes. Next, `gmetad` needs to be installed only on the head node, which then polls the local `gmond` instance and saves the data.

Another tool that is provided is `gmetric`, a commandline application. It is provided to support custom made metrics. This tool was deemed unnecessary, as Ganglia already collects many useful metrics by default. In particular the most important metrics captured by `gmond` are, among others, the following metrics:

- System load: average in minutes
  load_one, load_five, load_fifteen
- CPU usage: state percentage by type
  cpu_user, cpu_system, cpu_idle, cpu_nice, cpu_wio

- Memory usage: per type
  mem_free, mem_shared, mem_buffers, mem_cached, mem_total
- Network traffic: octets and packets
  bytes_in, bytes_out, pkts_in, pkts_out
- Process information: running and total number of processes
  proc_run, proc_total

Up until now, all metrics are collected and saved on the monitoring host. They still need to be visualized in order to be informative. Ganglia provides a web front-end, based on PHP and graphs made by RRDtool. Section 4.8.3 will discuss the features of the front-end in more detail.

4.8 The Cluster Front-End

The previous sections dealt with the underlying structure and software components of the system. This section however, will discuss the components that were provided as front-end for user access and monitoring.

4.8.1 Basic Job Submission and Compilation

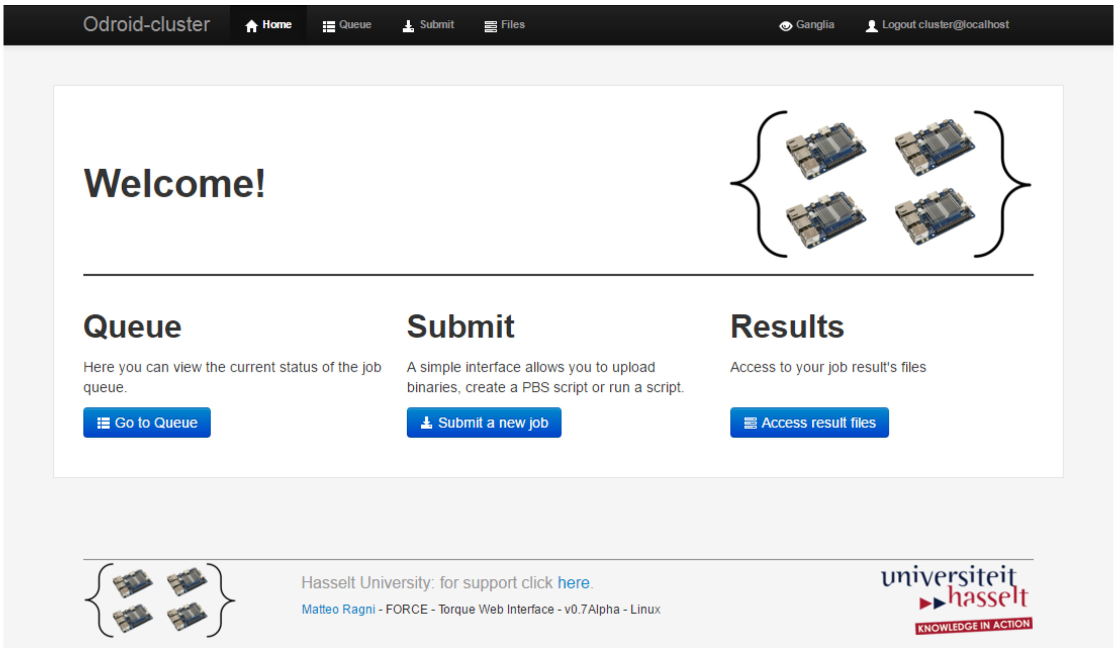
The job submission processes described here are assumed to take place in an SSH session or within a console on the monitoring host. Jobs specified by a PBS script can be submitted though the `qsub` command. However, due to the difference in architectures of the monitoring host and compute nodes, local compilation of code is not an option as the resulting binary would be incompatible with the target architecture. Instead, cross compilation can be deployed. However, the cross compilation setup is a hassle and has no hard success guarantees. In addition, not utilizing our own system would be somewhat wasteful. Thus, rather than cross compiling on the monitoring host, users can simply launch a job in order to compile their code natively on a compute node, matching the architecture for all boards. More even, they can launch an interactive job so that their code can be compiled interactively, as if they were in an SSH session with a compute node.

However, for less console-savvy users, a different approach is needed. As a solution, a web interface is provided through which users can create, submit and view jobs as discussed in Section 4.8.2.

4.8.2 Web-Based Job Submission

As mentioned in the previous section, a more user-friendly interface for novice users should be available instead of plain SSH. Force [77] is such an interface and is web-based. It allows the user to upload files or scripts, create PBS scripts with a 'wizard' and submit them as jobs. All files in the user's home directory can be viewed, deleted or downloaded if necessary. Lastly, the current status of the job queue can be requested, including detailed information for each of the user's jobs.

The project itself is described as being in alpha and under development, despite the last addition being made in June 2013. Nonetheless, it provided a good start, although plenty of bug fixes were required before actual usage. Furthermore, several components were customized in order to provide better functionality and looks. The resulting main page can be seen in Figure 4.8.



Welcome!

Queue

Here you can view the current status of the job queue.

Submit

A simple interface allows you to upload binaries, create a PBS script or run a script.

Results

Access to your job result's files

Hasselt University: for support click here.

Matteo Ragni - FORCE - Torque Web Interface - v0.7Alpha - Linux

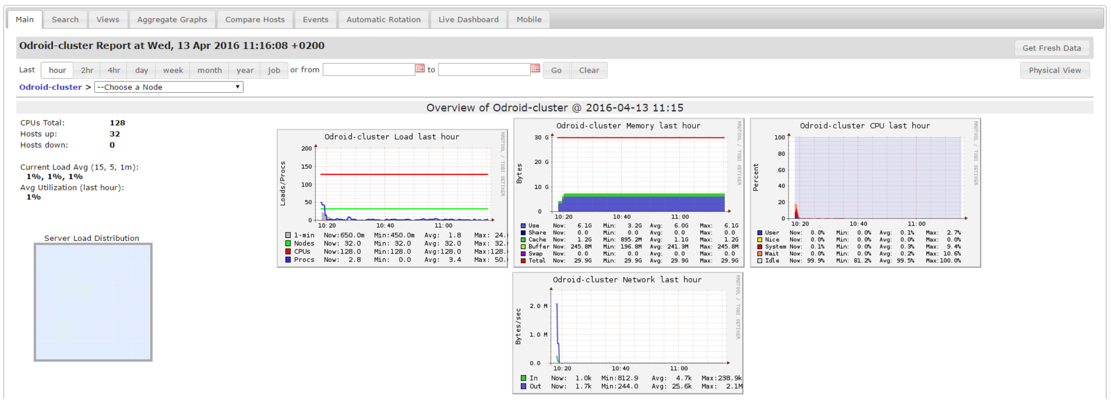
FIGURE 4.8: Customized main page of Force

Underlying, Force operates by directly querying Torque. This is done through an SSH session in the PHP back-end (libssh2-php) to localhost, since the interface is hosted on the same device as Torque, the monitoring host. This requires the user to log in first on the website, after which his credentials are used to open the SSH connection. For example, when viewing the queue, this means that the command `qstat -x` is executed through SSH, which returns information about the queue as an XML representation generated by Torque. The XML tree is then parsed in the PHP back-end and shown in a different representation to the user.

Note that since the user has to log in, his SSH credentials are required. Hence the website was made accessible by the HTTPS protocol only, although with a self-signed certificate. Another remark is that only the jobs owned by the logged in user are currently shown on the queue page, according to the default behavior of `qstat`. With the command `qmgr -c 'set server query_other_jobs = 1'`, to be executed as root on cluster-monitor, Torque can be instructed to show all jobs in the queue for all users.

4.8.3 Monitoring

Since Ganglia is being used to monitor the cluster, its web front-end is ideal for visualizing the captured metrics. It is installed and hosted on the monitoring host, as it needs to access the data collected by `gmetad`. In Figure 4.9, a part of the main page of the front-end is shown. By default, the load, memory, CPU and network metric aggregates of the entire cluster are shown at the top of the page. The user can also view the graphs for a specified metric of all nodes. Next, it is possible to look at statistics of the entire cluster, or of just one or more nodes. Furthermore, Ganglia provides graphs of the last hour, 2 hours, 4 hours, day, week, month and year. Custom time ranges can also be specified.



Odroid-cluster Report at Wed, 13 Apr 2016 11:16:08 +0200

Overview of Odroid-cluster @ 2016-04-13 11:15

```
CPUs Total:     128
Hosts up:       32
Hosts down:     0

Current Load Avg (15, 5, 1m):
  1%, 1%, 1%
Avg Utilization (last hour):
  1%
```

```
1-min   Now:650.0m  Min:450.0m  Avg:  1.8   Max: 24.
Nodes   Now: 32.0   Min: 32.0   Avg: 32.0   Max: 32.
CPUs    Now:128.0   Min:128.0   Avg:128.0   Max:128.
Procs   Now:  2.8   Min:  0.0   Avg:  3.4   Max: 50.
```

```
Use    Now:   6.1G  Min:   3.2G  Avg:   6.0G  Max:   6.1G
Share  Now:   0.0   Min:   0.0   Avg:   0.0   Max:   0.0
Cache  Now:   1.2G  Min: 895.2M  Avg:   1.1G  Max:   1.2G
Buffer Now: 245.8M  Min: 196.8M  Avg: 241.3M  Max: 245.8M
Swap   Now:   0.0   Min:   0.0   Avg:   0.0   Max:   0.0
Total  Now:  29.9G  Min:  29.9G  Avg:  29.9G  Max:  29.9G
```

```
User   Now:  0.0%  Min:  0.0%  Avg:  0.1%  Max:  2.7%
Nice   Now:  0.0%  Min:  0.0%  Avg:  0.0%  Max:  0.0%
System Now:  0.1%  Min:  0.0%  Avg:  0.3%  Max:  9.4%
Wait   Now:  0.0%  Min:  0.0%  Avg:  0.2%  Max: 10.6%
Idle   Now: 99.9%  Min: 81.2%  Avg: 99.5%  Max:100.0%
```

```
In    Now:  1.0k  Min:812.9  Avg:  4.7k  Max:238.9k
Out   Now:  1.7k  Min:244.0  Avg: 25.6k  Max:  2.1M
```

FIGURE 4.9: A part of Ganglia's web interface main page

Since Ganglia operates on cluster- or node-level, it is not possible to monitor the resources that are used for single job, since Ganglia has no knowledge of it. Although Ganlia can be filtered until only the statistics of a job are visible, it would be rather hard to do so for a novice user. Jobmonarch [78] is an addon for Ganglia and tries to overcome this problem. It gathers statistics on jobs and nodes and submits them to Ganglia. Through its own web interface, users can view statistics about the jobs that are currently running or in the queue. When finished, the statistics can optionally be archived. This is done through a Postgres SQL database for job statistics and RRD files for node statistics.

Although this addon sounded promising, it was ultimately not implemented. Since Force already supports viewing the queue on a web front-end, this information would be redundant. Also, since Jobmonarch's archiver stores data in a database and RRD files, data is essentially duplicated because Ganglia already does this. Lastly, installing and configuring the Jobmonarch software proved to be a challenge, due to a serious lack of documentation.

Instead of Jobmonarch, an own implementation was made as an alternative, dubbed "Job Lookup". It consists of a simple web interface (Figure 4.10) where users can enter the ID of the job they want to view runtime statistics of. Once submitted, the user is taken to Ganglia's front page. The job ID itself is obtained as usual, that is, by submitting a job through an SSH session or through the Force web interface.

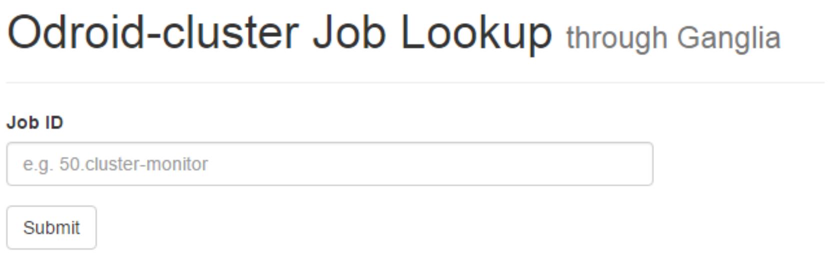


FIGURE 4.10: Job Lookup: a simple web interface to look up data of a job through Ganglia

By using Job Lookup, the data Ganglia displays gets automatically filtered for the user. Furthermore, all features of Ganglia are still available, since, for a given filter configuration, the user can still select and view any metrics available. Specifically, this means that the data that gets filtered in Ganglia corresponds with:

1. The date range: corresponds to the starting- and end times of the job.
2. The node list: all nodes that were used for the job. Ganglia accepts this as a regular expression, e.g. `node-1|node-2|node-3`.

Job Lookup operates through a web interface written in PHP. When jobs are completed, Torque adds extra information to the standard output file by applying the prologue and epilogue scripts, as mentioned in Section 4.6.1. The same scripts are used to store the job runtime timestamps and nodes used list in a file, written in a format acceptable for Ganglia. The file is named after the ID of the job. The PHP front-end reads the file with ID specified by the user and extracts the values. Ganglia, in turn, makes all settings accessible via GET parameters. Thus, the user is redirected to Ganglia's index page with the read values as GET parameters to filter the data.

4.8.4 A Multi-User System

Currently, the cluster is intended only for a single user. Multiple users could share the `cluster` account. However, this could introduce privacy and bookkeeping issues. Not every user or researcher wants his data to be visible to other users, and even if so, they would need to make agreements of which directory to use. This approach is in this case clearly not optimal.

To adapt the cluster, only few changes need to be made. First, on the monitoring host, other users besides the `cluster` cluster have to be created, each with its home directory in `/home`. Then, instead of only NFS exporting `/home/cluster` as described in Section 4.6.4, the whole `/home` directory should be exported instead.

4.9 Conclusion

This chapter described the hardware and software setup of the cluster and methods to improve usability. With the cluster properly set up, it is now a possibility to assess the performance of the cluster. In the next two chapters, the performance of the cluster will be evaluated through two real-world use cases. First, in Chapter 5, we will dive deeper in a problem from the field of statistics. Chapter 6 will in turn discuss the performance for the cluster through a physics problem case.

Chapter 5

UC 1: Measles Resurgence in a Highly Vaccinated Population

The first use case is situated in the field of statistics. A recent and relevant problem case was obtained through contact with Prof. dr. N. Hens and dr. S. Abrams. Before going over to the code and evaluation, this chapter will commence by briefly situating the use case in their research context.

5.1 Introduction

In the study of Hens et al. [79], the risk of measles resurgence is assessed for a highly vaccinated population with application to Belgium. For an age-specific group, serological data is required in order to determine a susceptibility profile. However, this is only a possibility if the data is collected recently. For Belgium there is no such recent data available. A standard transmission model therefore takes a lot of time to develop. Thus, another approach was needed since a far simpler alternative was desired. A multicohort model was developed, allowing non-recent serological data to be used. This data is combined with data on vaccination coverage, social contact data as well as immunological data obtained from systematic literature reviews.

For this problem, an implementation was made by the research group in the R programming language. The code was run on their own laptops and desktop computers, equipped with relatively recent Intel i5 or i7 processors. However, running the code is described as

> *"...taking a long time, for many hours consecutively."*

Re-running the code for different parameters is tedious – especially when deadlines have to be respected. The code itself was entirely sequential. While this indeed works and gives the desired results, it takes away a significant portion of time of the user's system. Additionally, it does not allow the code to perform additional iterations for better results due to the sheer time this problem takes to fully execute.

Furthermore, the code should be easy to execute for users other than the developers themselves. Since the code makes a prediction on how a measles outbreak will spread for a given year, this parameter and others should be entered in a user-friendly manner before execution. This would allow the code to be easily run on a more complex system than a typical user system, lowering the usage threshold.

The remainder of this chapter is organized as follows. First, the existing code is thoroughly investigated. Currently, the multi-core advantage on execution speed of the given i5 or i7 processors is unfortunately not gained. To do so and even more so when using parallel systems such as clusters, a parallel version of the code will be made. Next, when the given code is able to run on a cluster, novice computer cluster users should still be able to run the code with different parameters. Finally, the performance of the code and our cluster is compared with a system on which the code is currently run: a laptop.

5.2 Original Code

The code that was initially received can perhaps benefit from parallelization to improve execution time. Parallelization is not trivial and cannot be immediately applied on sight. First, the code should be thoroughly examined. Redundant parts should be removed, other parts can perhaps be optimized. Most importantly however, the main computationally intensive regions in the code should be identified as they can perhaps prove to be candidates for parallelization. The original code consists of several R source files. The main file, MeaslesBelgium.R, makes use of all other files. Eight main steps are executed after each other:

1. Read in the Belgian serological data from 2006
2. Estimation of the proportionality factor q in 2013
3. Generalized Additive Model (GAM)
4. Susceptibility per municipality in 2013: bootstrap approach
5. Susceptibility plots for Hasselt, Liege and Brussels
6. Estimation of the effective reproduction number R in 2013
7. Estimation of the effective reproduction number R during holiday period in 2013
8. Evolution of the effective reproduction number R between 2006 and 2015

The first three listed steps can be considered as preprocessing. Data is read in, transformed and prepared in steps 1 and 2. In step 3, the serological data is used to determine a susceptibility profile for the year 2006. This means that for 2006, the susceptibility probabilities per age group are calculated. This is done by using a generalized additive model (GAM). A GAM is a flexible model, which involves the determination of several parameters in order to estimate the susceptibility probabilities. The model has to be fit to the data, which is done through penalized maximum likelihood estimation. The fitting can be intensive based on the basis functions that are supplied, in our case, 3^{rd} degree functions. Five of such fittings are performed, of which numbers 2 and 3 in particular are especially demanding operations.

Now that a susceptibility profile has been determined for the year 2006, step 4 involves determining susceptibility profiles for years after 2006. To do so, since there is no serological data available as of 2006, vaccination coverage data from 2013 and waning data (vaccine failure data) are combined. To take uncertainty in account and to enable the calculation of a 95% confidence interval for the effective reproduction number, this process is repeated 500 times for slightly different samples. This is a parametric bootstrap process, which is an intensive part of the code as in every iteration another GAM is fitted and another loop is present for each municipality in Belgium (593). Steps 1 to 4 are illustrated in Figure 5.1. In the code, Evolution.R handles the bootstrap process up to 2013, after which no vaccination coverage data is available. Future_Prediction.R takes over from here on: the coverage data is extrapolated towards years after 2013.

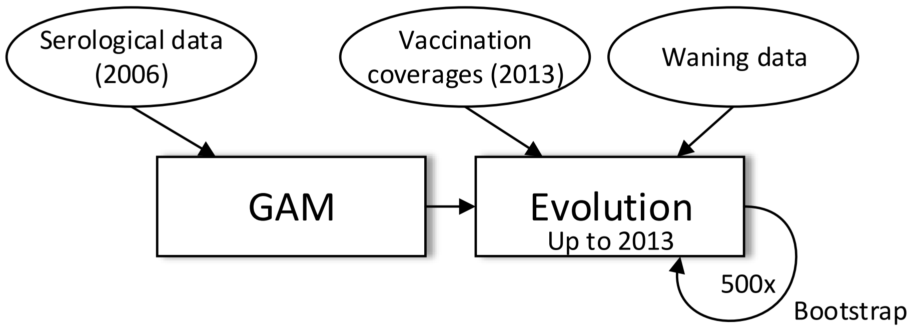


Figure 5.1: Code structure overview (first 4 steps). Evolution can be replaced by Future_Prediction for years >2013

Steps 5, 6 and 7 are essentially postprocessing steps, where plots and maps are drawn. Lastly, in step 8, the same bootstrap process is applied to different years (since step 4 only does this for 2013), resulting in another intensive step.

The initial code can be reduced: steps 1-3 always need to be present since they prepare the data. Steps 5-7 however, could be scrapped. The data itself is outputted to files anyway; postprocessing should not take place on a cluster per se. However, for user friendliness reasons, some of these steps were kept and are further discussed in Section 5.7. More interesting is step 4. Here, the bootstrap process is mainly a large loop, consisting of 500 iterations as defined by parameter `B` and must be further investigated. `B` itself was determined ad-hoc by the researchers and could in theory have a larger value, but due to large execution times it was not practical to choose a greater number. Step 8 is essentially a loop around the bootstrap process for the years 2006-2012 and a future prediction which works completely analogously for the years 2030-2040. In the end, three intensive regions have been identified and are possibly candidates for parallelization:

1. GAM fitting in step 3;
2. Parametric bootstrap approach in step 4;
3. Step 8, which is step 4 looped for different years.

Since step 8 is completely analogous to step 4, it is superfluous and can be scrapped if the code accepts a year-of-interest parameter. For this reason, only points 1 and 2 will be of further consideration in the remaining sections.

5.3 Parallel Computations in R

5.3.1 Packages and Options

In the R programming language, parallel computing can be done through various packages. Notable examples are foreach, multicore, snow, snowfall, parallel and Rmpi. Package parallel is part of the R base packages and includes snow and multicore. Multicore is used to take advantage of multiple cores in shared memory machines. Snow is the de facto standard for clustering and will be used as such. Snowfall is a wrapper for the snow package, in order to simplify the latter. The wrapper is not necessarily needed since snow is not overly complicated and is an abstraction itself.

Note that working with user-defined threads in R is not trivial, if not impossible by default since there are no API functions defined to do so [80]. Libraries and extensions to R must fall back to the C/C++ or Fortran programming languages, where OpenMP or POSIX threads and equivalents can be used. Moreover, R itself is not thread-safe by default. According to a quote from the R extension manual [80], using threads is indeed far from ideal:

> *"Calling any of the R API from threaded code is 'for experts only': they will need to read the source code to determine if it is thread-safe."*

5.3.2 Basic snow Functionality

The snow library is intended for multi-core, distributed memory clusters. It operates through a scatter/gather paradigm. Of all the processes that are spawned, one is chosen as the master while the rest are called workers or slaves. Thus, its modus operandi is the following:

1. The master breaks down the computations to be done in chunks and divides ("scatters") them over the slaves.
2. The slaves do their work and send the results back to the master.
3. The master then gathers all those results and aggregates them to solve the problem.

Snow consists of a few important calls, namely `makeCluster`, which creates the cluster, i.e. spawns slave processes; `clusterExport` which is used to send variables that need to be shared to the slave processes. Lastly, the scatter/gather paradigm is implemented in the `clusterApply` function. An example can clarify: in Listing 5.1, an MPI cluster is created. Local variables a and b are declared, initialized and exported to the workers. Then, a certain function is executed 100 times, with the x parameter of the function being the iteration number that gets automatically passed. The return values of each function are sent back to the master, which aggregates them into the results variable.

```
library(snow)

cl <- makeMPIcluster()
cat("MPI cluster has", length(cl), "slaves\n")
a <- 1
b <- 2
clusterExport(cl, c("a", "b"))
results <- clusterApply(cl, 1:100, function(x) {...})
```

Listing 5.1: Basic snow example code

Note that `clusterApply` operates in a round-robin fashion. If the number of tasks (e.g. 5) is higher than the number of processes P (e.g. 2), then P_1 gets assigned chunk 1, P_2 gets chunk 2, which then loops as P_1 again gets chunk 3. We can consider this to be a somewhat static version of scheduling. Another function exists however, namely `clusterApplyLB`, which is a load balancing version (dynamic scheduling) of `clusterApply`. When a job completes, the next job is placed on the available node as soon as possible; this continues until all jobs are complete. Using `clusterApplyLB` can result in better cluster utilization than using `clusterApply`, but may degrade performance due to the increased amount of communication that is required.

5.4 Parallelization of the Code

5.4.1 Method

The 500-iteration bootstrap loop proves to be an ideal candidate for parallelization, as every iteration can execute independently of each other, making this subproblem embarrassingly parallel. To use the advantage of our cluster, MPI should be used by means of snow. Since the code is quite complex, caution is advised. Snow cannot be simply implemented without verification, thus we propose the following scheme:

1. Implement parallelism using snow;
2. Verify that the output is still correct by setting the same random seed and by comparing the output files between test runs;
3. Refine the implementation by going back to step 1 if necessary.

5.4.2 Parallel Bootstrap Implementation

In general, the scheme of Listing 5.1 can be applied onto the bootstrap process. However, just calling `clusterApply` is not sufficient. The slaves also need to load in any packages that the code requires. To make things a bit more generic, an implementation of a `foreach` function was made that loads any required packages on all slave processes first and then calls `clusterApply` or `clusterApplyLB`, a load balancing variant, shown in Listing 5.2. For the original code, this means that the bootstrap process loop can now be called with the `foreach` function, where `X` is the number of iterations, `fun` is a function (the bootstrap code) and `"..."` represents any additional parameters that should be passed to `fun`.

```
foreach<-function(X, fun, Packages=NULL, loadBalance=TRUE, ...) {

  #load packages on every slave process only once
  if (!is.null(Packages)) {
    clusterApply(cl, seq(along=cl), function(x, libs) {
       sapply(libs, function(x) library(x, character.only = TRUE,
  ↪ quietly = TRUE))
    }, Packages)
  }

  #load balancing or static scheduling
  if(loadBalance) {
    out <- clusterApplyLB(cl, X, fun, ...)
  } else {
    out <- clusterApply(cl, X, fun, ...)
  }
}
```

Listing 5.2: Parallel foreach implementation in R using snow

As a start during development, *all* variables were exported to the slaves to guarantee correctness of the output. After internal testing, the output was indeed still correct – but for B = 3, the execution time increased tenfold compared to the sequential version.

Surely something was very wrong. After investigating, the `clusterExport` functions apparently took many minutes to complete. This was caused by the serialization of the variables, which can be a serious bottleneck [81]. Instead of exporting all variables, only the variables that are needed in the bootstrap process were exported, reducing the overhead to an order of seconds.

5.4.3 GAM Fitting Parallelization

The `gam` function of package mgcv is used for the GAM fittings. Although this function cannot be parallelized with MPI, it supports multithreading [82]. It can use threads on a shared memory multi-core machine via OpenMP. Doing so is made very easy as it has a built-in function for multithreading. The desired number of threads can be specified by setting `nthreads` in the `control` argument of `gam`.

The times of these function were verified and indeed, a speedup took place, although not as great as hoped. This can be explained through the internal workings of the `gam` function: only the dominant $O(np^2)$ steps are parallelized, where n is number of data and p number of parameters. Nonetheless, for four threads, the execution times were halved, which still is a significant improvement.

5.4.4 Proposed Scheme

The proposed parallelization strategies are summarized in Figure 5.2. The master process is first launched on a single core. Since every compute node in the cluster has four cores, 4 threads can be spawned on the master node to calculate the GAM fitting. Then in the bootstrap part of the code, the entire cluster is used by means of MPI. Running the code can be done through the statement

```
mpirun ./RMPISNOW CMD BATCH ./MeaslesBelgium.R
```

The RMPISNOW file is available in the installation folder of the snow package and allocates the MPI processes on the nodes. Upon running this statement, the entire cluster gets immediately launched. The `makeMPIcluster` function does not actually create the cluster, it just obtains a reference to the cluster. This means that while the master process is doing the preprocessing work, *all* other processes will be waiting as they have all already been created. This might seem not that bad at first. However, every MPI process keeps its CPU pegged at 100% usage. For the sake of performance, busy-waiting is done on blocking operations. Since it is assumed that only the MPI process is of importance and since the task is waiting for communications, it continually polls for that communication to reduce latency.

This behavior should be avoided not only to spare system resources, but also to ensure multithreading is actually useful in the GAM fitting part of the code. Otherwise, the

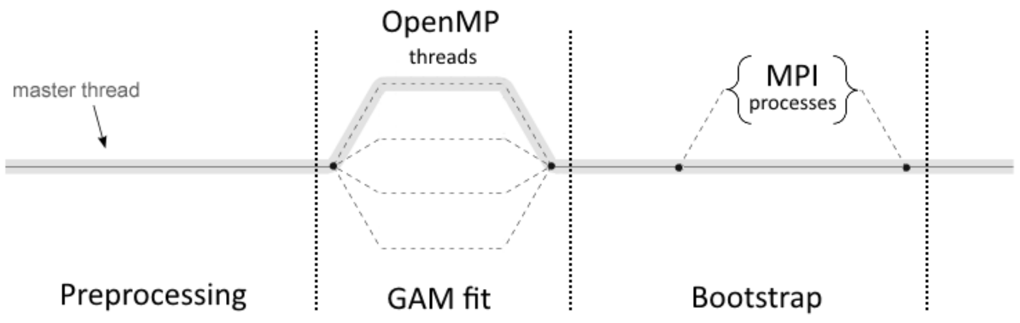


FIGURE 5.2: Proposed parallelization scheme for MeaslesBelgium

'master' node has 1 core reserved for the master process and three cores doing busy-waiting, diminishing the effect of OpenMP. Therefore, the code is changed yet again. The cluster is created not at the start of the program, but just before the bootstrap process explicitly through `makeMPIcluster`. All cores remain truly idle this way until this part is reached. The program must now be launched by

```
R CMD BATCH MeaslesBelgium.R
```

Since `mpirun` is no longer present, `makeMPIcluster` must be explicitly fed with the number of slave processes. Originally, this was handled internally due to the presence of the `mpirun` command.

5.5 Parallel Implementation Remarks

Although the scheme designed in the previous section seemed successful on a 4-core machine at first, issues were encountered when scaling up to more cores and more slave processes. This section discusses these problems and their proposed solutions.

5.5.1 Running the Program

The last section concluded that the code should not be run with `mpirun` so that the idle cores could still be useful for multithreading. However, when scaling up to e.g. four nodes, the spawned processes were mismatched on the nodes. Since every node has 4 cores, a maximum of 4 processes are spawned per node. Although this is correct, the `makeMPIcluster` function starts spawning on the node where also the master process resides, followed by the other nodes in groups of 4 processes. In case of 15 slaves, one core is left unused on the last node while there is an abundance of slaves on the master node, as summarized in Table 5.1. As a result, five processes each compete for maximal CPU usage on the master node, causing many context switches and heavily impacting performance. The last node in turn could have worked harder, but lacks one process to do so.

	node1	**node2**	**node3**	**node4**
Expected	M S S S	S S S S	S S S S	S S S S
Actual	M S S S S	S S S S	S S S S	S S S

TABLE 5.1: Node - process mismatch when dynamically spawning MPI processes. M and S are the master and slave processes respectively

As a solution, the `mpirun` command has to be used again since Torque and MPI have adequate knowledge of the cluster. To do so without hindering the preprocessing, the code has to be split into two parts. The code now has to be run with both statements:

```
R CMD BATCH MeaslesBelgium.R
mpirun ./RMPISNOW CMD BATCH ./Evolution.R
```

Where MeaslesBelgium.R now exclusively contains the preprocessing and GAM fit parts of the program and Evolution.R contains the bootstrap process. For this to work, Evolution.R still needs access to the variables calculated during the preprocessing. To accomplish this, the R workspace containing all variables of the first part is written to a file and loaded again in the second part. Since this is just a binary dump of the data, the impact on performance is minimal (< 25ms).

5.5.2 The clusterExport Function

In Section 5.4.2, the `clusterExport` function was discussed. Exporting only the required variables to the slave processes resulted in a time usage in the magnitude of seconds – for a 4-core machine. Although this seemed positive at first, when scaling up to more slave processes, performance issues were encountered. The `clusterExport` function scales with the total number of slaves, in the negative sense. Doubling the number of processes effectively doubles the elapsed time for the function to finish. For half of the cluster, a total of 127 slaves needs to be addressed. The time here to export the variables easily exceeded the 15-minute mark, heavily impacting the total execution time. The original code is shown in Listing 5.3 for reference.

```
clusterExport(cl, varlistGlobals, envir=.GlobalEnv)
clusterExport(cl, varlistLocals, envir=environment())
```

LISTING 5.3: Original bootstrap variables distribution

Since the `clusterExport` function severely degrades the total performance, another way of data distribution had to be found. The answer lay in the way R handles its data. Workspaces can be easily written (see previous section) and the same can be applied to a set of variables. Listing 5.4 shows the final data distribution strategy. The variables are written to files, which is constant in time use. Then, they are read by all slave processes by means of the `clusterEvalQ` functions. For the 32-node cluster or 127 slaves, this results in around 90 seconds time use, a significant improvement. Note that the slaves have access to the files written by the master only due to the existing NFS infrastructure of the cluster. For the remainder of this chapter, exporting the data implies that the data is distributed according to the new, file-based method and not via the original function.

```
save(list=varlistGlobals, file="EvolutionWS.RData", compress=FALSE
    ↪ , envir=.GlobalEnv)
save(list=varlistLocals, file="Evolution2WS.RData", compress=FALSE
    ↪ , envir=environment())

clusterEvalQ(cl, load(file="EvolutionWS.RData"))
clusterEvalQ(cl, load(file="Evolution2WS.RData"))
```

Listing 5.4: Improved bootstrap variables distribution

Lastly, it is worth mentioning that *all* cluster functions from the snow library take increased time when scaling up the total number of processes, as they all scale inverse linearly with the size of the cluster. The `clusterExport` function however, is affected most drastically.

5.5.3 Load Balancing and Static Scheduling

Initially, no real difference was found on a 4-core machine between the `clusterApply` and `clusterApplyLB` functions, static round-robin scheduling and dynamic scheduling respectively. Increasing the number of tasks so that it is greater than the number of processes allows the scheduling methods to do their work. Yet again, no real difference was found between them. Only when scaling up to 127 slave processes, the effects between both functions were noticeable. The `clusterApply` function left some cores idle between tasks. This is according to the documentation of both functions, which specifies that `clusterApplyLB` can potentially leave the cores significantly less idle, which indeed was confirmed.

Since the behavior of both functions seems to be very dependent on the number of tasks and the size of the cluster, it is interesting to further investigate into them. Currently, no real conclusions can be made based on a few observations. Section 5.6 will assess the performance of the cluster and will therefore include benchmarks for both functions to adequately determine the differences in performance between them.

5.5.4 File I/O

During the bootstrap process, each slave writes its results to files. A total of 593 files are written, one for each municipality in Belgium. Each slave writes a single line of data to each of these files during an iteration of the bootstrap process. The problem is that all slaves write to these 593 files concurrently, resulting in race conditions and erroneous output.

As a remedy, these files should simply not be written. Instead of concurrently writing the municipality data, each slave process writes all data to its own unique file. A file is thus written for each iteration of the bootstrap process (B = 500). The original postprocessing code assumes to read in the municipality files, which do not exist now. Ideally, the code should be adapted to the new format, but currently the files are rewritten on disk to the target format in an additional step during the postprocessing steps.

5.6 Performance

To assess the performance of the cluster, a comparison will be made with a machine that is similar to the devices of the end users. The 'benchmarking computer' contains a 6-core Intel Xeon E5-2630 v2 processor clocked at 2.60GHz. Two cores are disabled in the BIOS to better mimic the user systems that consist of four cores currently. Note that this CPU is probably still more powerful than their current systems, since the code is very often run on notebooks.

First, we should gain a better understanding of the parallelization efforts. The timings of the original, purely sequential version of the code should be examined before a comparison with the parallel versions is in order. The program was run five times with the parameters B = 500, R_0 = 12 and D = 6, as for all benchmarks in this section. Figure 5.3 shows the execution results for the benchmarking PC.

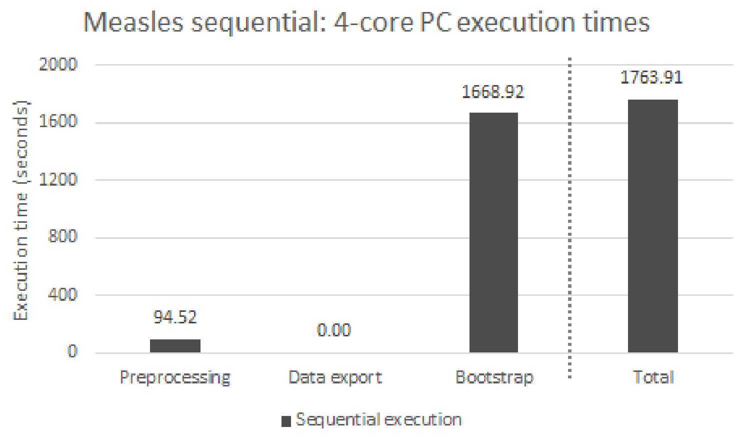


FIGURE 5.3: Measles: original fully sequential execution times. The total execution time of the program is shown on the right as a reference

The "preprocessing" bar encompasses the preprocessing up to and including the GAM fitting, since this is essentially a preprocessing step. The data export represents the time needed to distribute all variables needed by the MPI slave processes, which is of course zero since no exports need to be done as MPI is not used in this sequential version.

Finally, the loop of the bootstrap process is executed 500 times sequentially, as defined by B = 500. The standard deviation of the total execution time is 0.8 seconds, which is very constant.

Now, when looking at execution times of the parallel part of the code (after the GAM fitting) in Figure 5.4, we can determine that the parallelization was indeed effective. On the same benchmarking PC, the time for the 500 bootstrap iterations was reduced from 1668.92 to 878.23 seconds. This is for one core that plays the role of master and three worker cores. A speedup factor of 1.90 was reached. Note that the load balancing by the `clusterApplyLB` results in better performance, which is somewhat surprising given the low amount of slave processes. The standard deviation on the total execution time is again quite low, being 0.41 seconds.

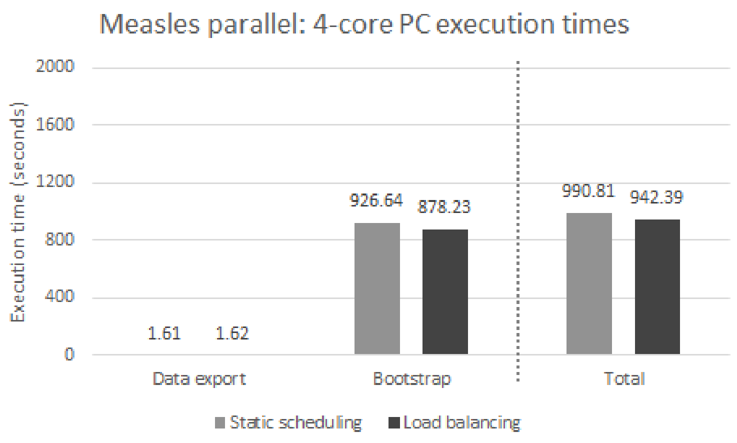


FIGURE 5.4: Measles: parallel execution time (4-core PC). The total execution time of the program is shown on the right as a reference

Next, it is interesting to make a comparison of the preprocessing part of the code between the cluster and the benchmarking PC. This part of the program is mostly single threaded when loading in and preparing data, followed by the now multithreaded GAM fitting functions. Since it can only be run on a single core initially and then on multiple cores within shared memory, these timings do not scale with cluster size but do slightly with the number of cores on a single machine. Figure 5.5 therefore essentially shows the execution times of a single Odroid board versus those of the PC. The PC has with its 57.93 seconds made improvements when compared with the fully sequential approach where it took 94.52 seconds (Figure 5.3) to complete. It takes the Odroid board a factor 13.90 times longer to complete the same part. This can mostly be attributed to the GAM functions that take significantly longer to complete. The performance per core of the Intel CPUs is clearly a lot better. What is also important is that the disk access times need to be considered. The Odroid board need to retrieve the data on an NFS drive that is a relatively slow hard disk drive on the monitoring host, while the PC features an SSD.

Now for the parallel part, the bootstrap process, was run on the cluster on 16, 32 and 64 nodes and thus for 64, 128 and 256 cores respectively. Each run was again done five

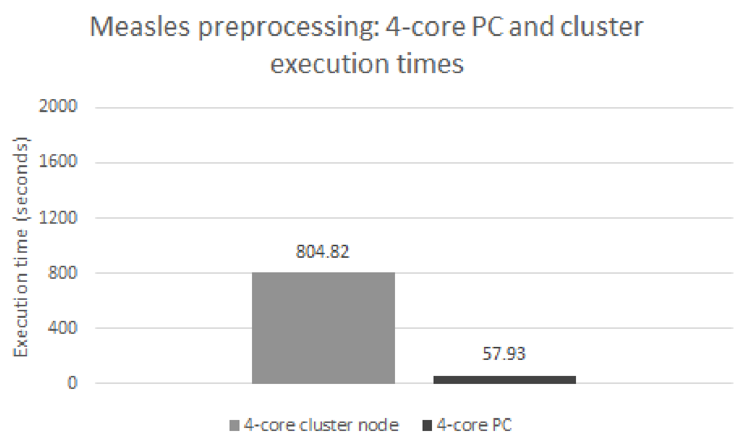


FIGURE 5.5: Measles: prepocessing execution time including GAM fitting

times and for static scheduling and load balancing. Figures 5.6, 5.7 and 5.8 illustrate the results. In the charts, the data export times listed scale linearly in the negative sense (inverse linearly) with the number of nodes or cores. This is the case for all cluster functions in general, as discussed earlier in Section 5.5.2. Doubling the number of processes doubles the time the functions take. The load balancing method seems to be always faster, although the difference with the static scheduling method gets smaller when scaling up. We can notice standard deviations of 21.24, 10.77 and 19.52 seconds respectively for the load balancing benchmarks. Clearly, this is a lot higher compared to the standard deviations of the 4-core PC. This is caused by the load balancing itself, which is a lot harder for these many cores. Do note that the 32-node version has a significantly lower standard deviation, as the load balancing seemed to have an easier time here.

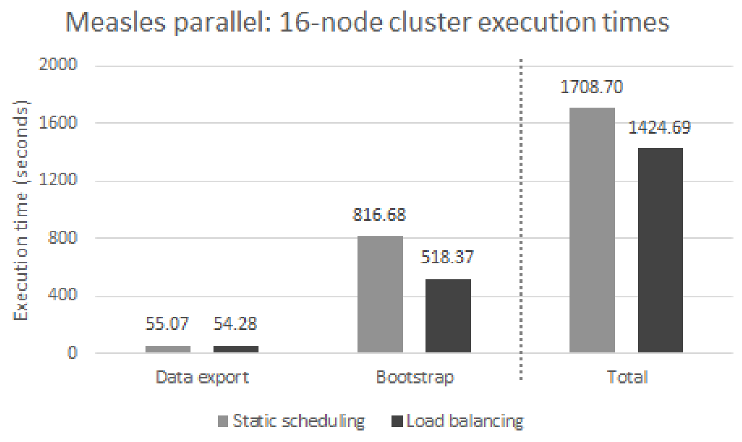


FIGURE 5.6: Measles: execution times of the data export and parallel bootstrap for a 16-node cluster. The total execution time of the program is shown on the right as a reference

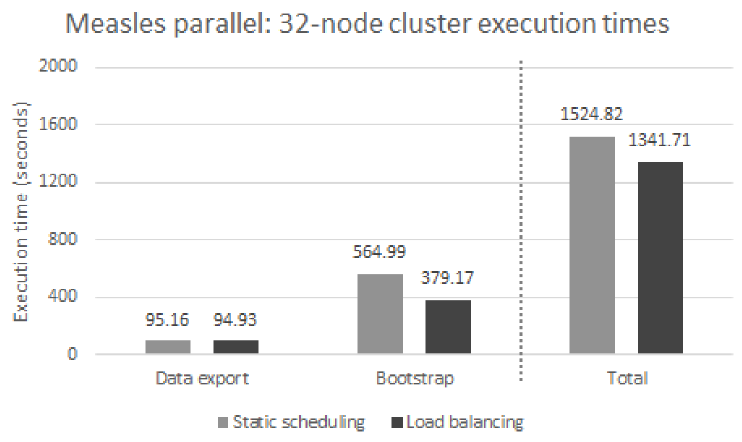


FIGURE 5.7: Measles: execution times of the data export and parallel bootstrap for a 32-node cluster. Performance is improved over the 16-node version. The total execution time of the program is shown on the right as a reference

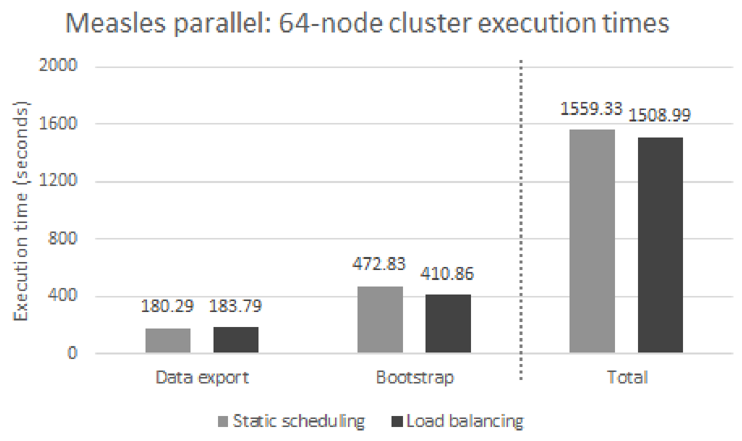


FIGURE 5.8: Measles: execution times of the data export and parallel bootstrap for a 64-node cluster. Performance is worse compared to the 32-node version. The total execution time of the program is shown on the right as a reference

What is most important is that the parallel bootstrap part performs better on the 32-node cluster than the 64-version. This can be attributed solely to the way the snow library handles the cluster functions. Therefore, it is of interest to determine the ideal number of nodes. The more workers there are, the faster work is done but the more overhead is present. A balance must be found. In Figure 5.9, the execution times for the parallel bootstrap are plotted by the number of nodes in the cluster. The most optimal number of nodes was determined to be 28.

The ideal number of nodes, 28, can be further confirmed by statistics captured in Ganglia. In Figure 5.10, the CPU usage is shown for the data export and parallel bootstrap for 28 (left) and 64 (right) nodes. As discussed earlier, the CPUs are always pegged at 100% usage due to MPI. The blue color shows user time. This means that when the usage plotted is mostly red, the CPUs are actually idle from the user perspective.

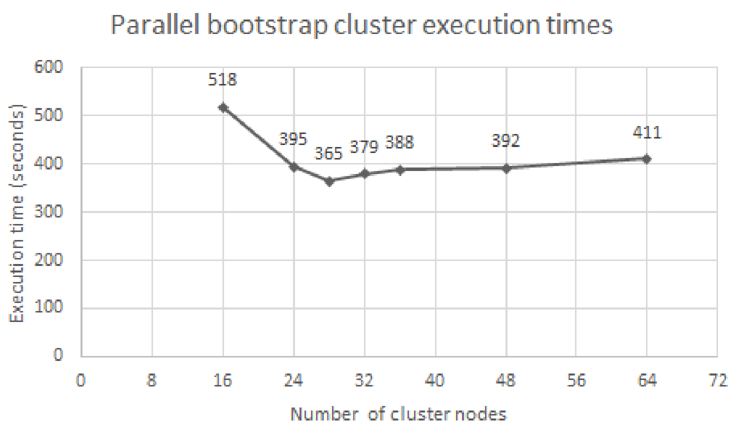
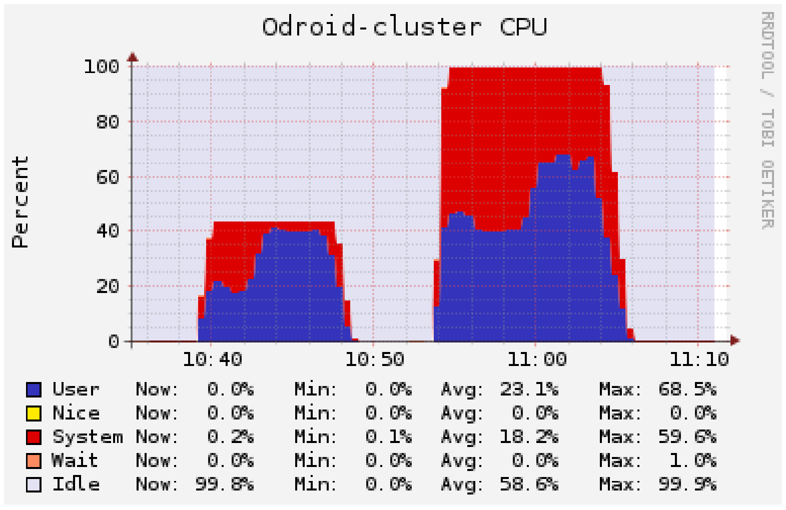


FIGURE 5.9: Measles: execution times of the parallel bootstrap for a varying number of cluster nodes



```
■ User    Now:  0.0%   Min:  0.0%   Avg: 23.1%   Max: 68.5%
■ Nice    Now:  0.0%   Min:  0.0%   Avg:  0.0%   Max:  0.0%
■ System  Now:  0.2%   Min:  0.1%   Avg: 18.2%   Max: 59.6%
■ Wait    Now:  0.0%   Min:  0.0%   Avg:  0.0%   Max:  1.0%
□ Idle    Now: 99.8%   Min:  0.0%   Avg: 58.6%   Max: 99.9%
```

FIGURE 5.10: Measles: CPU usage for the data export and bootstrap execution of the 28-node cluster (left) and the 64-node cluster (right)

Initially, the MPI processes are started but the master process still has some work to do. The cluster has to be 'started' and variables have to be exported, explaining the idleness of the cores. After a while, the parallel bootstrap part of the code is reached, causing the the user CPU usage to ramp up since the cores in the cluster are now no longer left completely idle. The vertical difference between the red system CPU usage and the blue user CPU usage is decreased until the job is finished. The difference between the 28- and 64-node version can be seen clearly. Not only does the 28-node version start earlier with the bootstrap part due to a reduced overhead, it also has a much lower system CPU usage during the bootstrap compared to the 64-node version. This can be attributed to the `clusterApplyLB` function, since the lower number of processes makes it easier to effectively schedule tasks. Although the 64 nodes are essentially faster when purely looking at the 500 iterations of the bootstrap, the overhead makes the 28 nodes a better solution.

Finally, considering the execution times of the PC and the cluster, what if we take the best of both worlds? Clearly, the single core performance of the Intel Xeon processor outshines a single Odroid board during the preprocessing, while the parallel bootstrap clearly benefits from the cluster. Figure 5.11 shows the combined times, further reducing the total time to just 562 seconds. The result is significantly faster compared with the initial sequential approach (Figure 5.3). Do note that the pure parallel part, the bootstrap process, consists of 500 iterations. Increasing the number of iterations to e.g. 5000 would only increase the advantage of the cluster. In fact, increasing B to a such a high number is now a possibility, which it was not before due to the long execution time it would take.

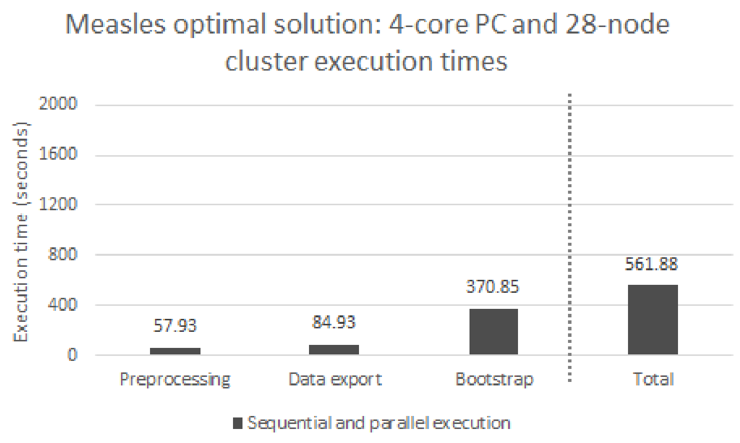


FIGURE 5.11: Measles: optimal execution times by combining both PC- and cluster approaches. The total execution time of the program is shown on the right as a reference

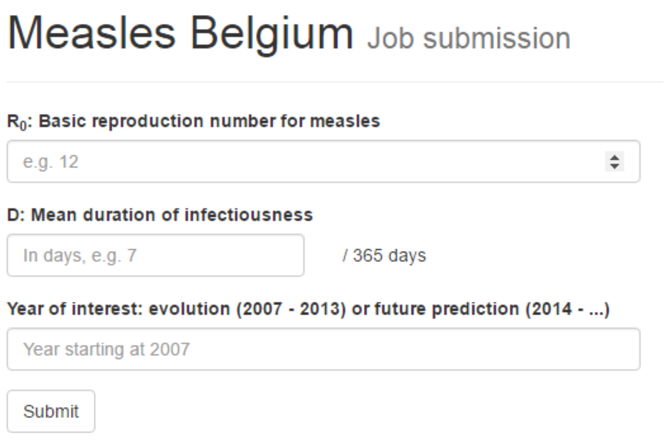
Finally, if only the year of interest parameter changes, the preprocessing could in principle be skipped after the first time, making it possible to do all computations on the cluster instead of the combined approach.

5.7 User Friendliness

As discussed in Section 4.8, jobs can be submitted through an SSH session or the Force web interface. Force already proves to be more user friendly than just an SSH session, but still requires some knowledge of the underlying system and of PBS scripting. Consequently, a threshold regarding the usage of the cluster can still be present for users that are used to writing and running code on their own machine. In an effort to lower this threshold and to support easy and user-friendly reruns of the code with different parameters for users other than the programmers, a user interface must be made available for this use case, hiding any technical details of the cluster. Only use case-related parameters should be entered by users. The following parameters have been identified:

- R_0: basic reproduction number for measles to determine the q-parameter;

- D: mean duration of infectiousness in days;
- Year of interest for evolution (2007-2013) or future prediction (>2013).



Measles Belgium Job submission

R_0: Basic reproduction number for measles

e.g. 12

D: Mean duration of infectiousness

In days, e.g. 7 / 365 days

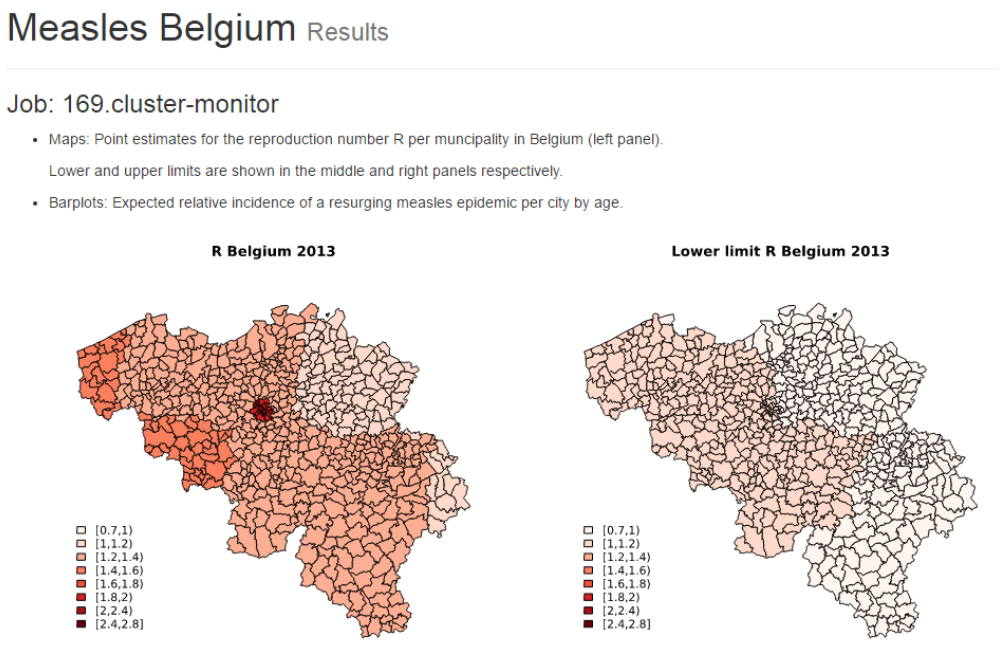
Year of interest: evolution (2007 - 2013) or future prediction (2014 - ...)

Year starting at 2007

Submit

FIGURE 5.12: Job submission interface for Measles Belgium

The web interface (Figure 5.12) works through a PHP back-end. It is integrated into the Force framework, which required additional rewriting of the code to support such situations. A custom PBS script is created with the user-specific parameters upon job submission, which is then passed to Torque. The job ID will be presented to the user upon a successful submission.



Measles Belgium Results

Job: 169.cluster-monitor

- Maps: Point estimates for the reproduction number R per muncipality in Belgium (left panel).
  Lower and upper limits are shown in the middle and right panels respectively.
- Barplots: Expected relative incidence of a resurging measles epidemic per city by age.

FIGURE 5.13: Part of the job result interface for Measles Belgium

Of course, so far only the input for the cluster has been made user-friendly. With regard to the output, the result files can be accessed through Force as usual once the job has been completed. However, the program can also output graphs and plots after postprocessing steps. To present these graphical data items to the user, another web interface is deployed. The job ID that was initially received can be used to view those results similar to the Job Lookup system shown before in Figure 4.10. Once the ID is entered, the user can view the graphical results at a glance, making it easy to compare the results between runs without the need to download and aggregate the output files first.

5.8 Conclusion and Lessons Learned

First and foremost, knowledge and best practices gained during the parallelization process have been passed to the original developers, Prof. dr. N. Hens and dr. S. Abrams. In this chapter, it was shown that even with small effort, parts of the code can be sped up and that it is worth investigating. This was the case for the `gam` function, that only required an additional parameter to effectively halve its execution time. Other parts require a greater deal of effort, that is, the bootstrap process, but is definitely worth the time. As initially said, the parameter B = 500 was determined ad-hoc. Larger values for B perhaps give better, more accurate results. Due to the sheer time the code originally took to run, the researchers settled for a value of 500. Now, with the help of the cluster, running the code with larger B values is perfectly possible within an acceptable time frame, directly benefiting their research.

The parallelization in R itself is not always as straightforward as it may seem. If the standard approach for snow was followed, the performance would be a lot worse than it is now. Another thing is that snow's functions scale very poorly, inverse linearly, with the size of the cluster. Instead, the Rmpi package should be used, reducing the overhead but at the same time greatly increasing the complexity of the code. For programmers with a non-IT background this may be a daunting task due to the increase in complexity and longer development time when compared to the snow package. Instead, snow could or should still be used, although its disadvantages must be kept in mind and be avoided.

Looking at the performance of the cluster, it is clear that the cluster benefits the most from the truly parallel region, that is, the bootstrap process. However, the mostly single threaded preprocessing diminishes the time gained through parallelization although increasing the parallel part increases the overall benefit. In the end, a combination between a desktop computer or notebook and the cluster could be used, with only the parallel part done on the latter for the best overall performance. Moreover, since the optimum number of nodes was determined to be 28, the calculation itself for a year of interest can be done in parallel. This means that 2×28 nodes can be used, or 4×16 nodes for an even larger speedup in total calculation time.

Finally, it is demonstrated that user friendliness is a possibility for our cluster. The ability to customize the cluster for a specific case has been shown, which cannot be immediately done on e.g. the cluster of the VSC. Through web interfaces, an abstraction layer was provided, making the cluster accessible to users without knowledge about clusters or HPC in general. This way, the threshold regarding the usage of clusters in general can be lowered even for domain-specific problems.

Chapter 6

UC 2: Gravitational Lenses

A second use case can be found in the area of astrophysics. The performance of the cluster will be assessed through the work of dr. J. Liesenborgs on gravitational lenses [83]. Before doing so however, a basic understanding of the gravitational lens effect is required first.

6.1 Introduction

In the source plane, a distant astronomical object such as a star or galaxy emits light. The light ray travels through space to reach us, observers, on or nearby earth. However, it is possible that the light ray does not follow a completely straight path. Due to an intermediate massive object such as a black hole, galaxy or galaxy cluster, the light ray gets deflected or 'bent' by the gravitational field of the object. This object is then called a gravitational lens. For the observers, a gravitational lens has multiple implications. The object of interest is to be viewed in a different location than if the intermediate object was not present. Also, in case of a strong lens, the same light from the source can be bent to follow several paths. This results in multiple copies of the same object appearing for the viewer. These concepts are illustrated in 2D space in Figure 6.1.

With this background in mind, we can now understand the use cases that will be ran. First, suitable programs for benchmarking purposes will be selected. Then, the performance of our cluster will be assessed through benchmarks and be compared with the large Tier-II cluster of the Flemish Computer Centre.

6.2 Cases of Interest

The work of dr. Liesenborgs encompasses methods for gravitational lens inversion. This means that, for a given image captured by a telescope, calculate the mass of the lens and the way it is distributed. Instead of inversion, a simulation of the lens effect was

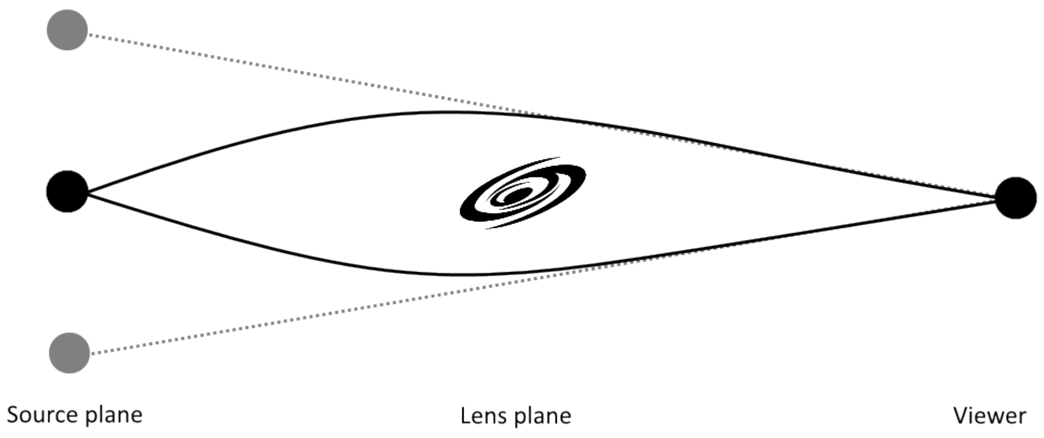


Figure 6.1: The gravitational lens effect. Two apparent objects can be observed from the viewer standpoint

researched as well, in order to view results of the inversion. For both cases, parallel implementations already exist by means of MPI. This section will discuss both cases.

6.2.1 Gravitational Lens Inversion

Given an observation of a lens by a telescope, what can be said about the mass distribution of the lens, based purely on the observed copies and distortions of objects? To answer this question, a genetic algorithm was developed by dr. Liesenborgs [83].

First, a set of random mass distributions is generated, called a population. A new population is created by combining and mutating the existing population, resulting in variations of the original. Each variation must get assigned a score. The ones that received a higher score should have more successors in the following generations. This process is repeated to further mutate the population in order to get better results. The evolution is guaranteed by the scoring mechanism by using the best 'chromosome' of the generation. This means that for each chromosome, the fitness must be calculated, which can be done in parallel since this process is independent from other fitness calculations.

For benchmarking purposes, it is absolutely necessary that the same random seed is used on the machines between which the comparison is made. Another thing to be considered is the difference in floating point accuracy between the ARM cores and x86 cores. Since the fitness is highly dependent on floating point numbers, small differences could result in other chromosomes being selected, which in turn results in more or fewer mass distribution generations being generated. Because the execution times of both benchmarking machines can therefore differ, this case was ultimately dropped as the results would not be representative.

6.2.2 Gravitational Lens Simulation

To view the results of the gravitational lens inversion, the lens has to be simulated. In other words, what is the effect of the gravitational lens on the light? The algorithm to do so, is a ray tracing algorithm. For each of the pixels of the result, the same calculation has to be done but on different data. Since each of these pixels can be calculated completely independent of each other, it is an embarrassingly parallel problem.

Implementation-wise, the ray tracing algorithm is made in a server-client fashion. The master MPI process represents the server, while the worker MPI processes are clients. Each worker process processes a set of pixel. It does so by requesting work from the server. When it is done, it reports its results back to the server and request a new task. This is repeated until no tasks are left.

This case proves to be an excellent way to benchmark the cluster, since the program will not produce different results based on the hardware where this was the case in the previous section. Furthermore, no sequential pre- or postprocessing takes place, making the entire code parallel.

6.3 Performance

For the benchmark, the SDSS J1004+4112 galaxy cluster lens data was selected and was made available by dr. Liesenborgs. The program was originally executed on the cluster of the VSC. Therefore, the performance of our cluster is compared to the Tier-II cluster of the VSC, ThinKing. The cluster has 208 nodes linked together with QDR Infiniband. Each node is equipped with two 10-core Intel Xeon E5-2680v2 Ivy Bridge CPUs clocked at 2.8 GHz. Another partition is available, featuring 144 nodes with two 12-core Intel Xeon E5-2680v3 Haswell CPUs clocked at 2.5 GHz and linked together with FDR Infiniband.

For the benchmark, a target image of 2048x2048 pixels is specified. On the VSC cluster, we opt to use the Ivy Bridge nodes since there are more of these nodes available, in an attempt to reduce queue times. Since the Ivy Bridge nodes have 20 cores per node, runs are executed in multiples of 20 cores for 1, 2, 4, 6, 8, 10 and 12 nodes respectively. The Odroid-cluster has analogous runs, albeit with different total number of CPUs. For each number of CPUs, five runs are performed. The results for both clusters are shown in Figure 6.2.

Clearly, this is an embarrassingly parallel problem. To gain a better understanding how the problem scales on both clusters, the speedups when increasing the number of cores were plotted in Figure 6.3 with baselines 20 and 64 cores for the VSC cluster and the Odroid-cluster respectively. 20 cores were chosen for ThinKing since those represent a single node, while 64 cores or 16 nodes were chosen for the Odroid-cluster, as this is a quarter of the cluster. Choosing a lower amount of cores, e.g. 20 as the ThinKing

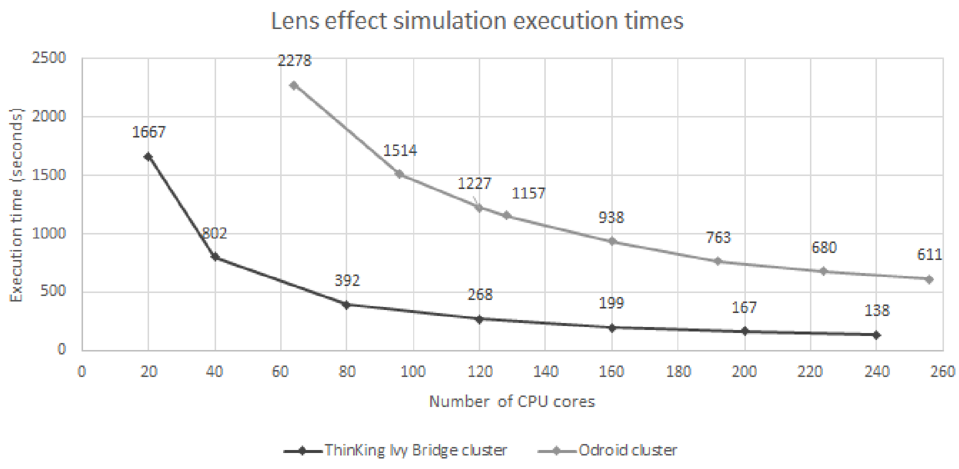


FIGURE 6.2: j1004: execution times for ThinKing and the Odroid-cluster

cluster, would make the execution times impractical for benchmarking. Interestingly, a *superlinear* speedup can be observed mainly for the ThinKing cluster. Doubling the number of CPU cores effectively more than halves the execution time. This raises questions however. How can the code execute more than double as fast when doubling the number of CPU cores, while network overhead is introduced?

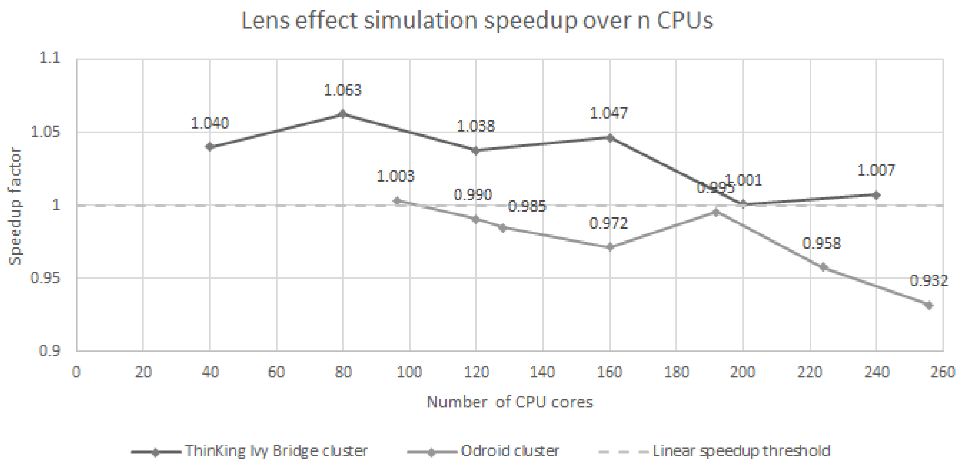


FIGURE 6.3: j1004: speedup factors for both clusters with baselines 20 and 64 cores for ThinKing and the Odroid-cluster respectively

The answer lies in the way the program is structured. As previously mentioned, a client-server system is used, meaning that 19 workers and one master are deployed for 20 cores. When doubling the number of cores, the resulting number of workers will be 39 while the master remains constant. The number of workers is thus $2n+1$ instead of $2n$ for the transition from 20 to 40 cores, causing the superlinear speedup. In general, the number of workers is $n-1$. Another thing that has to be considered is the choice of baseline. For ThinKing, a fraction of 1/20 cores is left idle due to the master process while the same fraction is 1/64 on the Odroid-cluster. Therefore, when scaling up, ThinKing will relatively have greater speedups than the Odroid-cluster since the latter already has more worker processes, causing the idleness of the master to be somewhat “filtered out”. Therefore, it is interesting to see how the program scales when looking at the workers only instead of the total number of cores, meaning $n - 1$. Those speedups are plotted in Figure 6.4.

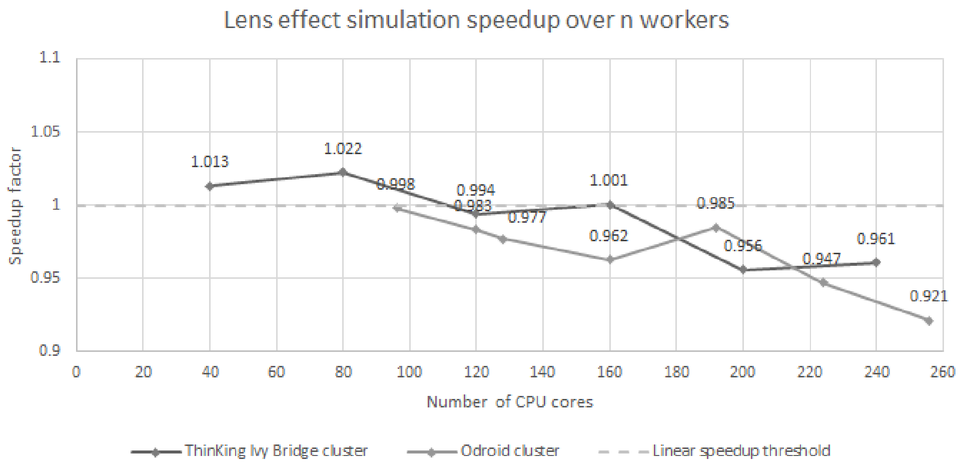


FIGURE 6.4: j1004: adjusted speedup factors for both clusters with respect to the number of worker processes

Indeed, the speedups are now mostly below the superlinear speedup threshold, as expected. Due to the choice of baseline, the speedups are initially still somewhat above the linear speedup threshold. It is possible that the number of cores for the baseline performed relatively worse than other numbers. Furthermore, both clusters scale very similarly. Local peaks are caused by the given number of cores. Since there is a constant portion of work to be done for 2048x2048 pixels, a certain amount of cores can better ‘fit’ into this number, explaining a local increase in speedup. Overall, the speedup slowly declines as the number of cores is increased due to more and more communication overhead. For the Odroid-cluster, we note standard deviations of the execution time to be 3.3 seconds for 64 cores, decreasing to 0.28 seconds for 256 cores. For 192 cores, an outlier is observed, peaking at 4.9 seconds. This can again be attributed to how the code works. For this number of cores, it is possible that some of them still receive a new portion work right before the total pool of work is exhausted.

Both clusters scale similarly, although the Odroid-cluster is, as expected when looking at the absolute execution times, slower. Yet, when we make a direct comparison at the 120-core mark, they differ a factor 4.6, which is *reasonable*. Although this result has to be nuanced: a factor of *only* 4.6 means that a job that would normally take a day to complete, would take over four and a half days to complete on our cluster.

With regards to the cost of both systems, it becomes clear why a factor 4.6 is indeed *reasonable*. The costs of the Odroid boards (including Micro SD cards) sum up to a total of €3000, excluding any of the power supplies, switches and other components that were used. On the contrary, a single node located in the cluster of the VSC typically costs

€5000-6000, again excluding any components besides the cost for the compute node itself. This means that our cluster costs only about as much as half a single ThinKing node, which only has 20 CPU cores compared to our 256. Naturally, this comparison is somewhat unfair. Each ThinKing node has, beside a very performing CPU compared to an ARM CPU, 64 or 128GB of RAM. The large amount of RAM allows it to tackle vastly larger, memory-intensive problems, which is impossible on our cluster. Furthermore, not only the costs should be directly compared. A more useful metric is the total cost of ownership, which is substantially larger for the ThinKing cluster.

Since this problem case also fully utilizes the cluster, it is interesting to measure the power draw of the system at full load. For the statistics use case, this was not an option since the cores were only pseudo-busy most of the time for 64 nodes. To gain insights in the power consumption, it is useful to make a comparison with another system. For this second system, we opt for the 4-core PC used earlier in use case 1, since we are unable to measure the power of a typical ThinKing node. The measurements are shown in Figure 6.5.

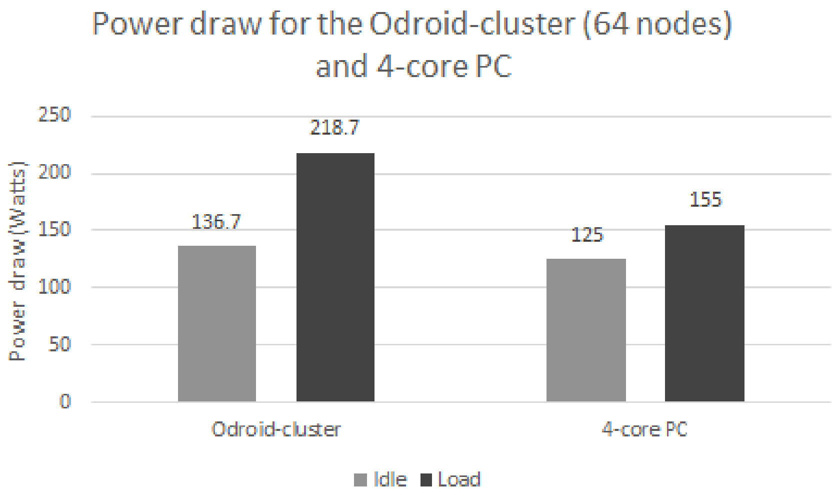


Figure 6.5: Power draw for the Odroid-cluster for 64 nodes and the 4-core PC when idle and under load

The power draw of the entire 64-node cluster when idle is slightly higher than that of the 4-core PC. This can be attributed to the frequencies of the CPUs of the cluster, which remain at 1.5GHz even when idle. In principle, they could be clocked down when not under load. Instead, an unused part of the cluster can simply be turned off, which is impossible for the PC. When both the cluster and PC are under load, their power draw increases by 82W and 30W respectively. This means that every Odroid board gets an increase of just 1.28W, totaling 3.42W per board. This difference is minimal compared to the increase of the PC.

Additionally, we can compare the cost of the power usage. At 218.7W and 611 seconds of execution, the Odroid-cluster uses 0.037kWh. It takes 6491 seconds to complete the program in case of the 4-core PC which, at 155W, results in 0.28kWh. This means that for the execution of the same program, when electricity is billed at 20 eurocents per kWh, using the Odroid-cluster would cost us 0.74 cents while the PC would cost us 5.6 cents. Unfortunately, we cannot include the ThinKing cluster in this comparison since we have no access to the power usage of this system, as mentioned before.

6.4 Conclusion

In this chapter, it was demonstrated that the performance of the cluster is very *reasonable* when compared to a full-fledged computer cluster. A difference factor of 4.6 in execution time was observed. Although the performance is quite good, the results obtained must absolutely be nuanced, as mentioned earlier. Small jobs that take 15 minutes at most on the ThinKing cluster would take over an hour on our cluster. Larger jobs taking multiple hours can then take up to a day – which is not ideal.

Conclusions about the scalability of the problem can be made by observing the speedups of the program plotted in Figure 6.4. The differences between the speedups of both systems are minimal, which can again be attributed to how the program operates. Also, the network can be of influence, since we are comparing QDR Infiniband with a gigabit ethernet network. In the end, we can ascertain that the ThinKing cluster scales very similar to the Odroid-cluster. Of course, for raw performance, a cluster such as ThinKing should still be used.

With regards to power usage, a comparison with the VSC cluster could not be made. Instead, a comparison was made with the benchmarking PC from the previous use case. We can conclude here that our cluster is significantly better in execution time and power usage. The reason for this is that the PC is limited to its four cores, making it unable to take advantage of the scalability of the program and resulting in a large execution time.

Chapter 7

Conclusions and Future Work

The goal of this thesis was to overcome some of the problems in HPC regarding access, multi-user problems, privacy concerns and usability. To do so, a private, in-house, low-cost computer cluster for HPC purposes intended for a single user was developed. Since the cluster has inexpensive SBCs as components, it is not necessarily intended to replace HPC systems but rather to be situated in-between existing HPC systems and the user's own notebook or desktop machine. Now that the system architecture and software components of the cluster have been discussed and the performance has been evaluated, it is now possible to formulate conclusions with respect to each of the research questions that were originally defined in Section 1.2, listed again below.

RQ 1. Is it possible to develop a computer cluster that

(a) consists of low-end single-board computers;

(b) remains inexpensive;

(c) maintains code portability, that is, code does not need to be drastically changed in order to run on the cluster.

The first research question was mainly covered in Chapter 4. In this chapter, it was shown that a cluster of SBCs for HPC purposes could indeed be constructed, similar in functionalities to a full-fledged computer cluster. At a price point of less than €3000, it can be considered to be very inexpensive when again compared to such a cluster. Lastly, as proven by both use cases, code does not require any platform-specific changes and remains portable.

RQ 2. If so, when compared to existing systems, does the SBC cluster

(a) scale well;

(b) deliver *reasonable* performance;

(c) have the ability to tackle diverse real-world problems.

This question was addressed in chapters 5 and 6. We can conclude that the cluster scales as well as a full-fledged cluster for problems that can be fully executed in parallel. A large parallel fraction is desired since the more sequential code is present, the more the advantage of our cluster is diminished when compared to a cluster consisting of x86 Xeon processors, since those CPUs have the upper edge there. The performance of the cluster is found to be reasonable when taking cost and power consumption into account. Although sequential execution of code could also be considered to be reasonable, in practice it can overrule the advantage of a parallel fraction of a program, which is best avoided as much as possible.

Moreover, since two domain-specific problems were discussed and used to evaluate the cluster, the cluster's ability to tackle real-world problems was demonstrated. In general however, a limiting factor for tackling larger HPC problems is not necessarily the number or speed of the CPU cores, but rather the amount of available RAM per core. Currently, each compute node has about 190MB RAM available per core, which is not a lot compared to a larger cluster, where typically gigabytes of RAM are allocated per core.

RQ 3. Can the cluster guarantee user friendliness for users even with a background different from IT, regarding

(a) viewing the cluster status and interpreting results;

(b) general usability and ease-of-use;

(c) the ability to support usability for domain-specific problems.

Chapter 4 described several ways to improve user friendliness as informally confirmed by users and thus lowering the cluster usage threshold, allowing less experienced users to utilize and monitor the cluster resources. It is shown that diverse interfaces can be deployed, which is not immediately an option for larger, non-private clusters with lots of users. Although interfaces can ease the use of such a system, it must be kept in mind that 'lower-level' alternatives such as an SSH session can still provide more flexibility.

Furthermore, in Chapter 5, it was shown that even users without any knowledge of the system can be supported for their domain-specific problems. This can go as far as from input to output, where only problem related input is expected and output can be visualized. Together with the other interfaces, the usage threshold for the cluster can again be lowered. However, do note that such a user interface is not always an option.

It takes time to develop and is only suited for programs that remain relatively static. Otherwise, the interface has to be adapted to the program each time it is changed, which is impractical.

Final Remarks With our SBC cluster in mind, the question "Why not build an equivalent cluster, consisting of fewer but more performing Intel Xeon CPUs?" could be asked. This would indeed solve the issue involving sequential parts in the code – but not at a price point similar to what we have achieved now. Besides the CPUs, other components such as motherboards and RAM have to be purchased as well. These components are already 'included' by default in an SBC, albeit very limited in features and size.

Finally, where can the SBC cluster be situated? We believe that the cluster is a successful attempt at bringing the power of parallelism to a broader public. It bridges the gap between direct user systems such as notebooks or desktop computers and actual computer clusters. It is a system complementary to both, and can be a help during the parallelization process. It does not replace high-end HPC machines, as it was never our intention to do so. If in the end raw performance is desired, such a high-end HPC system makes for a better option. Do note that, since sequential performance can be disappointing, another system can be used specifically for that part. Hence we have chosen the term 'complementary', which would give us the best of both the mainly sequential desktop system and the parallelism of a cluster, also shown in use case 1.

Future Work The cluster is currently intended for a sole user only, although multiple users could in principle share an account. Of course, this is not always ideal if users need to do their work privately. Therefore, the cluster could be adapted for a multi-user scenario. Doing so would require no architectural changes but mostly minor, user account related adjustments for which hints have already been given at the end of Chapter 4.

With regards to the user interfaces that were deployed, they could be improved to use the same styling. Currently, Force uses the Bootstrap 2 library for its CSS, while the interfaces that were newly created depend on Bootstrap 3. To support a uniform look for the cluster, Force should be upgraded to Bootstrap 3 and be stylized similarly. An exception has to be made for Ganglia, which does not really need to have the same look and feel as it can be considered to be somewhat 'separate' from the rest of the interfaces.

Furthermore, if another partition to the cluster were to be added or a new SBC cluster would be constructed, the Odroid-C1+ should be the minimal baseline for the boards that would be used. Although this suggestion is still relevant as of June 2016, a reevaluation of the available SBCs should best be made in the future since various new boards are continuously being released. As mentioned before, especially the available memory should receive attention. An upgrade to 2GB would allow for a lot more of RAM per core; a suggestion for such a board was already given in Section 4.3. Furthermore, having processors with more cores per board at a similar price point, e.g. octacore processors, would increase the cost-effectiveness of the cluster. However, do note that the memory should then be scaled up analogously to still respect the amount of RAM available per core.

Besides using the SBC cluster in an HPC context, it can perhaps be of interest to use it for purposes different from HPC as well. In Section 4.2, an example of such a different kind of cluster has already been briefly mentioned. There, a Raspberry Pi cluster was built to form a Docker cluster. Docker or containers in general were not explored in this thesis. In the future, deploying containers on the cluster could be an interesting topic.

Bibliography

[1] Gregory F. Pfister. *In Search of Clusters: The Coming Battle in Lowly Parallel Computing.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995. ISBN 0-13-437625-0.

[2] Mark Baker and Rajkumar Buyya. Cluster Computing: The Commodity Supercomputer. *Softw. Pract. Exper.*, 29(6):551–576, May 1999. ISSN 0038-0644. doi: 10.1002/(SICI)1097-024X(199905)29:6⟨551::AID-SPE248⟩3.0.CO;2-C. URL http://dx.doi.org/10.1002/(SICI)1097-024X(199905)29:6⟨551::AID-SPE248⟩3.0.CO;2-C.

[3] K. Hwang, J. Dongarra, and G.C. Fox. *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things.* Elsevier Science, 2013. ISBN 9780128002049. URL https://books.google.be/books?id=IjgVAgAAQBAJ.

[4] R.F. Freund, Michael Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J.D. Lima, F. Mirabile, L. Moore, B. Rust, and H.J. Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet. In *Heterogeneous Computing Workshop, 1998. (HCW 98) Proceedings. 1998 Seventh*, pages 184–199, Mar 1998. doi: 10.1109/HCW.1998.666558.

[5] Miron Livny, Jim Basney, Rajesh Raman, and Todd Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP journal*, 11(1):36–40, 1997.

[6] I. Raicu, I.T. Foster, and Yong Zhao. Many-task computing for grids and supercomputers. In *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, pages 1–11, Nov 2008. doi: 10.1109/MTAGS.2008.4777912.

[7] Various PlanetHPC experts. A Strategy for Research and Innovation through High Performance Computing. *PlanetHPC (supported under Objective "Computing Systems" of Challenge 3 "Components and Systems" of the ICT Programme of the European Commission)*, 2011.

[8] Constantinos Evangelinos and Chris N. Hill. Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. In *In The 1st Workshop on Cloud Computing and its Applications (CCA*, 2008.

[9] Victor Eijkhout. *Introduction to High Performance Scientific Computing*. lulu.com, 2014.

[10] D. Heggie and P. Hut. The Gravitational Million-Body Problem: A Multidisciplinary Approach to Star Cluster Dynamics. *Classical and Quantum Gravity*, 20 (20):4504, 2003. URL http://stacks.iop.org/0264-9381/20/i=20/a=603.

[11] Wikipedia. Molecular dynamics, December 2015. URL https://en.wikipedia.org/wiki/Molecular_dynamics. Last viewed on 19-12-2015.

[12] Wikipedia. Monte Carlo method, December 2015. URL https://en.wikipedia.org/wiki/Monte_Carlo_method. Last viewed on 09-12-2015.

[13] Niels Billen. Global Illumination, December 2014. Course slideset, not available online.

[14] Yongchao Liu, B. Schmidt, and D.L. Maskell. MSA-CUDA: Multiple Sequence Alignment on Graphics Processing Units with CUDA. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pages 121–128, July 2009. doi: 10.1109/ASAP.2009.14.

[15] Lusheng Wang and Tao Jiang. On the complexity of multiple sequence alignment. *Journal of computational biology*, 1(4):337–348, 1994.

[16] EMBL-EBI. Clustal Omega, December 2015. URL http://www.ebi.ac.uk/Tools/msa/clustalo/. Last viewed on 21-12-2015.

[17] Ken D Nguyen, Yi Pan, and Ge Nong. Parallel progressive multiple sequence alignment on reconfigurable meshes. *BMC genomics*, 12(5):1, 2011.

[18] Julie D Thompson, Desmond G Higgins, and Toby J Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic acids research*, 22(22):4673–4680, 1994.

[19] Mark Hachman. Intel Skylake: All the speeds, feeds, and prices, and which one is right for you, September 2015. URL http://www.pcworld.com/article/2976604/components/intel-skylake-all-the-speeds-feeds-and-prices-and-which-one-is-right-for-you.html. Last viewed on 02-12-2015.

[20] EMBL-EBI. The Economist explains, December 2015. URL http://www.economist.com/blogs/economist-explains/2015/04/economist-explains-17. Last viewed on 29-12-2015.

[21] Gabriel Mateescu, Wolfgang Gentzsch, and Calvin J. Ribbens. Hybrid Computing—Where HPC meets grid and Cloud Computing. *Future Generation Computer Systems*, 27(5):440 – 453, 2011. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2010.11.003. URL http://www.sciencedirect.com/science/article/pii/S0167739X1000213X.

[22] Pengfei Xuan, Jeffrey Denton, Pradip K. Srimani, Rong Ge, and Feng Luo. Big Data Analytics on Traditional HPC Infrastructure Using Two-level Storage. In *Proceedings of the 2015 International Workshop on Data-Intensive Scalable Computing Systems*, DISCS '15, pages 4:1–4:8, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3993-3. doi: 10.1145/2831244.2831253. URL http://doi.acm.org/10.1145/2831244.2831253.

[23] Gordon Bell and Jim Gray. What's Next in High-performance Computing? *Commun. ACM*, 45(2):91–95, February 2002. ISSN 0001-0782. doi: 10.1145/503124.503129. URL http://doi.acm.org/10.1145/503124.503129.

[24] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf: A Parallel Workstation For Scientific Computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[25] Paul E. Ceruzzi. *A History of Modern Computing*. MIT Press, Cambridge, MA, USA, 1998. ISBN 0-262-03255-4.

[26] Caryn Hannan. *Wisconsin Biographical Dictionary 2008-2009*. State History Publications, 2008. ISBN 9781878592637.

[27] Edwin D. Reilly. *Milestones in Computer Science and Information Technology*. Greenwood Publishing Group Inc., Westport, CT, USA, 2003. ISBN 1573565210.

[28] Rahul Garg. Exploring the Floating Point Performance of Modern ARM Processors, June 2013. URL http://www.anandtech.com/show/6971/exploring-the-floating-point-performance-of-modern-arm-processors. Last viewed on 10-04-2016.

[29] VSCentrum. The KU Leuven/UHasselt cluster (ThinKing and Cerebro), September 2014. URL https://www.vscentrum.be/infrastructure/hardware/hardware-kul. Last viewed on 17-04-2016.

[30] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier. High-performance computing: clusters, constellations, MPPs, and future directions. *Computing in Science Engineering*, 7(2):51–59, March 2005. ISSN 1521-9615. doi: 10.1109/MCSE.2005.34.

[31] N. Sadashiv and S.M.D. Kumar. Cluster, grid and cloud computing: A detailed comparison. In *Computer Science Education (ICCSE), 2011 6th International Conference on*, pages 477–482, Aug 2011. doi: 10.1109/ICCSE.2011.6028683.

[32] Rajkumar Buyya, Hai Jin, and Toni Cortes. Cluster computing. *Future Generation Computer Systems*, 18(3):v – viii, 2002. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/S0167-739X(01)00053-X. URL http://www.sciencedirect.com/science/article/pii/S0167739X0100053X. Cluster Computing.

[33] Addison Snell. Beyond Beowulf: Clusters, Cores, and a New Era of TOP500, December 2015. URL http://www.top500.org/blog/beyond-beowulf-clusters-cores-and-a-new-era-of-top500/. Last viewed on 29-12-2015.

[34] Scientific Computing World: HPC 2015-16. HPC evolves from the commodity cluster, 2016. URL http://www.scientific-computing.com/features/feature.php?feature_id=477. Last viewed on 13-06-2016.

[35] Top500. List statistics: Processor Generation, November 2015. URL http://www.top500.org/statistics/list/. Last viewed on 09-04-2016.

[36] I. Foster, Yong Zhao, I. Raicu, and Shiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, Nov 2008. doi: 10.1109/GCE.2008.4738445.

[37] P. Zaspel and M. Griebel. Massively Parallel Fluid Simulations on Amazon's HPC Cloud. In *Network Cloud Computing and Applications (NCCA), 2011 First International Symposium on*, pages 73–78, Nov 2011. doi: 10.1109/NCCA.2011.19.

[38] Blaise Barney. OpenMP, August 2015. URL https://computing.llnl.gov/tutorials/openMP/. Last viewed on 16-04-2016.

[39] Jay Hoeflinger. Cluster OpenMP for Intel Compilers, August 2010. URL https://software.intel.com/en-us/articles/cluster-openmp-for-intel-compilers. Last viewed on 16-04-2016.

[40] Christian Terboven, Dieter an Mey, Dirk Schmidl, and Marcus Wagner. *OpenMP in a New Era of Parallelism: 4th International Workshop, IWOMP 2008 West Lafayette, IN, USA, May 12-14, 2008 Proceedings*, chapter First Experiences with Intel Cluster OpenMP, pages 48–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-79561-2. doi: 10.1007/978-3-540-79561-2_5. URL http://dx.doi.org/10.1007/978-3-540-79561-2_5.

[41] Martin J. Chorley and David W. Walker. Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters. *Journal of Computational Science*, 1(3):168 – 174, 2010. ISSN 1877-7503. doi: http://dx.doi.org/10.1016/j.jocs.2010.05.001. URL http://www.sciencedirect.com/science/article/pii/S1877750310000396.

[42] Chao-Tung Yang, Chih-Lin Huang, and Cheng-Fang Lin. Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications*, 182(1):266 – 269, 2011. ISSN 0010-4655. doi: http://dx.doi.org/10.1016/j.cpc.2010.06.035. URL http://www.sciencedirect.com/science/article/pii/S0010465510002262. Computer Physics Communications Special Edition for Conference on Computational Physics Kaohsiung, Taiwan, Dec 15-19, 2009.

[43] Intel. Intel Many Integrated Core Architecture - Advanced, September 2011. URL http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html. Last viewed on 23-04-2016.

[44] ARM. ARM Holdings plc 2015 Analyst and Investor Day, September 2015. URL http://phx.corporate-ir.net/External.File?item=UGFyZW50SUQ9MzA0NTAyfENoaWxkSUQ9LTF8VHlwZT0z&t=1&cb=635778262245579577. Last viewed on 09-04-2016.

[45] Timothy Prickett Morgan. ARM Servers: Throwing Down The 25 Percent Share Gauntlet, January 2016. URL http://www.nextplatform.com/2016/01/06/arm-servers-throwing-down-the-25-percent-share-gauntlet/. Last viewed on 09-04-2016.

[46] Biju Dwarakanath. Chip designer ARM sees 20 pct market share in servers by 2020, March 2015. URL http://www.reuters.com/article/arm-servers-share-idUST8N0W500620150319. Last viewed on 09-04-2016.

[47] Thomas Petazzoni. ARM support in the Linux kernel, February 2013. URL https://archive.fosdem.org/2013/schedule/event/arm_in_the_linux_kernel/attachments/slides/273/export/events/attachments/arm_in_the_linux_kernel/slides/273/arm_support_kernel.pdf. Last viewed on 09-04-2016.

[48] Harel Kopelman. Secret Recipe for Raspberry Pi Server Cluster Revealed, July 2013. URL https://blogs.nvidia.com/blog/2013/07/19/secret-recipe-for-raspberry-pi-server-cluster-unleashed/. Last viewed on 04-04-2016.

[49] Bryan Wann. Tiny cluster, October 2013. URL https://binaryfury.wann.net/tinycluster/. Last viewed on 04-04-2016.

[50] Allen Alasdair. Build a Compact 4 Node Raspberry Pi Cluster, August 2015. URL http://makezine.com/projects/build-a-compact-4-node-raspberry-pi-cluster/. Last viewed on 04-04-2016.

[51] Jonas Widriksson. Raspberry PI Hadoop Cluster, October 2014. URL http://www.widriksson.com/raspberry-pi-hadoop-cluster/. Last viewed on 04-04-2016.

[52] Pieterjan Montens. Erlang: Pi2 ARM cluster vs Xeon VM, February 2016. URL https://medium.com/@pieterjan_m/erlang-pi2-arm-cluster-vs-xeon-vm-40871d35d356#.cbzh0llol. Last viewed on 04-04-2016.

[53] Michael Larabel. 8-Way ARM Board Linux Benchmark Comparison From The Pi Zero & ODROID To Tegra, January 2016. URL http://www.phoronix.com/scan.php?page=article&item=8way-arm-sbc&num=1. Last viewed on 04-04-2016.

[54] Hypriot. Introducing Hypriot Cluster Lab: Docker clustering as easy as it gets, December 2015. URL http://blog.hypriot.com/post/introducing-hypriot-cluster-lab-docker-clustering-as-easy-as-it-gets/. Last viewed on 14-05-2016.

[55] Simon J. Cox, James T. Cox, Richard P. Boardman, Steven J. Johnston, Mark Scott, and Neil S. O'Brien. Iridis-pi: a low-cost, compact demonstration cluster. *Cluster Computing*, 17(2):349–358, 2013. ISSN 1573-7543. doi: 10.1007/s10586-013-0282-7. URL http://dx.doi.org/10.1007/s10586-013-0282-7.

[56] Nikilesh Balakrishnan. Building and benchmarking a low power ARM cluster, August 2012. URL https://static.ph.ed.ac.uk/dissertations/hpc-msc/2011-2012/Submission-1126390.pdf. Last viewed on 04-04-2016.

[57] Josha Kiepert. RPiCluster, May 2013. URL http://coen.boisestate.edu/ece/files/2013/05/Creating.a.Raspberry.Pi-Based.Beowulf.Cluster_v2.pdf. Last viewed on 04-04-2016.

[58] David Toth. Dave Toth's Portable Compute Cluster Page, February 2016. URL http://web.centre.edu/david.toth/portablecluster/index.html. Last viewed on 04-04-2016.

[59] Jack Dongarra. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, February 2016. URL http://www.netlib.org/benchmark/hpl/. Last viewed on 28-05-2016.

[60] Olimex. A20-OLinuXIno-LIME2, March 2016. URL https://www.olimex.com/Products/OLinuXino/A20/A20-OLinuXIno-LIME2/open-source-hardware. Last viewed on 28-03-2016.

[61] Hardkernel. ODROID-C1+, August 2015. URL http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143703355573. Last viewed on 28-03-2016.

[62] BeagleBoard. BeagleBone Black, March 2016. URL https://beagleboard.org/black. Last viewed on 28-03-2016.

[63] Xunlong. Orange Pi Plus, March 2016. URL http://www.orangepi.org/. Last viewed on 28-03-2016.

[64] Sinovoip. banana pi BPI-M2, May 2015. URL http://www.banana-pi.com/eacp_view.asp?id=71. Last viewed on 28-03-2016.

[65] ODROID Forum. Minimal Debian Wheezy/Jessie images, May 2015. URL http://forum.odroid.com/viewtopic.php?f=114&t=8084. Last viewed on 04-04-2016.

[66] Adaptive Computing. Torque Resource Manager, April 2016. URL http://www.adaptivecomputing.com/products/open-source/torque/. Last viewed on 04-04-2016.

[67] SchedMD. Slurm Workload Manager, November 2013. URL http://slurm.schedmd.com/. Last viewed on 04-04-2016.

[68] Adaptive Computing. Moab Lite, April 2016. URL http://www.adaptivecomputing.com/products/hpc-products/moab-lite/. Last viewed on 12-04-2016.

[69] Adaptive Computing. Maui Cluster Scheduler, April 2016. URL http://www.adaptivecomputing.com/products/open-source/maui/. Last viewed on 12-04-2016.

[70] The Open MPI Project. Open MPI: Open Source High Performance Computing, January 2016. URL https://www.open-mpi.org/. Last viewed on 17-04-2016.

[71] MPICH. MPICH: High-Performance Portable MPI, January 2016. URL http://www.mpich.org/. Last viewed on 17-04-2016.

[72] Ganglia. Ganglia Monitoring System, April 2015. URL http://ganglia.info/. Last viewed on 04-04-2016.

[73] collectd. collectd – The system statistics collection daemon, May 2015. URL https://collectd.org/. Last viewed on 13-04-2016.

[74] The Cacti Group. About Cacti, February 2016. URL http://www.cacti.net/. Last viewed on 13-04-2016.

[75] Nagios. Project Overview, April 2016. URL https://www.nagios.org/projects/. Last viewed on 13-04-2016.

[76] Tobias Oetiker. About RRDtool, September 2014. URL http://oss.oetiker.ch/rrdtool/. Last viewed on 12-04-2016.

[77] Matteo Ragni. Force, July 2013. URL https://github.com/MatteoRagni/Force. Last viewed on 26-04-2016.

[78] SURFsara. Job Monarch, January 2014. URL https://oss.trac.surfsara.nl/jobmonarch. Last viewed on 24-04-2016.

[79] N. Hens, S. Abrams, E. Santermans, H. Theeten, N. Goeyvaerts, T. Lernout, E. Leuridan, K. Van Kerckhove, H. Goossens, P. Van Damme, and P. Beutels. Assessing the risk of measles resurgence in a highly vaccinated population: Belgium anno 2013. *Euro Surveill.*, 20(1), 2015. doi: http://dx.doi.org/10.2807/1560-7917.ES2015.20.1.20998. URL http://www.eurosurveillance.org/ViewArticle.aspx?ArticleId=20998.

[80] R Core Team. Writing R Extensions, May 2016. URL https://cran.r-project.org/doc/manuals/r-release/R-exts.html#OpenMP-support. Last viewed on 28-05-2016.

[81] N. Matloff. *Parallel Computing for Data Science: With Examples in R, C++ and CUDA*. Chapman & Hall/CRC The R Series. CRC Press, 2015. ISBN 9781466587038. URL https://books.google.be/books?id=SsbECQAAQBAJ.

[82] Simon Wood. Parallel computation in mgcv. URL https://stat.ethz.ch/R-manual/R-devel/library/mgcv/html/mgcv-parallel.html. Last viewed on 30-04-2016.

[83] Jori Liesenborgs. *Genetic algorithms for the non-parametric inversion of gravitational lenses.* PhD thesis, Hasselt University, Hasselt, 2010.